# Compilation-based Performance Improvement for Generative Planners

Julie Porteous
Department of Computer Science
City University

December 1993

This thesis is submitted as part of the requirements for a Ph.D. in Computer Science, in the Department of Computer Science of City University, London, England.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

This work was supported by a Co-operative Award in Science and Engineering (CASE) jointly funded by SERC and The Civil Aviation Authority and my thanks go to all at The CAA who were involved and in particular to Steve Wilmott, Chris Metcalf, Alastair Godbold and Paul Houselander.

I'd like to thank Sara Jones, Dave Bolton and Michael Jampel, colleagues at The City University, for enthusiastic reading of drafts of this thesis and valuable discussions about my work. Also, I am grateful to Nigel Mitcham and Andy Whitcroft for much help with technical difficulties.

I am indebted to my supervisor Lee McCluskey who was a constant source of encouragement throughout my time at City University. He was always available to offer advice and found time to painstakingly read and comment on the entire thesis.

Finally, I would like to thank my family, friends and especially my partner Michael for all their support.

# Declaration

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

# Abstract

When input a model of a planning domain and a problem to be solved, generative planning systems endeavour to generate a plan that is a solution to the input problem. There is a trade-off between the generality and efficiency of these systems which means that general planners tend to be inefficient. Our research has focussed on improving the efficiency of general planners. A popular approach to tackling this "efficiency versus generality" problem has been to embed systems with on-line and post-event learning components that use current and previous planning experience to increase subsequent performance. However, in this thesis we argue for a move away from this approach towards what we call off-line compilation, where all processing is independent of on-line planning. In support of this we have created two novel complementary techniques for off-line compilation, which we call PRECEDE and PROPOSE. In the course of our work they have been precisely defined, implemented and evaluated, and all aspects of this work are described in the thesis.

PRECEDE is a method for generating goal orders that are based on necessary interactions between goals. PROPOSE is a method for generating iterative macro operators for all *sorts* of objects in a modelled planning domain. It also generates sequences of constants that can be used to unwind the iterative parts of the macros. The PRECEDE goal orders can be used to select goals to achieve next and the PROPOSE iterative macros and enumerated sequences can be used to select ground operators for goal achievement during plan generation. These methods are off-line in the sense that all processing is performed a priori direct from the model of a planning domain.

The compilation methods were evaluated statically, empirically and in comparison with established methods of on-line and post-event learning. For the empirical testing of off-line compilation an example planner was selected and extended so that it could use domain models that had been compiled by PRECEDE and PROPOSE. This planner was then used in a series of empirical tests which revealed that the techniques, both individually and in combination, consistently improve the performance of the planner (a speed-up of between 2 and 5 times faster across a sample of planning domains). Also, the results of static and dynamic comparison with established methods show that off-line compilation with PRECEDE and PROPOSE compares favourably with existing post-event and on-line learning methods.

# Chapter 1

# Introduction

The problem of automating the process of plan generation has been an active area of research within Artificial Intelligence for most of the last three decades. In its broadest sense this process, referred to as planning, involves generating a plan which is some combination of actions that change the state of a modelled planning domain. The aim is that the plan will change the initial state of the planning domain into one that achieves some desired goal. This is a process that we perform frequently, for example, a trip to the shops can be preceded by generating a plan to go first to the bank to get some money before going to the shop to buy some groceries. The actions that are performed and the order that they are performed in are important since for example we must get the money before we can buy the groceries. With the earliest planning systems, in the form of GPS [58] and STRIPS [22], the target was to develop general methods for generating plans (or solving problems) that were domain independent. A plan generator could be seen like a "black box" that could reason about any domain, provided that the domain could be expressed in the language that the planner accepted.

Unfortunately, reasoning about actions, a central task of the plan generator, is computationally very expensive. This is largely the result of the number of combinations of actions that must be considered which increase with the number of actions that are combined in a plan. This is exacerbated by demands that may be placed on the planner, such as requiring it to guarantee that it can always generate a solution plan if a solution exists. Indeed, there has been a gradual realisation that it would never be possible to find a general solution to the planning problem that could solve all problems in reasonable time. This was confirmed in the form of research results which showed the general complexity of planning [9], [6].

The domain independent planning paradigm is still popular but there has been a change of emphasis towards finding ways of improving the efficiency of general planning systems in certain situations. A popular approach has been to embed the planning system with learning components that use past or current planning experience to improve subsequent planning performance. For example, this approach is an important feature of the PRODIGY system [8], where past and current planning experience is used to generate search control rules. Our work represents a move away from these approaches, which we term "post-event" or "on-line", towards a priori or "off-line" compilation direct from the domain model.

At the heart of this thesis is the argument that for certain types of domain model, off-line compilation holds the key to making planning systems more efficient. An assumption of these types of domain model is that they are "complete", in the sense that they contain a representation of all aspects of the domain that will be required to generate plans in the domain. As part of our thesis work, we have created two distinct but complementary new compilation methods, called PRECEDE and PROPOSE, that are examples of off-line compilation which compare favourably to established post-event and on-line learning methods.

In this introductory chapter our intention is to set the scene for the work that is presented in the thesis. Hence, we start with a brief introduction to planning and the planning systems that we are interested in. This is followed by an introduction to compilation for planning and the relationship between off-line compilation and established methods of post-event and on-line learning. We also give a brief introduction to the novel compilation methods PRECEDE and PROPOSE that we have created (sections 1.3 and 1.4). In section 1.5 we summarise what we feel to be the major contributions of the thesis and finally, we give an overview to the way that the thesis is organised (section 1.6).

## 1.1 Planning

We shall be viewing automated planning as a process which is performed by *planning systems* or planners for short. The input to a planning system is a *planning problem* and a *domain model*. The planning problem is specified in terms of an initial state, which is the state to start planning from, and a goal condition, that describes the desired goal state for the plan to achieve. The domain model is a symbolic representation of some planning domain which comprises a set of operators that represent how the state of the modelled planning domain can be changed, and additional environmental information about the domain. The output of a planning system is a solution to some input planning problem. The output language of the planning system prescribes the form of the solution to a given planning problem.

Some well known examples of systems in the class that we are interested include STRIPS [22], NOAH [63] and TWEAK [9]. The distinguishing features of these systems are that they are *Generative* and *Domain Independent*. They are generative in the sense that they generate entire plans before executing any of the actions in them and they are domain independent in that the part of the planning system that generates the plan, the *plan generation engine*, is logically separate from the model of the planning domain, the *domain model*, that the plan generation engine reasons about. This class of planners also make a number of assumptions. They assume that input domain models are complete (all aspects of the domain that are required to generate plans are represented). They assume a *closed world*, where all things not explicitly known to be true are assumed false. Also, they make the STRIPS assumption of *default persistence* where things that are not explicitly changed are assumed to remain unchanged.

## 1.2 Compilation for Planning

We define compilation for planning as the transformation of knowledge about a planning domain, in other words the domain model, into a form that can be more efficiently reasoned about and used by a particular plan generation engine. Within this framework, we are able to view many established methods of machine learning, which use domain independent planning systems as performance components, as compilation. For example, EBL (which stands for explanation based learning) is a post-event learning method that can be used to learn macro operators [65]. We say that this method is post-event because it uses example planning sessions for learning.

In one sense, nothing "new" is learned by this process of post-event learning (or compilation), except perhaps a bias towards the example problems, since everything was derived from the original domain model. This suggests that any "knowledge" that can be compiled post-event can be compiled a priori direct from the domain model. This is the type of compilation that we are interested in, where the compilation is "off-line" in the sense that it is totally independent of actual planning sessions. Our contention is that for complete domain models, where every problem has an equal likelihood of occurring, that the use of example planning problems for post-event compilation can unduly bias the compilation towards certain problem distributions, namely, the problem distributions that the example problems were drawn from.

We also see that compilation can help the task of modelling a particular planning domain. Planning domains tend to be represented declaratively rather than procedurally, since this eases the task of the domain modeller. Unfortunately, an input language that is well suited to the needs of a domain modeller is not necessarily the best format for the planning system that it is input to. Indeed, the inefficiency of declarative representations is one of the reasons for the general inefficiency of DIP's. This "mismatch" between what is best for the modeller and what is best for the plan generation engine is exacerbated because different modellers can encode the model differently and slight differences in representation can cause considerable changes in the performance of the planning system. We argue that an off-line compiler can transform the domain model, encoded in as natural a format as possible by the domain modeller, into a form that can be more efficiently reasoned about by the plan generation engine. We also argue that the off-line compiler can be used to check the consistency of modelled planning domains.

## 1.3 Compilation with PRECEDE

During plan generation an "efficient" planning system faces a number of choices, including the choice of which outstanding goal to achieve next. The specific motivation for PRECEDE was to develop an off-line method of compiling information that would be useful when choosing which goal to achieve next. PRECEDE is a novel method of compilation that compiles *goal orders* direct from the model of a planning domain. It uses information about inconsistent states in a modelled planning domain, which can be either supplied by the domain modeller or automatically generated from the domain model. If certain combinations of goals are found to be inconsistent then PRECEDE orders them with respect to one another. This uses a condition which we call $\mathit{necessary\_before}(x, y)$ which orders the goals $x$ and $y$ so that $x$ must be achieved before $y$ during planning. The compiled goal order that is output is an accumulation of pairs of necessary before orderings.

The goal orders are incorporated into the domain model and can then be used by a planning system to help select which goal to achieve next during plan generation.

In empirical tests we found that PRECEDE goal orders were useful in improving the performance of an example generative planning system.

One important advantage of PRECEDE compilation is that it does not suffer from the same problem of "collapse" that can limit the utility of other methods of ordering goals. This problem occurs when no partial order of goals can be found for a particular modelled planning domain. PRECEDE does not suffer from this problem because the goal orders are based on necessary interactions.

## 1.4 Compilation with PROPOSE

Another choice which occurs during plan generation is the choice of which operator (and binding) to use to achieve a planning goal. The motivation for PROPOSE was the need for an off-line method of compiling information that would be useful when making this choice. PROPOSE uses information about the sorts of objects in a domain model to produce iterative macro operators. Each sort, for example *robot*, *box* and *room* in a STRIPS [22] robot planning domain, is analysed by considering the possible situations that objects of that sort can occupy. The iterative macro-operators that are generated form abstract solutions which span all of the possible start and end situations for the sorts of objects in the domain model.

PROPOSE also produces sets of enumerated sequences which can be used to "unwind" iterative components of the macros. For example, if an iterative macro for the robot planning domain consists of a sequence of operators that repeatedly move the *robot* from one *room* to another then enumerated sequences are compiled for every possible start and end *room* within the domain model.

The advantage that PROPOSE has over post-event acquisition of macro operators (for example, with EBL in [67]) is its "coverage". The use of sort abstraction means that macros can be compiled for every possible start and end situation for objects of each sort in the domain model. An important consequence is that the iterative macros are not biased towards particular problem distributions.

## 1.5 Contributions

Overall, this thesis contributes to a shift away from post-event and on-line learning towards off-line compilation as a way to improve the performance of generative domain independent planning systems that reason about complete domain models. It is our contention, that for complete domain models, post-event learning adds nothing to the domain model that is not already inherent in the complete domain theory and hence can be compiled direct from the domain model. We summarise the main contributions of this thesis as follows:

- we have identified a useful condition for goal ordering. This condition, which we refer to as *necessary_before*$(x, y)$, orders two predicates $x$ and $y$ with respect to each other so that $x$ must be achieved before $y$ in goal directed planning. The condition is based on necessary interactions between predicates in the domain. A global partial order over domain predicates can be constructed by the accumulation of pairs of predicates for which *necessary_before* is true. An approach to automatically compiling PRECEDE goal orders has been developed which is encapsulated in a procedure which we refer to as *precede*.

- we have identified a useful level of abstraction for the compilation of iterative macro operators. We used the *sort* of objects in a domain model as a method to decompose the domain so that macros could be compiled for every possible start and end situation for that object. We have developed a method for compiling PROPOSE macros and enumerated sequences which is encapsulated in a procedure which we refer to as *propose*.
- we have shown how a domain independent planning system can be extended so that it can reason about domain models that have been compiled, either individually or in combination, by PRECEDE and PROPOSE.
- we have shown empirically that PRECEDE and PROPOSE compiled domain models can improve the performance of a domain independent planner. The results of our empirical tests also showed that PRECEDE and PROPOSE are complementary. For example, the combined compilation techniques improved the performance of the basic planner by between 2 and 5 times (in speed-up terms) across a sample of different planning domains. Also they have been shown to compare favourably with established methods of on-line and post-event compilation.

## 1.6 Organisation of the Thesis

The chapters in the thesis fall into the following broad sections, chapters 2 and 3 review research in planning and compilation for planning, chapters 4 and 5 focus on the novel compilation methods that we have developed, chapters 6 and 7 evaluate the effect on an example planning system of off-line compilation, chapter 8 relates our work to other work in the area and finally in chapter 9 we present our conclusions.

In chapter 2 we describe the class of planners that we are interested in and present a model of planning that will be used throughout the thesis. We also introduce an example planning architecture (the FM system [47, 46]) which we shall use later for illustration. Within the model of planning that we develop, the main way that planning systems differ is in the strategies they use to "specialise" partially developed plans during plan generation. In this chapter we review different strategies that have been developed to help at certain choice points that occur during plan generation and we discuss selected classical planning systems in terms of these different strategies.

The second of the introductory chapters (chapter 3) focuses on compilation for planning. In this chapter we define the term and introduce a model of compilation for planning that we shall be using as a framework. In the remainder of the chapter we review work from the literature on generative planning and machine learning and recast it as compilation for planning.

The main part of our thesis work is presented in chapters 4 and 5 when we discuss the compilation methods PRECEDE and PROPOSE that were developed. In chapter 4 we precisely define the compilation method PRECEDE and outline an implementation which compiles a domain model by generating a PRECEDE goal order. We also statically evaluate the partial orders that are output and show the correspondence between this compilation method and other comparable methods.

In chapter 5 we turn our attention to the second of the new compilation methods that we have developed, namely PROPOSE. There are two parts to the output of PROPOSE, a set of macros for different sorts in some modelled planning domain and a set of enumerated sequences that can be used to unwind iterative parts of the macros. The compiled macros and enumerated sequences are precisely defined in this chapter and an outline is given of an implementation that generates them. Finally, we statically evaluate the PROPOSE macros and the enumerated sequences.

The third main division of the thesis is in chapters 6 and chapter 7 where we empirically evaluate the use of PRECEDE and PROPOSE compilation. To this end we have selected an example planner from the class that we are interested in and amended its plan generation algorithm so that it accepts as part of its input, domain models that have been compiled by PRECEDE and PROPOSE. In chapter 6 we outline the example planner in some detail and consider how the algorithm is changed to exploit the compiled domain models. Then in chapter 7 we discuss the empirical testing of compilation with PRECEDE and PROPOSE. The results of the tests are presented and discussed.

In chapter 8 we evaluate the compilation methods by exploring their relation to other work in the area. As part of this we also consider the scope of the two new compilation methods and some useful extensions that would further exploit the potential power of the compilation.

The final chapter of thesis is the conclusions in chapter 9 where we summarise the thesis and the work that we have presented in support of it. At the end of this chapter we also discuss future work which will build on the work in the thesis.

At the end of the thesis, in appendices A, B and C we list the example modelled planning domains that we have used as illustrations throughout the thesis. There are three modelled planning domains, a robot planning world $SW$, a two robot variation of it called $SW^2$ and a warehouse planning world. All references that are used in the text are listed in the bibliography which begins on page 229.

# Chapter 2

# Planning

The design and implementation of automatic planning systems (referred to as planning systems or planners for short) has been the focus of much work over the past 25 years or so and during that time the term has come to cover a wide area of research activity. For example, planning systems are often classified as either *generative* (also referred to as *classical* planners) or *reactive*. Simply put, the difference between them is that generative planners formulate entire plans based on the initial state of the planning domain and then execute the plan, whereas reactive planners select the next action based on the current state of the planning domain, execute it and then select the next action based on the new state of the planning domain.

It is because the term planning has come to cover a range of approaches that we start this chapter by precisely describing the class of planning systems that we are interested in. Firstly, we give a general introduction to these planning systems (section 2.1) and then we formalise planner action (section 2.2). Within this framework planning systems can differ in the strategies they use to select actions and place them in a plan and these are discussed in section 2.3. The end of the chapter looks at the problems of planning, considers different solutions and introduces the approach that we have taken in our work.

## 2.1 Generative Domain Independent Planning

We shall call the class of planning systems that we are interested in *Generative Domain Independent Planners* (DIP). They are generative in the sense that they generate entire plans before any action in them can be executed and they are domain independent in that the part of the planning system that generates the plan, the *plan generation engine*, is logically separate from the encoding of the planning domain, the *domain model*. This generative domain independent paradigm has received continued interest from the earliest planning systems such as GPS [58] and STRIPS [22] through to more recent systems such as PRODIGY [8] and O-Plan [13].

One reason for the continued popularity of this approach to the design of automated planning systems is because the logical separation between the plan generator and the model of a planning domain allows the plan generator to be seen as a "black box" that can reason about any domain, provided that the domain can be expressed in the language that is accepted by the planner. In the rest of this section we shall consider domain independent planning systems in terms of some general features of their input and outputs.

There are two main components of the input to a DIP, a *planning problem* and a model of some planning domain. We assume that planning problems are specified in terms of an initial state $I$, which is the state to start planning from and a goal condition $G$, which describes what must be true in the modelled planning domain after the operators in the solution plan have been executed. The domain model is an encoding of the planning domain which is comprised of a set of operator representations that model "actions" in the planning domain in terms of state changes within the model of the domain. The domain model will also contain information about unchanging features of the planning domain.

As an example, consider a robot planning domain like the one used by Sacerdoti in [62]. This consists of a series of interconnecting rooms where a robot can open and close doors and move boxes to different locations in different rooms. The model of this planning domain contains operators such as $pushthrudoor(Box, Door, Room)$. This operator moves $Box$ from one room, via the $Door$, to $Room$ (which is some room adjacent to the original room). The domain model will also contain information about unchanging relations in the domain such as which rooms are connected to each other and which doors connect them. It could also include information about the "type" of objects in the domain, for example, that $box1$ is of type $box$.

It is desirable for a DIP to have associated with it a well defined, declarative input language, within which the planning problem and the domain model can be specified. This can help, for example, when validating the domain model and when checking the consistency of the initial state of some input planning problem. For many DIP's this input language is based around some variation of 1st order logic where the user encodes a high level declarative representation of the domain without expressing how it will be used (for example FM [45], PRODIGY [54]). An example of part of an encoding in such an input language can be found in appendix A starting on page 141. This shows a set of operators and environmental information for a robot planning world taken from [45].

The output of a DIP is a solution to an input planning problem. The output language of the DIP prescribes the form of the solution to a given planning problem. The solution may be:

- a total ordering of instantiated operator as in STRIPS [22] and MEA [46]
- a partial order over operators as in TWEAK [9] and NLP [45]
- it may involve iterative or conditional operators as in the NOAH system [63]
- it may involve operations overlapping in time as shown in [75] and [3]

The expressiveness of the input and output languages of a DIP indicate the degree of domain independence of the planner. If these languages are too restricted then a natural encoding of many planning domains may be impossible.

## 2.2 Model of Planning

Recently, there has been growth in work on the mathematical foundations of generative domain independent planning. This has been spurred on, in part, by the formalisation of old planning frameworks such as Lifschitz' work on the STRIPS system [42] and Chapmans work with TWEAK [9]. Formalisation has a number of advantages, for instance it can help facilitate replication and the correct implementation of planning algorithms. It has also been incorporated in research work in the computational complexity of planning such as [41, 6].

The formal model of planning developed by Lifschitz has been widely used as a basis for planning work. For example, it was used in the definition of planning given in [36] and [23], and as point of comparison in [2]. But this framework, although capturing the representation of the planning system does not capture the way in which a developing plan is represented. For this reason, we have loosely based our planning model on Christensen and Groves' model for classical planning [11] which is powerful enough to capture the class of DIP that we are interested in, the different strategies that are used during plan development and the structure of partially developed plans. The model provides a framework within which DIP's can be expressed and studied.

In this section we shall define the model of planning that will be used through the rest of the thesis. We start by looking at the way that states, operators and problems are represented and then consider how partially developed plans are represented during plan generation. Throughout the discussion we will use *FM* to illustrate the model. This is an integrated planning and learning system which allows users to "mix and match" different plan generation engines, learning components and modelled planning domains [47, 46]. The effects of a number of learning methods on the planning system have been explored and are reported in for example [48]. Also for illustration we will be using three example domain models which are described in full in appendices A, B and C.

### 2.2.1 State Representation

Attached to the model of a planning domain is a set of predicates that represent different features of the planning domain. For some of these predicates their truth value depends on the current state of the modelled planning domain and we refer to these predicates as *achievable* (the set of achievable predicates *preds* is defined on page 15). All the other predicates are assumed to be always true and we refer to these as *static*.

Within our planning model, states are represented as sets of achievable *ground* predicates, where the arguments of ground predicates are bound to objects of the correct sort (the set of ground predicates $preds_g$ is defined on page 15). A state representation assigns a truth value to every achievable ground predicate in the domain. This assignment is completed under the *closed world assumption* where every ground predicate not in the state description is assumed to be false. This is a feature of planners in the class of DIP's that we are interested in, such as FM (as we shall see in the example below) and PRODIGY [54, pp 77].

For any planning domain, within our model of planning, there is a finite set of states that describe snapshots of situations in that domain. We refer to this set as $SS$.

**Example: state representation in FM**

As an example, figure 2.1 depicts the warehouse planning world, $WW$, which consists of a series of locations, a truck, driver, a number of crates and so on.

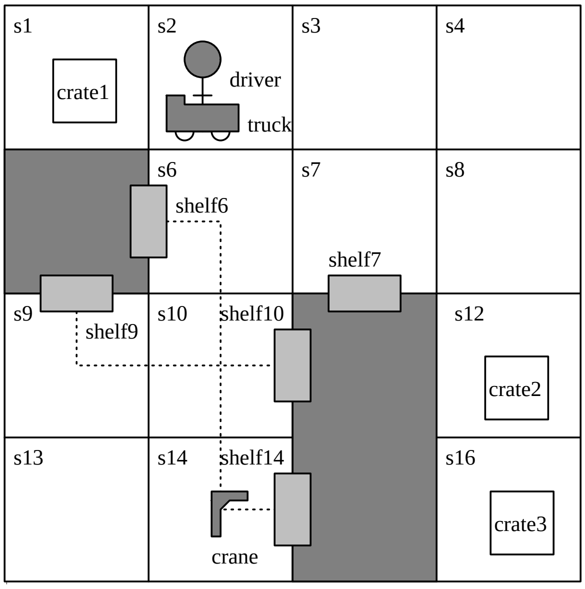


Figure 2.1: The warehouse planning world

The state of the world that is shown in the figure has the driver in the truck at location $s2$, $crate1$ is on the floor at location $s1$ and so on. This state is represented, in the FM specification language, as a set of predicates as follows:

$$\{truck\_at(s2), driver\_at(s2), on\_floor(crate3, s16), on\_floor(crate2, s12),$$
$$on\_floor(crate1, s1), crane\_at(s14), clear(s9), clear(s3), clear(s8), clear(s4),$$
$$clear(s6), clear(s7), clear(s10), clear(s14), clear(s13), in\_truck\}$$

Truth values of achievable predicates are determined simply by membership of the state description. So for example, the predicate $on\_floor(crate3, s16)$ is *true* since it is contained in the state description, but the predicate $on\_floor(crate3, s3)$ is false since it is not contained in the state description.

For FM the set of static predicates are supplied as part of the environmental information in a set that we refer to as $ES$. This set $ES$ shall be used to refer to environmental information in FM throughout the thesis. As an example, $ES$ for the warehouse planning world, $WW$, is partially specified below and listed in full in appendix C.2.

$$ES = \{pickup\_point(s6), pickup\_point(s9), \ldots, connect(shelf6, s6),$$
$$connect(s10, s6), connect(s6, s10), \ldots, next(s1, s2), next(s2, s1), \ldots$$
$$between(s7, s6, shelf6), between(s3, s7, shelf7), between(s13, s9, shelf9)\}$$

which reflects the fact that for instance location $s1$ is always "next" to location $s2$, $shelf6$ is always "between" location $s7$ and $s6$ and so on. The environmental information $ES$ also contains rules which specify the relation between certain predicates in the domain. These rules are expressed as pairs $(a, b)$ where $b$ can be inferred from $a$.

So for the warehouse planning world the rules that relate achievable predicates in the domain model include:

$$(truck\_loaded(C,S), truck\_at(C,S))$$

from which we can infer that if the truck is loaded at location $S$ then it is also at location $S$. There are also a number of rules that relate unchanging predicates in the domain model. For the warehouse world this includes the following:

$$(next(X,Y), next(Y,X))$$

from which we can infer that if location $X$ is *next* to location $Y$, then location $Y$ is also *next* to location $X$.

### 2.2.2 Operator Representation

For a given planning domain we assume that the domain modeller supplies, as part of the model of that domain, a set of operator schema that represent all the actions that can be performed in that domain. We shall refer to this set, throughout the thesis, as $OS$. In this section we shall look at four features of operator representation within our model. These are, firstly, the representation of the effects of an operator, secondly, the special operators that are used to represent the initial state and goal condition of an input planning problem, thirdly, the way that variables within operators can be bound and finally some important properties of operators. We shall look at each of these in turn and in each case we will illustrate the model with a concrete example from the FM planning system.

#### 2.2.2.1 Representing the Effects of Operators

An important problem in operator representation is how to represent the effect that an operator has on the state in which it is applied. One possibility is to represent an operator as a set of axioms which specify the effects of the operator. This requires an axiom for everything that the operator changes and it also requires an axiom for everything that is not changed. However this can be a problem since the number of axioms required to describe those things that are unaffected by an operator can be considerable. These axioms are called *frame axioms* and the problem of having to state all of them is often referred to as the *frame problem* [29].

A solution to this problem was found in the operator representation used in the STRIPS system [22], a planner that has shaped much of the planning research over the last 25 years or so. It was realised that an operator $o$ need only specify those things that it changes, that is its *effects*. It is then assumed that everything else in the input state of $o$ remains unchanged. This is known as *default persistence*. When default persistence is assumed the effects of an operator $o$ can be represented as a set of predicates that are removed from $s$ (the *delete* set $o.e^-$) and a set of predicates that are added to $s$ (the *add* set $o.e^+$).

We have included this solution into our model of planning and operators are specified in terms of *pre-* and *post-conditions*. The form of operators is 4-tuples as follows:

$(n, pre, e^+, e^-)$

The first argument is the name of the operator and this is often used to refer to the operator. The second argument of the operator are its pre-conditions which contain a set of predicates that must be true of the state that the operator is "applied" in (operator application is discussed in section 2.2.2.4). In later sections we will to refer to these states, in which all an operators pre-conditions are true, as *pre-condition states*.

We will also see (section 2.3.3) how these predicates can be introduced as goals to be achieved during planning. For some operator $o$ the pre-conditions are written as $o.pre$. The third and fourth arguments of the operator are its positive and negative post-conditions respectively (alternatively referred to as the add and delete sets). These describe the properties of states that result from application of the operator and are written as $o.e^+$ and $o.e^-$.

**Example: representation of FM operators**

For efficiency reasons the pre-conditions and positive post-conditions of FM operators are divided into a number of subsets as follows:

- pre-conditions in FM operators are split into the following subsets:
  - $sp$: a set of *state* pre-conditions that must be true from the current state of the modelled planning domain
  - $ep$: a set of *environment* pre-conditions that must be true from static predicates in the set of environmental information $ES$
  - $p$: a set of predicates that can be achieved by the effects of operators and can be introduced as "goals" to be solved during planning
- positive post-conditions of FM operators are split into the following:
  - $e^{+1}$: a set of *primary* effects that the operator can be used to achieve directly
  - $e^{+2}$: a set of *secondary* effects that the operator cannot be used to directly achieve

Hence, FM operators are represented as 7-tuples of the form:

$(n, sp, ep, p, e^{+1}, e^{+2}, e^-)$

As an example, consider the following FM operator from the warehouse planning domain $WW$:

$$
\begin{aligned}
(\{&load(C, X, Y),\\
&\{at(C, X)\},\\
&\{type\_of(X, space), type\_of(Y, space), type\_of(C, crate), next(X, Y)\},\\
&\{in\_truck, on\_floor(C, X), truck\_unloaded, truck\_at(Y)\},\\
&\{at(C, Y), truck\_loaded(C, Y), clear(X)\},\\
&\{\},\\
&\{on\_floor(C, X), at(C, X), truck\_unloaded\})
\end{aligned}
$$

### 2.2.2.2 Representing the Initial State and Goal Condition

Our model of planning also accepts two "special" operators, which we shall refer to as *current* and *goal*. The operator *goal* corresponds to the *finish* operator used in [19, pp 963] and the *goal* operator described by McCluskey in [45]. It is used to represent an input conjunctive planning goal and its pre-conditions are the set of predicates in the input planning goal.

In some respects the special operator *current* corresponds to the *start* operator of [19, pp 963] and the *init* operator of [45]. This is because *start* and *init* are used to represent the set of predicates that are achieved by the initial state of an input planning problem. However *current* is more general than this. It can be used to represent the initial state of a planning domain but it can also be used to represent the current state of a modelled planning domain and the sequence of operators that have been applied to the initial state to achieve the current state. Hence, this operator allows us to capture the action of plan generation algorithms such as *MEA* (one of the FM plan generation engines) which "advance" the state of the planning domain during plan generation.

The form of the special operator *current* is a 3-tuple as follows:

$$(current, oseq, e^{+})$$

where *current* is the name of the operator, *oseq* is a sequence of operators from the set of domain operators $OS$ (possibly empty) and $e^{+}$ is the set of positive post-conditions of *current*, in other words the predicates that are true from the current state of the planning world. The values of *oseq* and $e^{+}$ change during planning so that the set $e^{+}$ always contains the set of predicates that describe the current state of the modelled planning domain. For systems such as TWEAK [9] and NLP (as described in [45]), $current.e^{+}$ is always the set of predicates that form the initial state of the planning problem. Similarly, *current.oseq* is always the empty set $\{\}$ since the state of the modelled planning domain is never "advanced" during plan generation. It is only for plan generation engines, such as *MEA* (one of the plan generators within the FM system) where $current.e^{+}$ and *current.oseq* can change because operators can be applied to change the state of the modelled planning domain during planning. We will discuss this in more detail in section 6.1.

The form of the special operator *goal* is :

$$(goal, pre)$$

where *goal* is the name of the operator and *pre* is the set of pre-conditions of the operator. These are the set of predicates that form the goal condition of the input planning problem.

### 2.2.2.3 Binding Variables within Operators

Domain operators as discussed and illustrated in section 2.2.2.1 contain sets of predicates whose arguments can be variables. Consequently each operator describes a class of possible operators since the variables can be replaced by constants to derive ground instances of the operators. In this section we consider the way that predicate variables in operators can be instantiated.

Firstly, we define the set $SUBS$ which is the set of "legal" substitutions that bind variables within predicates to objects of the correct sort. It is assumed that relevant information about the sort (or type) of variables is supplied by the domain modeller as part of the environmental information for the domain. Using this set $SUBS$ we can refer to "ground" operators as operators where all variables in the operator are bound to objects of the correct sort, under some set of substitutions in $\mathbb{P}\, SUBS$. These ground operators correspond to the *primitive actions* described by Tate and colleagues in [73, pp 29].

The set of operators for a modelled planning domain are referred to as $OS$ and so we will refer to the set of all ground operators for a modelled planning domain as $OS_g$.

Finally, we note that because operators are modelled as logical terms the scope of variables within predicates extends right through the operator. Throughout the thesis we will use the convention that capital letters in predicate arguments represent variables and that different letters represent different variables. Hence, legal substitutions of operators will bind distinct variables within an operators to different objects. In situations where the binding of a variable in a predicate argument is unimportant, the anonymous variable "_" will be used.

### Example: binding variables in FM operators

To illustrate the way that variables are instantiated, consider the following warehouse world $WW$ operator:

$$
\begin{array}{l}
(\{load(C, X, Y), \\
\quad \{at(C, X)\}, \\
\quad \{type\_of(X, space), type\_of(Y, space), type\_of(C, crate), next(X, Y)\}, \\
\quad \{in\_truck, on\_floor(C, X), truck\_unloaded, truck\_at(Y)\}, \\
\quad \{at(C, Y), truck\_loaded(C, Y), clear(X)\}, \\
\quad \{\}, \\
\quad \{on\_floor(C, X), at(C, X), truck\_unloaded\})
\end{array}
$$

where the sorts of the different variables are listed in the set $ep$. For example, the variable $C$ is of sort (or type) *crate* and this can be replaced by any object of sort *crate*. Similarly, $X$ and $Y$ are of sort *space* and they can be bound with any *different* objects of sort *space*.

In FM, it is expected that sort information is supplied as part of the set of environmental information, $ES$. For example, $ES$ for the warehouse world includes (this is listed in full on page 202):

$$
\begin{array}{l}
\{type\_of(crate1, crate), type\_of(crate2, crate), type\_of(crate3, crate), \\
type\_of(s1, space), type\_of(s2, space), type\_of(s4, space), type\_of(s9, space), \ldots\}
\end{array}
$$

then the set $SUBS$ for this modelled planning domain will include the following substitutions:

$$\{\{crate1/C, s1/X, s2/Y\}, \ldots, \{crate3/C, s9/X, s4/Y\}, \ldots\}$$

#### 2.2.2.4 Properties of Operators

In this section we define a number of properties of the operator representation, that will be useful in later chapters. We start by defining the way that operators can be applied to a state of the modelled planning domain. Then we define how operators *achieve* and *clobber* goals. Finally, at the end of the section we extend the definition to cover sequences of operators that achieve and clobber goals.

Firstly, lets consider the result of *applying* a ground operator to some state, lets call it $s$, of a modelled planning domain. This results in a different state which is given by:

$$apply(o, s) = \{\{s \setminus o.e^-\} \cup o.e^+\}$$

so the the state that results is formed by simply deleting the predicates in the delete set and adding the contents of the add set. This is similar to the *application* procedure defined in [36, pp 12] which was based on Lifschitz' model [42].

Now lets consider what it means for operators to *achieve* and to *clobber* goals. We say that an operator, let it be $o$, achieves a ground goal, let it be $g$, if $g$ is one of the positive post-conditions of $o$ or in other words if:

$$g \in o.e^+$$

If a predicate, $p$, has been achieved then we say that it is *true*. The achiever of a goal corresponds to what Chapman [9, pp 338] and Christensen and Grove [11] referred to as a proposition that *asserts* a goal. We define the set of achievers of some ground goal $g$, as follows:

$$achievers(g) = \{o \in OS \mid \exists\, s \in \mathbb{P}\, SUBS : g \in o_s.e^+\}$$

which states that an operator is an achiever of $g$ if, under some legal substitution of operator variables, $g$ is amongst the operators positive effects. We can also define the set of ground or instantiated achievers of a ground goal $g$ as follows:

$$i\_achievers(g) = \{o \in OS_g \mid g \in o.e^+\}$$

Using the definition of *achievers* we can now define the set of *achievable predicates* for a domain model, which are contained in the set *preds*. This is given as:

$$preds = \{p \mid \exists\, o \in OS : p \in o.e^+\}$$

The set of ground predicates for a modelled planning domain is given by the following:

$$preds_g = \{g \mid \exists\, s \in SUBS, \exists\, p \in preds : g = p_s\}$$

which states that elements in the set $preds_g$ are elements from the set *preds* under some legal substitution from $SUBS$.

Now lets consider what it means for an operator to clobber a goal. A ground operator, let it be $o$, is said to clobber a ground goal, let it be $g$, if the operators negative post-conditions contain $g$ or in other words if:

$$g \in o.e^-$$

If a predicate, $p$, has been clobbered then we say that it is *false*. This corresponds to the situation where a proposition *denies* a goal as described in [9, pp 338] and [11]. Now the set of clobberers of a goal can be defined as follows:

$$clobberers(g) = \{o \in OS \mid g \in o.e^-\}$$

It will be useful for us to define what it means for a sequence of operators to achieve and clobber a goal. We start by defining the set of all sequences of operators from $OS$ for a modelled planning domain. This is given by:

$$seqs(OS) = \{o \in OS^* \mid \exists\, s \in \mathbb{P}\, SUBS : valid(o_s)\}$$

which states that elements in the set $seqs(OS)$ are sequences of operators from $OS$ (using the notation $X^*$ to denote the set of sequences of elements in the set $X$). A sequence is said to be *valid* if the pre-conditions of each operator in the sequence are achieved (and not clobbered) by operators earlier in the sequence and if the first operator in the sequence is "anchored" in any valid initial state.

Now we can define the positive and negative post-conditions of some $s \in seqs(OS)$. The positive post-conditions of $s \in seqs(OS)$ (the predicates that are achieved by it) are the set of predicates that are achieved by operators in the sequence and then not clobbered by later operators. So a sequence of operators from $seqs(OS)$ that achieve some goals can be said to be "justified" with respect to those goals in a similar sense to that used by Knoblock [36, pp 59]. Within his model, a plan (sequence of operators) is justified with respect to some set of goals if every operator in the plan either achieves one of the goals or is an achiever for pre-conditions of justified operators in the plan.

The negative post-conditions of $s \in seqs(OS)$ (the predicates that are clobbered by it) are the set of predicates that are clobbered by some operator in the sequence and then not re-achieved by some later operator in the sequence.

### 2.2.3 Problem Representation

Within the model of planning, input planning problems are composed of the following parts:

- An initial state $I$, where $I \in SS$ (as defined in section 2.2.1) which describes the initial state of the modelled planning domain.
- A goal condition $G$ which is a subset of a valid state of the planning domain, let it be $s$, which can be written as:

  $$\exists\, s \in SS : G \subseteq s$$

  Hence, a goal condition $G$ describes a set of states, all of which satisfy the goal condition.

### 2.2.4 Partial Plan Representation

Later, in section 2.3, we will model the process of plan generation as a search through a space of partially developed plans or partial plans for short. In this section we look at the structure of these partial plans and the relation between partial plans and fully developed plans that are solutions to input planning problems.

The representation for partial plans is based on that given in [45] where partial plans are represented as structures of the form:

$$pp(Os, Ts, Ps, As)$$

where elements in $pp$ are as follows:

- $pp.Os$ is a set of instances of operators from $OS$, the set of operators for the modelled planning domain (defined in section 2.2.2.1), along with the special operators named *goal* and *current* (also defined in section 2.2.2.1).
- $pp.Ts$ is a temporal order over members of $pp.Os$ which is represented as a set of pairs of the form $(o_1, o_2)$ where $o_1$ is "earlier" in the temporal order than $o_2$. The temporal order can be either *partial* or *total*. If the order is total then every operator in the partial plan is ordered with respect to every other whereas in a partial order operators can be unordered with respect to each other. The special operators *current* and *goal* are always the lower and upper bounds respectively, of the temporal order.
- $pp.Ps$ is a set of pairs of the form $(p, o)$ where $p$ is an unachieved pre-condition of some operator in $pp.OS$. For the special operator *goal*, its pre-conditions are exactly those predicates that appear in the input planning problem goal. When the planning system is initialised, the first partial plan $pp_i$ contains just these predicates from the problem goal.
- $pp.As$ is a set of pairs $(p, o)$ where each $p$ is an achieved pre-condition of some operator in $pp.Os$. Initially, the positive post-conditions of the special operator *current* contains all the predicates that are true from the initial state of the input planning problem. In a valid partial plan the intersection of the sets $pp.As$ and $pp.Ps$ must be disjoint.

#### Example: partial plan representation in FM

To illustrate the representation of partially developed plans we shall consider an example for the warehouse planning world $WW$. For the input planning problem with goal condition $G$ as follows:

$$G = \{crane\_at(s6), on\_floor(crate2, s13), clear(s2)\}$$

and initial state $I$ as follows:

$$I = \{truck\_at(s2), driver\_at(s2), on\_floor(crate3, s16), on\_floor(crate2, s12), on\_floor(crate1, s1), crane\_at(s14), clear(s9), clear(s3), clear(s8), clear(s4), clear(s6), clear(s7), clear(s10), clear(s14), clear(s13), in\_truck\}$$

then the initial partial plan that is formed, we shall call it $pp_i$, is as follows:

$$\begin{aligned}
&pp_i(\{current, goal\},\\
&\qquad\qquad \{(current, goal)\},\\
&\qquad\qquad \{(crane_at(s6), goal), (on_floor(crate2, s13), goal), (clear(s2), goal)\},\\
&\qquad\qquad \{(truck_at(s4), current), (driver_at(s2), current), \ldots,\\
&\qquad\qquad\quad (in_truck, current)\})
\end{aligned}$$

where for example the temporal order $pp_i.Ts$ contains the pair $(current, goal)$ which means that *current* is before *goal* in the temporal order (this is always the case since these special operators form the lower and upper bounds of the temporal order).

#### 2.2.4.1 Problem Solution

Now we can define what it means for a partial plan to solve an input planning problem. This is the case for a partial plan, let it be $pp$, if the result of "applying" any linearisation (total order) of the operators in the temporal order of $pp$ (along with any in the sequence of operators in *current*) to the initial state of the modelled planning domain results in a state which satisfies the goal condition. In other words a partial plan solves a planning problem with initial state $I$ and goal condition $G$ if the following is true:

$$\forall\, p \in linear(pp): \ G \subseteq apply\_plan(p, I)$$

where $linear(pp)$ is a function that returns the set of linearisations of some partial plan $pp$ and $apply\_plan(p, I)$ is a function that returns the state that is the result of sequentially applying (using *apply* as defined on page 15) each operator in $pp$ to the initial state $I$.

### 2.2.5 Planning Space

The process of plan generation, can be viewed as a search through a space of partially developed plans. Nodes in the space are partial plans and directed arcs in the space represent the way that partial plans can be specialised.

For a particular planning problem, this space is a tree structure which is rooted in a node that represents an initial partial plan $pp_i$. So for example, for a planning problem where the initial state consists of the set of predicates, $\{i\}$, and the goal condition is the set of predicates, $\{g\}$, the root node of the partial plan space is as follows:

$$pp_i(\{current, goal\}, \{(current, goal)\}, \{(g, goal)\}, \{(i, current)\}\}$$

Directed arcs in the space, represent the way that partial plans can be *specialised.* For example, if there is an arc from partial plan $pp_a$ to $pp_b$, this represents the fact that partial plan $pp_b$ is a specialisation of $pp_a$.

A partial plan $pp$ is specialised by the unique addition to the partial plan of a composite of three elements. The first of these is the addition of an operator from $OS$ to $pp.Os$. The second is the instantiation of variables in some operator in $pp.Os$ (or the addition of constraints on variable instantiation). The remaining way that partial plans can be changed is by the addition of a temporal order to $pp.Ts$.

The main difference between planning systems in the class DIP, as observed by for example Christensen and Grove [11] is the way that partial plans are *specialised*. In the next section (section 2.3) we shall review a number of different planning systems and the strategies that they use for plan specialisation. The reason why these different strategies have emerged is largely because there is a potentially (very) large number of partial plan specialisations. This means that only some of the partial plan specialisations can be considered if plans are to be generated in some reasonable period of time. In fact, we shall see in section 2.4 that the number of alternative specialisations to consider grows exponentially with the depth of the partial plan through the search tree.

Search through this space of partial plans is not guaranteed to lead to a partial plan that is a solution for a particular planning problem. This is unavoidable because there are certain planning problems for which no solution exists. A planning system is said to be *complete* if for any problem for which a solution exists, the system can generate a solution plan. A planning system is *correct* if every partial plan that it outputs is a solution to the planning problem that was input.

### Summary of the Planning Model

We are interested in a class of planning systems, that are generative and domain independent. They are generative in the sense that they generate entire plans for the problem before executing any part of the plan in a planning domain and they are domain independent because there is a logical separation between the encoding of the domain and the plan generation engine.

This class of planning systems, which we call DIP, generate plans by a process which can be viewed as a search through a space of partial plans. A partial plan is a solution to an input planning problem if all linearisations of the operators in it transform the initial state of the modelled planning domain into a state in which the goal conditions are true.

## 2.3 Planning Strategies

The main differences between planning systems in the class DIP lies in firstly, the way that they select partial plans for specialisation and secondly, the way that they specialise partial plans once they have been selected [11, pp 248]. The purpose of this section is to review a number of planning systems and to illustrate the different planning strategies that they use.

The discussion is divided into sections which discuss different *choices* that occur during planning. These are the "choice points" discussed by Minton and colleagues in, for example, [54, pp 81 – 82]. We summarise them as:

- which partial plan to specialise next
- which unachieved pre-condition in the selected partial plan to achieve next
- which operator to use to achieve the selected goal

- which substitution of variables within predicates in the selected operator

In the next four sections we will consider each of these four planning choices in turn and some of the weak general strategies that have been developed to choose amongst these alternative.

### 2.3.1 Partial Plan Choice

This is the problem of which partial plan to select to specialise next during plan generation. An exhaustive search strategy is one which considers every possible alternative, that is, every partial plan in the partial plan space. For example, a breadth first search of a partial plan space would consider every partial plan at one "level" before considering every descendent of partial plans at that level.

Such a strategy is *complete* in that it is guaranteed to find a solution to a problem if one exists. It is also guaranteed to find a plan of optimum length. However, for all but the most trivial domains this strategy is very inefficient since the number of alternatives to consider grows exponentially with respect to the *depth* of the search (see for example [41]).

Consequently, some choices must be made over which partial plans to specialise and which branches of the search tree to explore. One strategy is to use *weak heuristics* to guide the search, in other words to select partial plans. These are weak in the sense that they are domain independent and they are heuristic in that they can improve performance by reducing search but this cannot be guaranteed. So in some cases, the heuristic may cause the planning system to select partial plans and explore parts of the search tree that do not lead to a solution. One commonly used weak heuristic is to develop a plan *linearly*, as is the case with STRIPS. Under this strategy partial plans are selected in a strict linear order.

An important issue is how the choice of which partial plans to specialise influences the completeness of the planner. As we saw the completeness of a planner that exhaustively searches in a breadth first manner can be guaranteed, but this is not necessarily the case for a planner that, for efficiency reasons, does not consider all partial plans.

As well as strategies that select promising partial plans, heuristics have been used to reject partial plans that look unlikely to lead to a solution. For example, Dawson and Siklossy [14] and Drummond and Currie [19] present similar approaches to rejecting partial plans whose outstanding pre-conditions do not describe a valid state of the world. We shall see more of this approach when we consider approaches to compilation for planning in chapter 3.

### 2.3.2 Goal Choice

This is the problem of selecting which goal or outstanding pre-condition to achieve next in a partial plan that has been selected for specialisation. Weak strategies for selecting goals fall into two broad categories. The first of these focuses on whether sets of pre-conditions are viewed as linear or non-linear and we discuss this below in section 2.3.2.1. The second type of strategy, orders pre-conditions according to their relative importance, by the *abstraction* of detail (see section 2.3.2.2).

#### 2.3.2.1 Linearity Assumption

The STRIPS system [22] made an assumption that any planning problem with a set of goals $G$ (a conjunctive goal problem) could be solved by first forming a plan for each $g \in G$ and then combining the individual plans in some way to form a plan that solved $G$. In other words, it assumed that there is some *linear* order in which the goals in $G$ can be tackled.

But, as shown in the Sussman Anomaly [70] this assumption of linearity is not true for all conjunctive goal problems and for the generation of optimal plans. There are some conjunctive goal problems where some $g \in G$ **must** be achieved and then clobbered in the process of constructing a plan that solves the conjunction of goals in $G$. In this case the problem is said to be *non-linear*, since there isn't a linear order in which the goals in $G$ can be tackled. Although it should be noted that the anomaly is an artifact of the goal representation and expansion of the goal set may well mean that there is a linear order in which the goals can be tackled [41]. An example of the Sussman Anomaly and the effect of goal expansion can be found in section 3.3.5 on page 38.

A general solution to the Sussman Anomaly is attributed to Sacerdoti [63], first presented in the NOAH planner. He realised that it is the premature commitment to a linear order that causes difficulties for STRIPS type planners on problems such as the Sussman Anomaly. His solution was to avoid premature commitment to orderings by representing plans as partial orders of operators with respect to time. Since this strategy delays commitment to an ordering on plan operators, until interactions are known it is known as a *least commitment* strategy.

In his 1987 analysis [9], Chapman argued that all non-linear planners up to this time had been much the same. They tended to use the STRIPS assumption of default persistence for operator representation, the assignment of truth values to predicates that described a planning domain was completed under the the closed world assumption and they could tackle conjunctive goal problems. Chapman presented TWEAK, which was a rigorous re-construction of earlier systems such as NOAH and NONLIN [72]. He formalised planner action in the *Modal Truth Criterion* (MTC), which is a characterisation of conditions under which a pre-condition will be true at any point in a partially ordered plan. This was transformed into a goal-achievement procedure, that could choose completions of a partial plan where a pre-condition is true and then specialise that partial plan.

The assumption of linearity, in goal directed search, can directly effect the completeness of a planning system. This occurs for those planning problems for which a solution plan exists but which cannot be generated by planners that assume linearity. However, sometimes this assumption is made because it can greatly increase efficiency by reducing the number of partial plans that need be searched during plan generation.

#### 2.3.2.2 Abstraction

Another general strategy that has been used to influence goal choice is to use an abstraction hierarchy where the more "important" a goal the higher it is in the hierarchy. The two main types of abstraction that we consider here are *aggregation* and *approximation* from the taxonomy of abstraction types given in [18].

Under aggregation complex structures at one level of abstraction are subsumed under simpler structures, perhaps even single terms at higher levels. Aggregations involve changes in grain size as described in [30].

The NOAH system [63], was novel for a number of reasons, amongst them was that it planned using an abstract specification of a domain. The user of the system was expected to supply a hierarchical specification of the domain in the form of abstract operators. During planning, problems were first tackled with the most abstract operators and as planning proceeded successively detailed operators were introduced until a solution to the problem was found at the most detailed level. Abstraction of operators in this tradition has featured in many planning systems such as NONLIN [72], MOLGEN [68], SIPE [76] and FORBIN [16].

One advantage of this type of abstraction is that it permits the interleaving of different levels of abstraction which enables opportunistic problem solving, that is, in certain cases it may be beneficial to plan certain parts of a plan to lower abstraction levels. Care must be taken if one is to avoid *Hierarchical Promiscuity*, which is associated with the problem of determining the truth of a proposition when different planning levels are interleaved and their interactions are not well defined [76].

However one limitation of this type of abstraction is that the abstraction spaces tend to be hand tailored to a specific modelled planning domain. Also it is the task of the domain modeller to construct the abstraction hierarchy for a planning domain since the problem of how to automatically construct them has been largely unaddressed.

Approximations are the general class of abstractions which are formed by removing, or abstracting away some aspects of the problem in question. ABSTRIPS [62] is an example of a planner that created abstract planning spaces, by dropping sets of predicates that were deemed to be less important. The levels in the hierarchy came from criticality levels that were assigned to the predicates and the process of assigning criticality levels in ABSTRIPS was semi-automated. Interest in this form of abstraction has recently been revived with Knoblock's development of ALPINE [36], an efficient algorithm to generate abstraction hierarchies. We will return to this algorithm when we discuss it in more detail in section 3.3.1.

These automatically generated abstraction hierarchies can greatly increase the efficiency of a planning system, as demonstrated in our own experiments with ABGEN, an implementation of the ALPINE algorithm (see for example [50]). Another advantage of this type of abstraction is that the abstraction hierarchies can be automatically generated, hence reducing the burden on the domain modeller.

### 2.3.3 Operator Choice

The third of the four planning choices that we summarised on page 19 is the choice of which operator to use to achieve some outstanding pre-condition goal from a selected partial plan.

A well known example of a weak strategy for selecting operators to use to achieve preconditions in a partial plan is Means End Analysis. This was first introduced in the General Problem Solver (GPS) of Newell and Simon [58]. The strategy is to search for operators that reduce the difference between the goal condition and the initial state of the modelled planning domain in the current planning problem. This strategy can guide the search, starting with the goal condition $G$ and choosing operators that achieve $g \in G$ and as far as possible leave other goals unchanged. Then any outstanding pre-conditions of the operator are added to the set of goals to be solved.

This strategy was used by GPS and STRIPS and has been used by many AI planning systems since. For example it is the principal strategy available to the FM planning system [46], which we are using as an example throughout the thesis.

### 2.3.4 Binding Choice

The fourth choice that can occur during planning is how to bind any predicate variables in operators that have been selected to achieve outstanding goals. In a sense the separation between the choice of binding and the choice of operator, discussed in the previous section, is rather arbitrary, since we could merge the two and discuss the choice of "ground" operators (as defined on page 13).

If predicates in operators can contain variables then they describe a set of ground operator instances that result from replacing predicate variables with constants. Hence during plan specialisation when an operator is selected to achieve outstanding pre-conditions in some partial plan there can be a choice of which ground operators to use to achieve the outstanding goal. In the previous section we discussed MEA as a strategy for selecting from a set of operators that could be used to achieve an outstanding goal. But of course MEA can also be seen as a strategy for selecting from amongst ground operators.

Also, planners in the class DIP, can leave variables within operators unbound, if the binding is not important in the context of some partial plan. For example the domain dependent planner MOLGEN [68] was the first planner to generate a plan by incrementally specifying constraints that the plan must satisfy. This could include constraints on the way that variables within predicates in plan operators could be instantiated. So a plan could be developed within which variables were left uninstantiated.

## 2.4 Planning Complexity

In this section we turn our attention to recent research findings on the computational complexity of the planning problem. These results are important for this thesis because we are interested in ways to improve the performance of planners in the class DIP and the extent to which the performance can be improved depends directly on the general complexity of the planning problem. The planning problem can be formulated in a number of ways, such as:

- *a decision problem*: does there exist a plan which maps the initial state into a state satisfying the goal condition
- *a search problem*: find the shortest plan which maps the initial state into a state satisfying the goal condition

One reason for analysing the complexity of a particular problem is to determine whether we can design an algorithm which will correctly solve all instances of the problem in polynomial time. For example, if theoretical analysis shows a problem to be NP-complete then we know (assuming $P \neq NP$) that we cannot design an algorithm which will correctly solve *all* instances of the problem in polynomial time. However, further complexity analysis may enable us to identify *some* instances of the problem which can be solved in polynomial time. The significance of a proof of intractability is that any algorithm we design will never be able to solve *all* instances of the problem in polynomial time.

Dean and Boddy [15] looked at the complexity of a problem called *temporal projection*. This is the problem of determining whether a set of operators can be applied to some initial state of a modelled planning domain in an order consistent with some partial order. This is the case if the pre-conditions of the first operator thats applied are true in the initial state and if all pre-conditions of all operators applied later are true of the state they are applied to. This corresponds roughly to determining if a sequence of operators are justified (this was discussed earlier on page 16).

This is somewhat different to the problem of plan generation where the task also includes generating a partial order of operators to achieve some goal condition. Nevertheless, their complexity results are relevant since temporal projection may be performed during planning in determining what conditions must hold before and after each operator in a partially developed plan. In their analysis they considered a number of instances of the temporal projection problem in which certain aspects of it were restricted. Problem restrictions include such things as limiting the number of post-conditions which are permitted in action representations, allowing only positive post-conditions (no clobbered predicates). Their results show that for all but the most restricted instances of the problem, temporal projection is at least *NP-hard*.

Chapman [9] identified some general properties of operators that greatly affect the complexity of non-linear planning. The action representation of his planning algorithm TWEAK was restricted in that it did not permit conditional operators, dependency of effects on the input situation and derived side effects. With this restricted operator representation he was able to prove the completeness and correctness of TWEAK. He also showed that the time complexity of TWEAKs' goal achievement procedure is polynomial with respect to the number of operators in a plan. In addition he proved that a more expressive operator representation, such as one that permits conditional operators, makes goal achievement *NP-hard*.

A number of researchers have turned their attention to domain dependent planning, as in the *Blocks World*, a domain that has been used in the testing of many planning algorithms. Both Gupta and Nau [28] and Chenoweth [10] have independently shown that the problem of determining whether there exists a solution plan shorter than some specified length in the Blocks World is *NP-complete*. They have also shown that the problem of finding the shortest length (optimal) solution plan is *NP-hard*.

Bylander [6] also presents complexity results for the blocks world. He created a series of different planning problems by placing different constraints on the type and size of the pre- and post- conditions of operators and analysed the complexity of the various instances of the planning problem. The finding for the most general case, where operators can have any number of pre-conditions and post-conditions, is that the planning problem is *PSPACE-complete*. Indeed, this is true even if each operator is limited to just one post-condition. He found that the blocks world becomes tractable (in other words solvable in time that is polynomial with respect to the length of the plan), when one restricts the operator representation to positive pre-conditions and one post-condition. The apparent difference between the results of Bylander and those of [10, 28] above lies in the fact that Bylander was not concerned with finding optimal solutions to the blocks world planning problem. We can conclude from this that for some planning tasks and problem domains, such as the blocks world, it is possible, by giving up optimality, to render the problem tractable.

In 1992 Bylander [7] concluded that the complexity of planning arises from the properties of the *problem* and not any algorithm for it. Complexity results show how hard it is to find a set of operators that accomplish goals but are neutral on how the sequence of operators are generated. The key issue, according to Bylander, is how much can domain independent planning live up to its name.

## 2.5 Building Efficient Planning Systems

Recent research findings on the abstract complexity of planning, reviewed in the last section, strongly support the view that we will never be able to develop an algorithm that can solve *all* instances of the planning problem in reasonable time. Indeed, the complexity of the problem is such that even building planners that perform efficiently for *some* problems is very difficult.

Nevertheless, we are still interested in building domain independent planning systems that can perform efficiently on a selected range of problems and we are not alone in this endeavour. For example, there is considerable work ongoing with the PRODIGY system (see for example [8]) learning control rules that will guide search during planning. Another well known research project is the O-Plan system (see [13] for an overview) which is a domain independent planning system that uses a variety of search control heuristics to control the size of the search space during planning.

Both these planning systems are domain independent, and there are a number of reasons, mostly from a software engineering perspective, that explain why this approach is popular. These include:

- the logical separation between the plan generation engine and domain model mean that theoretical properties of the plan generation engine such as the inherent computational complexity and the completeness of the planner can be studied.
- maintenance of the domain model is made easier because of its logical separation from the plan generation engine since for example, the domain model can be independently validated
- the planner can be applied to any planning domain which can be captured adequately in the input language of the DIP
- a user of the planner only needs to be familiar with the semantics of the input language, and not the procedural details of the planner itself

Our approach to improving the efficiency of domain independent planning systems lies in *problem reformulation.* There is a strong relation between the way a problem is represented and algorithm efficiency. Also, problem reformulation can be a very powerful problem solving tool. For example, Korf [39] showed that even quite dramatic shifts in representation can be seen as a series of relatively minor *isomorphisms* or *homomorphisms*, which are structure preserving transformations. Although Korf looked at very domain specific representations, the effect of changes in representation do seem to extend to the domain independent representations of the planning systems that we are interested in. One illustration of this is the ALPINE system [37] which automatically generates abstraction hierarchies where levels in the hierarchy are homomorphisms of the original problem since detailed information has been dropped from the problem representation. For certain planning problems this re-representation can change the complexity of plan generation from exponential to linear in the length of the solution to the problem [36, pp 43].

Our approach to improving the efficiency of domain independent planning systems is to reformulate the problem so that planning problems become "easier" to solve, as measured against some performance metric. In the context of domain independent planning this means a reformulation of the model of some planning domain rather than changing the plan generation engine. We call this process *compilation for planning* and it is the subject of the next chapter.

# Chapter 3

# Compilation for Planning

As we saw in the previous chapter, the application domain of a generative domain independent planner (DIP) is encoded as part of the input to the planning system rather than being part of the plan generation engine. Ideally, this encoding, called the domain model, is written in a well defined language in which a domain expert can model actions and environmental information free from the efficiency issues of the planner. This language tends to be based around some variation of first order logic where the user encodes a high level declarative representation of *what* the domain information is without expressing *how* it will be used (for example [45] and [22]). To some degree the more declarative the input language the more likely a user with no knowledge of the DIP could model the domain [74, pp 89]. Interestingly, this is similar to the situation in state-based software specification where systems are initially specified implicitly, with no (apparent) thought given to how they will be implemented.

Unfortunately, an input language that is well suited to the needs of a domain modeller is not necessarily the best format for the planning system that it is input to. Indeed, the slowness of declarative representations is one of the reasons for the general inefficiency of DIP's. This "mismatch" between what is best for the modeller and what is best for the plan generation engine is exacerbated because different modeller's can encode the model differently. Also, slight differences in representation can cause considerable changes in the performance of the planning system (for example [20]). We argue that a *compiler* can transform the domain model, encoded in as natural a format as possible by the domain modeller, into a form that can be more efficiently reasoned about by the plan generation engine.

We start this chapter by defining compilation for planning (section 3.1) and a model of compilation for planning (section 3.2). In section 3.3 we review work in learning and compilation within our model of compilation. Finally, in section 3.4, we summarise the main features of compilation for planning.

## 3.1 Definition

The term *Knowledge Compilation* has been quite widely used in different fields of AI, since its introduction by Neves and Anderson in 1981 [56]. For example, it has been applied to compilation of rules for diagnosis and design in model based reasoning systems [35] and to hybrid knowledge bases [4]. Also, there is a thriving subfield of machine learning that uses techniques such as experience based learning to improve the performance of domain independent planning systems [54]. In section 3.3 we shall look at a number of these techniques within a framework of compilation for planning.

Despite the wide use of the term knowledge compilation, there is no one accepted definition of its meaning. For example, Keller [35] cites the following alternatives:

- shifting from a declarative to a procedural form of knowledge representation;
- automatically transforming explicit but inefficient knowledge representations into implicit but more efficient forms;
- production of knowledge based systems from higher level specifications.

A slightly different definition comes from the machine learning literature [24]:

- conversion of a declarative specification or domain theory into an efficient program

Within this context, knowledge compilation is linked to *Speedup Learning* (as defined in [24]), which is the problem of improving the performance of planning systems, because a declarative specification or theory of a planning domain can be seen as a slow planner.

How then can we define knowledge compilation for planning ? A common theme in all the definitions of knowledge compilation, is a transformation from a *declarative* or *explicit* form of knowledge, to a *procedural* or *implicit* representation of the knowledge. Also, the reason for compiling the knowledge is to transform it into a more *efficient* form. Since our focus is compilation for domain independent planning the "knowledge" that is compiled is knowledge about particular planning domains. This can include the domain model, in terms of the operators $OS$ and environmental information $ES$ and planning problems, as an initial state $I$ and goal condition $G$. It can also include knowledge about previous planning sessions, such as a solution plan from a successful planning session. Now we can define compilation for planning as:

- the transformation of knowledge about a planning domain into a format that can be more efficiently reasoned about by a particular plan generation engine

Although knowledge compilation and compilation for planning are similar in some respects to programming language compilation, there are differences. We can illustrate these differences by looking at the difference between *vertical* and *horizontal* compilation (in a similar sense to [4]).

Programming language compilation can be seen as vertical in that it is a *translation* from a "higher" level language (e.g. programming language source code) to a "lower" level language (machine code). As an example, we shall consider how programs written in Prolog are compiled (this is loosely based on an example given in [4]). Prolog is a programming language that consists of Horn clauses, which are a subset of first order predicate logic, where only rules with single conclusions are permitted. A Prolog interpreter can execute Prolog programs using a theorem proving technique known as resolution refutation. But the Prolog code can also be compiled to a set of instructions for an abstract machine that is designed to efficiently handle unification, a time consuming process that degrades the performance of the the Prolog interpreter. The compiler analyses the program and generates unification instructions in the language of the abstract machine. This translation of the original Prolog program results in much faster runtime performance.

As an example, a knowledge compiler might transform from a knowledge structure which is more natural for a domain modeller, such as general rules with multiple conclusions as follows:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

(where the conclusions $C_1, \ldots$ are true if the premises $P_1, \ldots$ are true) to a form that is more efficient for traversing the rules from conclusions to premises. If this was the case then the general rule above could be transformed to:

$$\begin{aligned}
&C_1 \leftarrow P_1 \wedge \ldots \wedge P_n \\
&C_2 \leftarrow P_1 \wedge \ldots \wedge P_n \\
&\vdots \\
&\mathit{Conclusions} \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n
\end{aligned}$$

The point that we wish to illustrate here is that compilation of knowledge is horizontal if the transformation is within the same language. There is no reason why the horizontal and vertical compilation cannot be combined, so that for example, first the multiple conclusion rules are transformed into horn clauses (horizontal) and then translated into instructions for the abstract machine (vertical).

## 3.2 Model of Compilation

In this section we shall develop a model of compilation for domain independent planning. The model is based on Bylanders' model of knowledge compilation that he developed to explore the potential of knowledge compilation [5]. It distinguishes between knowledge at the *knowledge* level and the *symbol* level [57] but we will restrict our discussion to the symbolic representation of knowledge for planning.

As we mentioned in the previous section, knowledge for planning can include a model of the planning domain, planning problems for that domain and the results of previous planning sessions.

We start by introducing a compilation function that we shall call $C$. It takes as input a domain theory, $d$, containing all the encoded knowledge for some planning domain, and a domain independent planning system $P$. Its output is a compiled version of the domain theory, which we refer to as $d'$. The compilation function $C$ is dependent on the expected class of the planner (in this case the class of DIP's) and hence the planning system $P$ is part of the input to $C$. The compilation function can be defined as follows:

$$C: \; P \times d \rightarrow d'$$

A domain model, compiled or otherwise, can be bound, using the function $B$, with $P$ to form a planning application $P_{d'}$ for run-time execution. We assume that the binding function can have no effect on the domain theory or the planning system. The binding function $B$ is defined as follows:

$B: P \times d' \rightarrow P_{d'}$

We see three ways that the model of a planning domain can be changed by compilation. The knowledge in the domain model can be stored more efficiently, new knowledge can be added or knowledge can be removed. Lets consider each of these in turn.

- $\mathit{Theory}(d) = \mathit{Theory}(d')$

  Compilation can change a theory of a modelled planning domain by changing the way that knowledge is stored. In this case the original domain theory $d$ is equivalent to the compiled domain theory $d'$, since no new knowledge has been added to the domain theory. In other words everything in $d'$ must have been implied by $d$ and vice versa. The advantage that $d'$ has over $d$ is in terms of efficiency, because the knowledge in $d'$ has been represented and stored in a more efficient form than in the original. This is the case with the horizontal compilation example that was given on page 29.

- $\mathit{Theory}(d) \subset \mathit{Theory}(d')$

  Compilation can change the domain theory by adding new knowledge [5]. In this case the original theory $d$ is a subset of $d'$, so some new knowledge has been added. We assume that this knowledge could not come from the planner $P$ (which is part of the input to the compilation function) or from the binding function since they are both assumed to be domain independent. So the new knowledge must have been introduced by the compilation function $C$. If $P_{d'}$ is more efficient than $P_d$ it is due to the knowledge that has been added by the compilation function.

- $\mathit{Theory}(d) \supset \mathit{Theory}(d')$

  Compilation can change the domain theory by removing knowledge [5]. In this case the compiled theory $d'$ contains less knowledge than the original, because aspects of the domain knowledge have been omitted or approximated. A by product of this compilation is that the compiled knowledge can be less *effective* than the original since it contains less information.

In our work we are only interested in the first of the cases listed above, where compilation transforms the representation of the domain theory, or in other words:

$\mathit{Theory}(d) = \mathit{Theory}(d')$

We focus on this form of transformation since we are interested in finding *general* ways of transforming a domain theory into a form that can be more efficiently reasoned about by a planner. For a compilation function to add knowledge from the domain model then it requires specific knowledge about that particular domain model and hence is unlikely to be general. Also, if a compilation method can remove knowledge about a domain then it may compromise the completeness of the planner by removing knowledge of the domain.

It should be noted that there is a limit to the amount of improvement that can be achieved by representing domain knowledge more efficiently. It may be possible, in some cases, to change the complexity of generating a plan from exponential to linear (in the length of the plan) by compiling the domain model. This is the case, for example, with a particular encoding of the Tower of Hanoi planning domain when it is compiled using the ALPINE algorithm [36]. Of course this degree of improvement will not be possible in all cases because of the general complexity of plan generation since we know that plan generation is intractable in general (see the discussion on page 23). This means that we cannot design an algorithm which will correctly solve all instances of the planning problem in reasonable time (assuming that $P \neq NP$). Consequently we will never be able to discover a tractable compilation function that always outputs compiled domain models that make plan generation tractable. Rather, we would expect to find methods that can derive tractable compiled planning applications in some instances.

## 3.3 Compilation Methods

In the previous section the compilation function $C$ was defined as a function that takes a domain model and a plan generation engine as input and outputs a more efficient representation of the domain model. It does this by applying a series of transformations which we call *compilation methods* to the domain model.

In this section we review a range of different compilation methods that could be part of the compilation function $C$. The methods are grouped together on the basis of the structure of the compiled domain "knowledge" that is explicitly represented in the domain model after compilation. The different knowledge forms include abstraction hierarchies, goal orders, control rules, macro operators, sub-goals and state invariants. We shall consider each of these in turn.

### 3.3.1 Abstraction

The use of abstraction during planning, has been widely explored as a means of reducing search. The basic idea is that during planning, the less important aspects of a problem can be abstracted away, so that the planner can concentrate on the most important parts of the problem first. Once an abstract plan has been generated for the most important parts of the problem then the less important parts of the problem can be re-introduced. This process is repeated until the problem is completely solved at the most detailed level.

In section 2.3.2.2 two different types of abstraction were discussed. These were *aggregation* where a user specifies a domain in terms of operators at different levels of abstraction and *approximation* where abstract planning spaces are created by dropping from the state description predicates that are unimportant. Aggregation has been widely used in the modelling of planning domains (see for example: [16], [68], and [76]) but there has been little work on automatically generating these abstraction hierarchies. On the other hand, there are a number of methods that automatically generate *approximate* abstraction hierarchies. These methods fit our model of compilation for planning because they transform the representation of the domain into a more efficient form. In the remainder of this section we shall look at a few examples and consider in what situations they can improve efficiency of a planning system.

ABSTRIPS [62] was the first system to attempt to automate the construction of abstraction hierarchies for planning. The main limitation of ABSTRIPS was that it could overlook situations where goals in a lower level of the abstraction hierarchy violate goals at higher levels. As a result it could offer incorrect plans as solutions to planning problems (see for example [34] and [38]). Also, ABSTRIPS was only partially automated, since it relied on user supplied criticality levels that gave some measure of the relative difficulty of achieving certain goals in a domain.

Some of the ideas of ABSTRIPS have been recently resurrected in the ALPINE algorithm [37]. It is a tractable algorithm for automatic generation of abstraction hierarchies that overcomes some of the limitations of ABSTRIPS. It constructs an abstraction hierarchy by placing goal predicates at different levels of the abstraction hierarchy, called *criticality* levels, depending on how important they are judged to be, in the context of some conjunctive planning goal. The importance of a goal is determined by looking at the possible ways that it could be achieved. So for example, a predicate $g_1$ is more important than some other $g_2$, if $g_2$ could possibly be clobbered in the process of achieving $g_1$. (For more detail on this algorithm see: page 43 of chapter 4). The use of these abstraction hierarchies can greatly increase the efficiency of a planning application, for example in the tower of hanoi domain, it is claimed that they can reduce the search from exponential to linear in the length of a problem solution [36].

However there are some planning domains where it is not possible for ALPINE to judge which goals are more important. In this situation it generates hierarchies with only one abstraction level (or very few), which is the same as the original planning model before compilation. This problem, *collapsing*, occurs because the algorithm considers *possible* interactions between goals during planning. So while it could determine that $g_1$ is more important that $g_2$ because $g_2$ could possibly be clobbered whilst achieving $g_1$ it could also determine that $g_2$ is possibly more important than $g_1$. Hierarchies collapse when the interrelation between predicates in the operators are such that all predicates are assigned the same criticality level.

This problem can be lessened by generating problem specific abstraction hierarchies because only a subset of all the predicates in the domain need be considered. This is what is done in ALPINE, it always compiles a hierarchy for a specific problem. The compilation is therefore done at run-time and the cost of generating the hierarchy is included in the overall on-line planning cost.

The collapsing problem may be lessened by using operators whose positive effects have been separated into *primary* and *secondary* effects [23]. For any two effects of an operator, called $e_p$ and $e_s$, $e_p$ is primary because to achieve it $e_s$ must be clobbered but $e_s$ is secondary because there is a way to achieve it without clobbering $e_p$. This can reduce the collapsing of abstraction hierarchies because it reduces the number of possible ways of achieving goals (the number of goals are reduced since the set of $achievers(g)$ now contains only those operators for which $g$ is primary). The results of experiments described in [23] show that the use of operators with abstracted effects can lead to finer grained abstractions than produced by ALPINE.

Can we predict in what situations the effect of using automatically generated abstraction hierarchies will be effective in improving the efficiency of plan generation ? The results of experiments by Bacchus and Yang [2] suggest that it is only possible to improve planning performance by using abstraction hierarchies that have, or are close to having, the *Downward Refinement Property* (DRP). Simply put, an abstraction hierarchy has this property if most abstract plans can be refined to a concrete level without the need to backtrack across abstraction levels. If this is not the case then, they argue, performance can be degraded. They showed that hierarchies generated by ALPINE do display the DRP.

As we shall see in subsequent chapters, part of our work has led to the development of a compilation method, PRECEDE, that compiles orders amongst predicates in a domain model. A useful consequence of this work, was the realisation that this method does not suffer from the the collapsing problem of methods such as ALPINE. This new compilation method PRECEDE is the subject of chapter 4.

### 3.3.2 Goal Orders

We saw earlier, in section 2.3.2.1, that interactions between goals in a conjunctive goal planning problem add to the difficulty of generating plans. For example, interactions between goals mean that it is not always possible to assume that there is some linear order in which goals in the conjunction can be tackled. An active research issue has been the problem of how best to deal with these interactions in a domain independent way during planning (see for example: [9], [22] and [63]).

Other research has focussed on developing automatic methods for identification of interactions between goals before planning which can then be used to order problem goals during planning. The work that we review in this section can be seen as compilation methods for planning since they are input a domain theory and output a transformed domain theory that contains an explicit representation of interactions between goals in the form of goal orders. In some senses methods that compile abstraction hierarchies are much the same as methods that compile goal orders. This is because automatically generated abstraction hierarchies, such as ALPINE [36], group goals from the planning domain in different "levels" of abstraction hierarchies according to their relative importance. The most important goals are at the highest levels in the hierarchy and the less important goals are at lower levels. A goal is less important than another if it can be inserted into a partially developed plan without clobbering any of the goals that appear at a higher level in the hierarchy. This means that the abstraction levels specify an order in which to tackle the goals, namely tackle goals in a descending order, starting with the goals in the highest level in the hierarchy.

The use of goal orders can significantly improve the performance of a planning system but this depends on the relationship between the goals in any particular planning problem. Korf [41] showed that the amount of improvement depends on whether problem goals are *independent*, *serializable* or *block-serializable*. If a conjunction of goals are *independent* then there is no interaction between them. In this case each of the goals can be solved separately and then concatenated into a global plan that solves the whole problem. This can significantly reduce the size of the search for the problem. For example, the size of the search space for an exhaustive search strategy is in the order of:

$$O(b^d)$$

where $d$ is the depth of the search (in other words the number of operators in the global plan) and $b$ is the average branching factor of the search. But if the goals in the planning problem were independent then the search could be reduced to the sum of the much smaller searches for each of the sub-goals. Indeed, this could be reduced even further since concurrent processes could be used on each of the independent sub-goals.

The reduction in search is not so straightforward when sub-goals are *serializable*. A pair of sub-goals, lets call them $g$ and $pg$, are serializable if one of them, let it be $pg$, can be achieved and *protected* whilst $g$ is achieved. If a serial order such as this can be identified prior to run-time then the planning system can tackle $pg$ first and only then start to work on the plan for $g$. Decomposition of a planning problem in this way, can reduce the branching factor of the search if there are less achievers for $g$ once $pg$ has been solved. The number of achievers of $g$ can be reduced because all the operators that clobber the protected goal $pg$ are not considered. But a drawback of this decomposition is that it can lead to longer (non-optimal) plans, since extra operators might be needed to avoid clobbering the protected goal. For example, in figure 3.1 for the robot planning world $SW$, the *robot* is next to $box1$ at $door25$. If we suppose that the conjunctive goal planning problem includes:

$$G = \{closed(door25), in\_room(box1, room5), \ldots\}$$

then $closed(door25)$ is already achieved (as shown in the figure) and the other goal $in\_room(box1, room5)$ can be achieved without clobbering it, by for example pushing $box1$ into $room3$ and then into $room5$. But this plan is longer than the optimal plan, which would open $door25$, hence clobbering the protected goal and then re-achieve it once $in\_room(box1, room5)$ had been achieved.

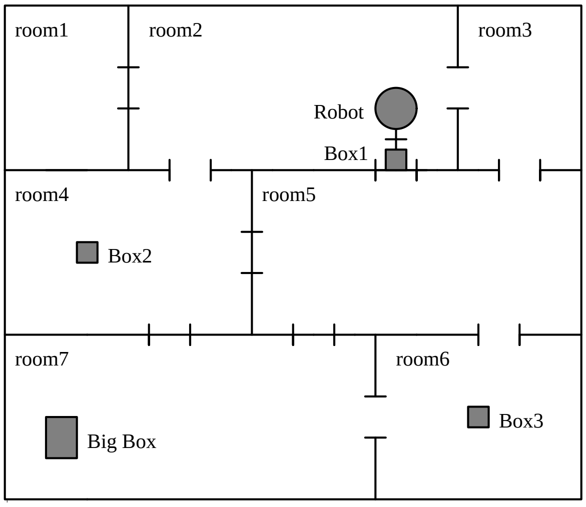


Figure 3.1: Example Problem $SW$

If a pair of goals are *non-serializable* then one of the goals must be achieved, clobbered and re-achieved in the course of making them both true. For some planning domains it can be useful to decompose problems because this can help identify partial solutions closer to the goal state. As Korf points out, serializability is often a function of the granularity of the set of sub-goals. This means that in some instances a set of non-serializable sub-goals can be made serializable by grouping them into "blocks", which can then be treated like single sub-goals. This occurs, for example, in the 8-puzzle planning domain, (consisting of a $3 \times 3$ grid with 8 numbered movable tiles and one space left blank, problems are to arrange the tiles in certain configurations), where individual sub-goals are not serializable but they can be grouped into blocks that are.

It is relatively straightforward to test whether a set of goals are independent (see for example [55]). However, this is not the case for sets of goals that are serializable. It is not possible, in general, to test if a set of sub-goals are serializable, since this is as complex as proving that a problem is solvable from all initial states [41, pp 72]. Consequently, it is only possible in certain circumstances to identify a serial order in which to tackle a set of serializable goals. For example, the abstraction hierarchies generated by ALPINE [36] provide a way to decompose goals in a planning problem but this method is not guaranteed to find an order to decompose a problem in all cases.

Another example of a problem specific approach to ordering goals is that of Cheng and Irani [32, 33]. Their approach to ordering planning problem goals is based on identifying necessary interactions between them. The algorithms they have developed enable the construction of a partial order in which problem goals can be attacked without having to backtrack across the goals. The method for generating the partial orders is to look for goals which can be achieved *later* than any others. For example, some goal $g$ in an input conjunctive planning goal $G$ can be achieved later than all of the "other" goals in the set $Og$ as follows:

$$Og = \{G \setminus \{g\}\}$$

if there are achievers of $g$ whose pre-conditions are consistent with the goals in $Og$ and which do not clobber any of them. We will consider this method in more detail in section 8.2.1.

### 3.3.3 Control Rules

Many systems have been developed which create control rules to help guide search during planning. One example of this approach is the PRODIGY system (see for example: [8]). which relies on control knowledge, generated from learning and static analysis of the domain model, to select from amongst alternative choices during planning [1]. Another example of this approach is the FM system [46] which learns *chunks* to help guide search during planning. We shall be considering this method in more detail in section 6.2.4.

Once again the distinction between this type of compiled structure and the others (e.g. abstractions and goal orders) is somewhat blurred since the other types of compiled structures that are generated also affect these four planning choices. As we have seen, abstractions and goal orders directly influence the choice of which goal to achieve next. They could also influence the choice of which partial plan to specialise next since for example a partial plan with more outstanding goals that are "lower" in an abstraction hierarchy could be judged to be closer to a solution state. We included a heuristic of this sort in our experimental testing of ABGEN, our implementation of ALPINE [48]. Macros tend to influence the choice of operator and binding but in cases where they are generated for a specific goal they can also affect goal choice for example if the macro contains a solution to the goal.

One method that has been used to generate control rules for planning is EBL, which stands for Explanation Based Learning. EBL is an area of active research in machine learning [17] and we shall discuss its use again in the section on macro generation

[1] Recall from section 2.3 that these choices are which partial plan to specialise next, which goal to try and achieve next, which operator to use to achieve the goal and which bindings of the selected operator.

(section 3.3.4). Here we consider how EBL can be used to generate control rules for planning, within the PRODIGY system [54]. Part of the EBL process is to generate an explanation of why a training example is an instance of some concept. The training example is a trace of a planning session and concepts to be learned include success and failure. The explanation is a proof that the training example is an instance of the target concept. Then by computing the weakest pre-conditions under which the explanation holds, a sufficient condition for recognising the concept can be obtained. For PRODIGY, the weakest pre-conditions of the explanation become the antecedents of the control rules. Control rules generated from successful planning traces are used to form preference rules and from planning failure they are used to form rejection rules.

Control rules have also been generated for the PRODIGY system by a process of static analysis of the problem space, as shown in the STATIC system of Etzioni [21]. STATIC constructs a set of graphs, referred to as PSG's, one for each achievable predicate in a domain model. The graphs represent relationships between goals and operators in the problem space, where nodes in the graphs are predicates from the domain and arcs in the graph represent the action of domain operators. Each graph is rooted in a distinct sub-goal literal. So for example, in the robot planning world $SW$ where the set of achievable predicates are as given below, a PSG would be constructed for each of these predicates.

$$\begin{aligned} preds = \{&robot\_in(R), robot\_at(D,R), robot\_near(B), in\_room(B,R), \\ &at\_door(B,D,R), next\_to(B,B'), open(D), closed(D)\} \end{aligned}$$

An important feature of the graphs, as observed by Etzioni, is that they correspond to non-recursive explanations in EBL as used by the PRODIGY system. These graphs (non-recursive explanations) are used by STATIC to analyse goal clobbering and goal interaction. For each PSG, STATIC analyses the status of predicates at nodes in the graph in the context of achieving the goal that is at the root of the PSG. Then nodes are labelled as achieved, clobbered or unknown and are used to form control rules about goal clobbering and goal interaction. The explanations in the graphs are non-recursive and the interactions that are checked are only binary so the coverage of STATIC is more limited than EBL but the cost of acquiring and utilising control rules that are formed are less. Etzioni does not see STATIC as an alternative to EBL, rather he sees it as a complementary method and sees a hybrid system where some processing is performed as a result of static domain analysis and some by EBL as being the most useful approach.

### 3.3.4 Macro Operators

A macro operator is simply some sequence of operators which in some sense have been merged to form one new operator. The process of macro operator construction can be seen as one of proceduralisation [56] with parallels in human learning whereby frequently occurring actions are proceduralised into a new action. As such it can be seen to fit within our model of compilation where declarative knowledge about a domain is re-represented into a more procedural or efficient form.

Macro construction has been a feature of many planning systems from the early STRIPS system onwards [22]. In STRIPS macro operators were formed by dropping constants and regressing goals through successful problem solving traces. These macros were little more than frozen linear sequences of operators. The main problem with linear macros, as generated by STRIPS, is that slight variations in a problem can mean that plan traces which differ by only a few steps may require different macros. Also linear macro's cannot be generalised to an arbitrary number. This problem is often referred to as the *generalisation-to-N* problem [66].

Recent work in macro generation has led to methods for the generation of iterative and disjunctive macros. Shell and Carbonell [67] propose a solution, implemented in FERMI, to both the composition of disjunctive and iterative operators and a method to determine when they should be formulated. Other recent examples of approaches that generate iterative macros include [12], [69] and [65]. Another direction of recent work has been a shift in emphasis to focus on when and how to build useful macro operators. For example, Korf [40], Minton [51] and Iba [31] searched for useful heuristics that helped decide when a macro operator might improve planner performance.

How can the use of macro operators improve the performance of a planning system ? If they restrict the number of operators to be considered at each planning node then the average branching factor of the search is reduced. However, the full set of operators after compilation, $OS'$, can include new macro operators, hence the set of operators is:

$$OS' = OS \bigcup \{macros\}$$

where $OS'$ is larger than the original set of operators $OS$. The size of the set of operators is increased by the addition of macro operators and this means that the average branching factor is increased because the number of alternatives to consider at each choice point during planning is increased. If the number of operators is increased by addition of macros then the cost of searching for achiever operators can be greater than in the original set of operators. Until recently there had been little success in overcoming this problem of the utility of using learned macro operators, this is often referred to as the utility problem [51, 52].

Recently, however there has been some encouraging results, such as with the disjunctive and iterative macros of Shell and Carbonell [67]. An interesting feature of the macros generated by their FERMI system is that because they are disjunctive they are more general than linear macros and hence the size of a set of disjunctive macros is smaller than a set of linear macros. In addition, they argue that the addition of iterative macros doesn't increase the size of a set of operators if each iterative macro consists of sequences of operators from the original set. This is because the original operators need not be matched during planning, only the iterative macros.

Another encouraging result comes from Shavlik's [65] demonstration of the utility of EBL in some recursive domain theories. He showed that in order to acquire some concepts it is necessary to generalise the structure of an explanation that is repeated over a number of iterative applications (the problem of generalisation-to-N). Other methods of EBL seek to relax constraints on the variables in an example. He was able to show that generalising the structure of explanations in the construction of iterative macros can help avoid some of the negative effects of compilation. These negative effects of compilation, grouped together as the utility problem, had questioned whether the use of compiled knowledge such as macros would always slow a planning system down (see for example: [52]).

As we shall see in subsequent chapters, part of our work has led to the development of a compilation method, PROPOSE, that compiles iterative macros and enumerated sequences which can be used to "unwind" the iterative macros direct from a modelled planning domain. This new compilation method PROPOSE is the subject of chapter 5.

### 3.3.5 Sub-goals

We use the term *sub-goal* to refer to a goal that can occur as a pre-condition goal to be achieved in a partial plan during plan generation and that is known to be an intermediate sub-goal of the current conjunctive goal of the planning problem. If sub-goals can be identified before or during planning then they can be used to decompose a planning problem. The idea is that the sub-goal state is a state in which some intermediate sub-goal is true and this state lies on the path between the initial and goal states. The use of sub-goals during planning in this way is analogous to the use of a lemma during theorem proving.

The benefit of using intermediate sub-goals is that it can improve planning performance by reducing search, because the presence of intermediate sub-goals decreases the distance between problem states without increasing the branching factor. The value of using sub-goals was shown by Korf [41]. For example, for a planner that uses an exhaustive search strategy, the search for a problem is:

$$O(b^d)$$

where $b$ is the average branching factor of the search and $d$ is the depth or number of operators in the solution. However, if $k$ intermediate sub-goals are identified, then the search for the same planner is reduced to:

$$O(kb^{d/k})$$

So clearly there is a large potential dividend to be gained from using sub-goals during planning. It is different from the use of macros which also reduce the distance between problem states but they can, as we have seen, also increase the branching factor by increasing the total number of operators to consider. Alternatively, learning control rules can decrease the branching factor but leaves the distance between problem solving states unchanged.

Ruby and Kibler acknowledged the value of using sub-goals during planning and addressed the problem of learning intermediate sub-goals, with the SteppingStone system [61]. It learned sequences of intermediate sub-goals at planning nodes where protected sub-goals must be clobbered to solve some other goal. The algorithm makes use a general heuristic, called *openness*, that predicts the likelihood of solving the reminder of a set of goals while preserving any goals that have already been solved.

In their work on goal ordering, Cheng and Irani [32] looked at augmenting planning problem goals with additional sub-goals. This is because problem goals are often incompletely specified and in order to find optimal solutions it may be necessary to add extra goals to the goal condition. This is the case for the blocks world version of the Sussman Anomaly which has an initial state and goal condition as shown in figure 3.2 where the left hand side of the figure is the initial state with block $c$ on block $a$ and block $a$ is on the *table*. If the goal condition is represented as:

$$\{on(a,b), on(b,c)\}$$

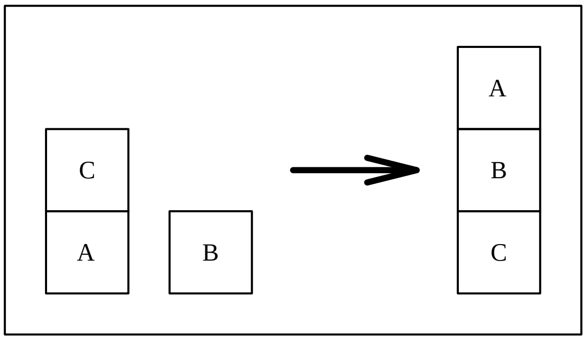


Figure 3.2: Blocks World: Sussman Anomaly

then this is non-serializable given the initial state of the world. This means that there is no serial order in which one goal can be achieved and then the other goal can be achieved from that state. Korf [41] pointed out that this anomaly is not a property of the problem but an artifact of the problem (initial state and goal) description. If the same goal was re-represented as:

$$\{on(c, table), on(b, c), on(a, b)\}$$

then the goal becomes serializable. Interestingly these goal augmentations are generated in the approach of Cheng and Irani [32], Korf [41] and Steppingstone [61].

### 3.3.6 Condition – Action Pairs

It has been suggested that during compilation the consequences of a domain model could be completely enumerated so that search in planning could be completely replaced by some sort of "look-up" of the appropriate sequence of actions for some set of conditions (see for example: [27]). Such an approach would result in planning systems that were ideal in that they are correct, optimal and complete. An example of such an approach is the system by Pearce [59], which compiles model-based fault simulations into an exhaustive rule-base for the purposes of diagnosis. However this approach is not useful in many cases because both the time complexity for the enumeration and the space complexity for the storage of the look-up table are intractable.

Some researchers have tried to overcome the complexity of enumeration by only enumerating certain consequences of the domain model. One example of this can be found in the approach of Flann [25] for enumeration in game domains such as chess and checkers. The approach is to use a heuristic best-first search to control the enumeration so that there is a trade-off between the time taken for compilation and the coverage of the domain. In addition abstraction is used to control the space needed to represent the enumerated values.

### 3.3.7 State Invariants

Some of the compilation methods that we have seen so far rely on a *state invariant* being included as part of the domain theory. A state invariant is a condition which must be satisfied by valid states of a modelled planning domain. The invariant can be negative and specify those things that cannot be true of a valid state or it can be positive and specify those things that must be true of a valid state.

Examples of approaches that we have reviewed which use state invariants include static generation of control rules in STATIC [21] and the problem specific approach to goal ordering of Cheng and Irani [33]. These compilation methods expect that a state invariant was encoded by the domain modeller when the planning domain was modelled.

Some observers have criticised the use of user supplied state invariants because, they argue, requiring a user to specify, for example, things that cannot be true simultaneously in a state, is expecting too much of the domain modeller. For example, Tadepalli and Isukapalli [71] argue that if a domain modeller is required to supply what they call "domain specific simplification rules" then there is no reason why they shouldn't be required to supply useful information about plan generation for that domain as well.

However, we argue that supplying a state invariant for compilation is analogous to supplying a state invariant as part of the formal specification of software systems and consequently is as easy (or as difficult!) as encoding operators in terms of pre- and post-conditions. Also, as we shall argue later, a state invariant forms a check on the consistency of the domain model. The class of planners that we are interested in assume that input planning problems are consistent and state invariants could be used to automatically test that this is the case.

In addition, state invariants as supplied by the user, in for example [21], can be compiled direct from the domain model. A method for automatically compiling a state invariant from a set of operator representations was introduced in the REFLECT system of Dawson and Siklossy [14]. In a pre-processing phase, REFLECT generated a negative state invariant that consisted of pairs of predicates that could not occur in a valid state of the world. A pair of predicates belong to the state invariant if there is no way, through application of the operators, to make both predicates simultaneously true. So for example, in a STRIPS robot planning domain the following two predicates:

$$open(Door)$$
$$closed(Door)$$

(where *Door* is a variable that can be bound to objects of sort *door*) can not be achieved in the same state because every operator that achieves $open(Door)$ clobbers $closed(Door)$ and vice versa.

REFLECT used the compiled state invariant to reject partial plans whose outstanding pre-conditions described a state of the world which violated this invariant. Other work which rejects partial plans on the basis of state violations include [19]. These approaches are discussed in more detail in section 4.1 on page 44.

## 3.4 Summary

The main problem that our thesis work addresses is the inefficiency of generative domain independent planning systems. In this chapter we have reviewed a variety of strategies for improving the performance of DIP's within our framework of compilation. The prevailing strategy is to use *prior* or *current* planning experience to transform a modelled planning domain and hence improve planner performance. We refer to these two strategies of compilation (or learning) as "post-event" and "on-line" compilation respectively. In this thesis we are arguing for a move away from these approaches towards off-line compilation. At the heart of this thesis is the contention that for planning systems which reason about "complete" domain models, off-line compilation holds the key to making them more efficient.

In support of this thesis our goal is to develop new methods for compilation that are independent of planning and which can be applied a priori or "off-line" direct to the domain model but that are as powerful as established on-line and post-event methods. Within the context of this thesis we have focussed on the specific problem of developing methods for off-line compilation of knowledge that can be used to choose amongst alternatives during partial plan specialisation. We sub-divide this into the following problem areas:

- to develop an off-line method of compiling goal orders that can be used for selecting which goals to achieve next during partial plan specialisation
- to develop an off-line method for compiling knowledge that can be used to select operators (and their bindings) that can be used to achieve outstanding predicates in the context of certain planning goals, during partial plan specialisation

During the course of our work we have explored a number of methods of domain model compilation. This work has led to the development of two new complimentary off-line compilation methods, which we call PRECEDE and PROPOSE. These methods are problem distribution independent since they are generated without the need for training examples. PRECEDE is a method for transforming a domain model by making explicit necessary orders between predicates in the domain model. This is the subject of chapter 4. PROPOSE is a method for transforming a domain by compiling iterative macros and enumerated sequences that can be used to "unwind" the iterative sequences of operators. Compilation with PROPOSE is the subject of chapter 5.

# Chapter 4

# PRECEDE

Central to this thesis is the argument that off-line compilation offers a way of increasing the efficiency of domain independent planners. This represents a move away from prevailing approaches that use current planning experience (for example [36] and [33]) and prior planning experience (for example [65] and [67]) to improve subsequent planning performance. In the previous chapter we concluded that there is a need for an off-line method of compiling *goal orders* that are useful in improving the performance of generative domain independent planners. The subject of this chapter is the novel compilation method PRECEDE that we have developed to meet this specific requirement. PRECEDE is an off-line method of compilation that outputs goal orders which are based on necessary interactions between goals.

The aim of this chapter is to introduce and define the compilation method PRECEDE. To put the work in context, we start with a discussion of motivating work (section 4.1). This is followed (section 4.2.1) by a discussion of the negative state invariant that is used by PRECEDE during compilation. The output of PRECEDE compilation is an order on goals in a planning domain and in section 4.2.3 the properties of these orders are precisely defined. This is followed by a specification of an implementation that generates these orders (section 4.2.4). Finally, we evaluate the goal orders statically (section 4.3). This evaluation takes two parts, firstly, an analysis of the properties of PRECEDE orders and secondly an assessment of the utility of the compilation. This analysis will then feed into the design of the empirical evaluation of PRECEDE which is the subject of chapter 7.

## 4.1 Motivation

Our work with PRECEDE was motivated by a desire to develop an off-line method for compiling goal orders direct from a model of a planning domain without the need for training examples. The starting point for our work was experience with ABGEN, our implementation of the ALPINE algorithm [36]. Initial experiments showed that compilation with ABGEN could significantly improve planning performance in some modelled planning domains but not in others, one example being the FM warehouse planning domain *WW* [50]. Another area of interest that stemmed from our experience with ABGEN was the difference between problem specific and problem independent abstraction hierarchies. As discussed earlier (see: section 3.3.1) ALPINE generates problem specific hierarchies because this reduces the likelihood of the abstraction hierarchy "collapsing". We wanted to develop a problem independent method of generating goal orders since then the compilation need only be done once for a particular modelled planning domain. Our solution was inspired by approaches that use information about valid states to prune partial plans during planning, such as [19].

We have already seen the ALPINE algorithm in section 3.3.1 but in this section it is considered in more detail so that the conditions under which the abstraction hierarchies "collapse" can be appreciated. The problem specific ALPINE algorithm takes as input a set of operators $O$ and a conjunctive planning goal $G$. It outputs a hierarchy which is a sequence whose elements are sets of predicate names. The main steps in generating a problem specific ALPINE hierarchy are paraphrased, using our model of planning, as follows:

- The first step of the algorithm, is to construct a directed graph $D$, that represents the relationships between predicates that are "relevant" (appear in sequences of operators that achieve some $g \in G$) in the context of the problem goal $G$. The directed graph $D$ is a pair $(A, N)$, where $A$ is a set of arcs between nodes in $D$ and $N$ is the set of nodes in the graph. Nodes are the names of predicates in the domain and the arcs are labelled with the names of operators. Arcs in the graph are of the form $(a, b)$ where either $a \in G$ or $a$ is a pre-condition of an operator in a sequence of operators that achieve some $g \in G$ and for some $o$ which is an achiever of $a$, $b$ is either in $o.e$ or $o.pre$

- The second step of the problem specific ALPINE algorithm is to identify any strongly connected components[1] in the directed graph $D$ and then construct a reduced graph $D_r = (A_r, N_r)$. Nodes in $D$ that were in a strongly connected component correspond to a single node in the reduced graph $D_r$ and there is an arc $(a, b)$ in $D_r$ if there was an arc between the corresponding nodes in $D$.

- The final step of the problem specific ALPINE algorithm is to transform the partial order over the nodes in $D_r$ into a total order by topologically sorting them. If the topological sort can result in more than one total order a set of heuristics are used to select one sequence. This sequence is the hierarchy $H$, where levels in the hierarchy are the sets of predicates that formed the nodes in the reduced directed graph $D_r$.

From this overview of the ALPINE algorithm it can be seen that the "granularity" of an ALPINE hierarchy, in other words the number of elements in the sequence that forms the hierarchy, depends on the number of nodes in the reduced directed graph $D_r$. In the worst case, the entire graph $D$ will be strongly connected and there will only be one level in the hierarchy. In other cases, there will be few nodes in $D_r$ resulting in a flat hierarchy with few abstraction levels. This problem that causes ALPINE to generate "flat" hierarchies has been referred to as the *collapsing problem* (see for example: [23]).

Intuitively, the likelihood of this problem occurring is related to the way that a planning domain has been represented. This has been supported by a number of researchers such as Holte and colleagues who showed conditions under which abstraction generation was free from the effect of representation [60]. Also, Fink and Yang demonstrated that the

[1] A strongly connected component of a directed graph $D$ is a directed graph $D' = (A', N')$ such that: $D' \subset D$; and for all $n, m \in N'$ there is a path from $n$ to $m$ (where path is defined as follows: $path((a, b), A')$ if $(a, b) \in A'$ or $\exists\, z \in N' : path((a, z), A') \wedge path((z, b), A')$).

degree of interaction between the pre- and post-conditions of a set of domain operators can contribute to hierarchy collapse [23]. Another factor in the collapsing problem, as noted by Knoblock, is that ALPINE hierarchies are based on *possible* interactions between goals [36, pp 127]. Since the set of possible goal interactions is a superset of the necessary interactions, this means that ALPINE considers more interactions than may actually occur in the context of solving some particular planning problem. Our approach was to reduce the likelihood of hierarchy collapse by reasoning just about necessary interactions between predicates.

The approach that we took to identifying necessary interactions was in part inspired by methods that reduce search during planning by rejecting partial plans with inconsistent outstanding pre-conditions (that describe inconsistent states of a modelled planning domain). This is the approach taken by for example Dawson and Siklossy with their REFLECT system [14] and Drummond and Currie [19]). Before we define PRECEDE in the next section (section 4.2) we shall consider this work in more detail.

These approaches require information about what constitutes an inconsistent state within a model of a planning domain. As observed by Drummond and Currie [19] these are typically binary constraints on pairs of predicates that cannot be true in the same state, such as a *door* cannot be open and closed in the same state. They questioned the utility of using constraints containing larger sets of predicates because the efficiency of consistency checking is determined by the number of constraints on predicates and the size of each constraint. Likewise, we restricted our discussion to constraints on pairs of predicates that cannot be achieved in the same state.

For the moment we say that a set of predicates are inconsistent if they contain a pair of predicates that cannot be achieved in the same state (this will be precisely defined in section 4.2.1). For example, in the STRIPS robot planning world $SW$ (defined in appendix A) the following predicates cannot be achieved in the same state:

$$(closed(Door), open(Door))$$

because all operators that achieve $closed(Door)$, clobber $open(Door)$ and vice versa (where $Door$ is a variable that can be instantiated to any object of sort *door*). So any set of predicates that contains $closed(Door)$ and $open(Door)$ is inconsistent.

The planning algorithms presented in both [14] and [19] use inconsistent state information to identify and reject partial plans during planning. To illustrate the sort of approach that they take we shall give a brief example, again in the robot planning world $SW$. Suppose that we start with the following conjunctive planning goal:

$$\{robot\_in(room6), in\_room(box1, room1)\}$$

and that for each goal $robot\_in(room6)$ and $in\_room(box1, room1)$ there is one achiever operator which is $gothrudoor(door56, room6)$ and $pushthrudoor(box1, door12, room1)$ respectively. This situation is represented in figure 4.1 where the special operator *goal* is shown in the box at the right hand side of the figure with the pre-conditions $robot\_in(room6)$ and $in\_room(box1, room1)$. The achiever operators for each of these pre-condition goals are shown in boxes along with the pre-conditions of the operator. The pre-conditions of $gothrudoor(door56, room6)$ are:

$$\{robot\_in(room5), open(door56)\}$$

and the pre-conditions of $pushthrudoor(box1, door12, room1)$ are:

$$\{at\_door(box1, door12, room1), open(door12), robot\_near(box1),$$
$$robot\_in(room2)\}$$

Lets now suppose that the goal $robot\_in(room6)$ (from the top-level planning goal) is achieved first by the operator $gothrudoor(door56, room6)$ which moves the *robot* from $room5$ to $room6$. Since an operators pre-conditions must be true before it can be applied, we know that the predicates $robot\_in(room5)$ and $open(door56)$ will be true when the operator $gothrudoor(door56, room6)$ is applied.

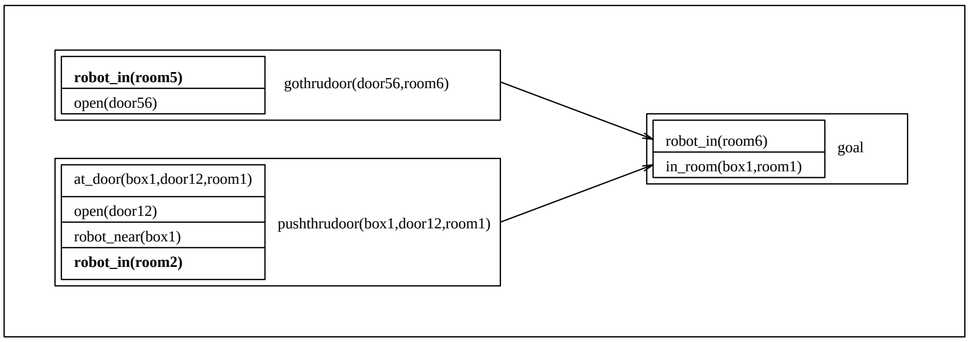


Figure 4.1: Example Partial Plan

The goals that remain to be achieved at this stage, called the outstanding goals, are the original goal $in\_room(box1, room1)$ and any pre-conditions of the operator $gothrudoor(door56, room6)$ that are not true from the initial state. Suppose that we choose to try and achieve $in\_room(box1, room1)$ next using the operator:

$$pushthrudoor(box1, door12, room1)$$

which moves $box1$ and the *robot* from $room2$ to $room1$. From the pre-conditions of this operator we know that, amongst other things, $robot\_in(room2)$ must be true of the state to which *pushthrudoor* is applied. At this stage we have a partially developed plan containing the two operators, as represented in figure 4.1.

The outstanding pre-conditions of this partially developed plan are all the unachieved pre-conditions of the operators in the plan. This set includes $robot\_in(room5)$ and $robot\_in(room2)$ which describes an inconsistent state because a *robot* cannot be in two rooms at the same time. When a situation such as this is identified during planning the algorithms described in both [14] and [19] reject the current partial plan.

In our work with PRECEDE we have taken a different approach, motivated by the idea that inconsistent state checking could be used prior to planning, in other words during off-line compilation. It seemed likely that we could reason about the states described by operator pre-conditions (pre-condition states as defined in section 2.2.2.1) and inconsistent states that could arise whilst making a pre-condition state true.

The basic idea is that for any operator $o$, it may be possible to identify useful orders in which to achieve its pre-conditions, $o.pre$, by first checking the consistency of its pre-conditions with the pre-conditions of the achiever operators of each $p \in o.pre$. Then orders in which to achieve the pre-conditions can be formulated on the basis of any inconsistencies that are detected.

As an illustration, consider the situation in figure 4.2 that shows three example operators $operator1$, $operatorX$ and $operatorA$ with their pre-conditions and suppose that we are trying to identify an order to achieve the pre-conditions of $operator1$. The arrows between operators show how pre-condition goals are achieved in the goal directed search. We can see that there is one achiever for both of the pre-conditions of $operator1$, these are $operatorX$ for $p1$ and $operatorA$ for $p2$. We'll also suppose that we know that the pair of predicates $(px, p2)$ cannot be achieved in the same state (in other words there are constraints on this pair of predicates).

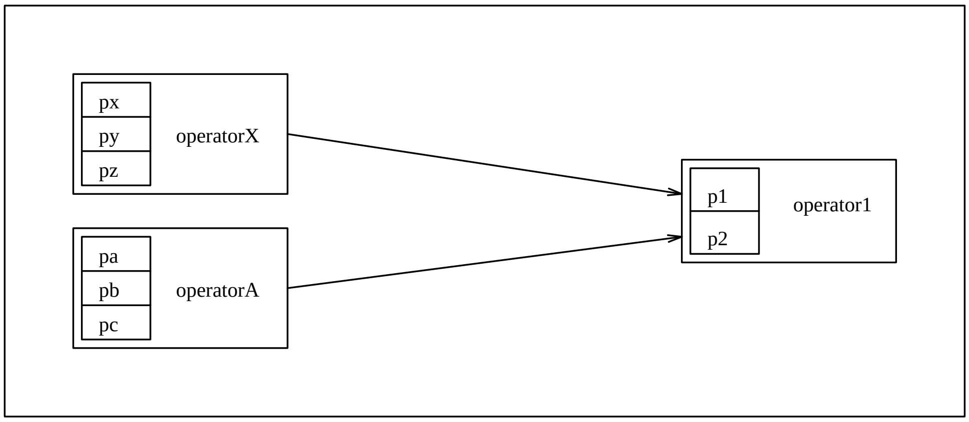


Figure 4.2: Motivating Example for PRECEDE

To identify orders in which to achieve the pre-conditions of $operator1$ we consider the achievers of its pre-conditions. Lets look first at pre-condition $p1$ for which the only achiever operator is $operatorX$. Using $operatorX$ to achieve $p1$ means that in the state before it is applied the predicates $px$, $py$ and $pz$ must all be true. What we are interested in is whether this pre-condition state is consistent with the other pre-conditions of $operator1$. In other words, is the set:

$$\{px, py, pz, p2\}$$

consistent ? It turns out that it is not because, from the inconsistency information, we know that $px$ and $p2$ cannot co-exist in the same state. This means that if we achieve $p1$ with $operatorX$ (the only way to achieve it), $p2$ will be clobbered. Assuming that the opposite case is not true[2] (the only way to achieve $p2$ clobbers $p1$), then $p2$ must be achieved *after* $p1$ in the sequence of operators that achieve the pre-conditions of $operator1$. So the pre-conditions of $operator1$ are ordered so that $p1$ is achieved earlier in the sequence of operators than $p2$.

[2] In section 4.2.3 we show that this can not occur for a consistent operator set

## 4.2 PRECEDE

In the previous section we outlined the motivation for the development of PRECEDE, a compilation method that outputs goal orders for planning which are based on necessary interactions between goals. It requires information about inconsistent states of a modelled planning domain to identify these interactions between goals and this information is supplied as a negative state invariant.

This section is divided into three parts. We start in section 4.2.1 by considering the negative state invariant that is used by PRECEDE. Then in section 4.2.3 we define the properties of the goal orders that are compiled by PRECEDE. Finally, in section 4.2.4 we outline a procedure that generates PRECEDE goal orders.

### 4.2.1 Negative State Invariant: $IN$

$IN$ is a negative state invariant used by PRECEDE during compilation. It is *negative* in the sense that it states what cannot be true of valid states of some modelled planning domain. For example, the invariant for a STRIPS robot planning world might include a constraint that $robot\_in(R_1)$ and $robot\_in(R_2)$ (where the variables $R_1$ and $R_2$ can be instantiated with any different instances of sort *room*) are not permitted in valid states of this planning domain. The purpose of this section is to precisely define the properties of this negative state invariant before we define what it means for sets of predicates to be *inconsistent*.

It is important to note that $IN$ can either be supplied by the user or automatically generated from the set of operators for a modelled planning domain, so long as all elements of the invariant satisfy the membership condition that we define below. There are advantages to both input forms of $IN$ and we will return to these at the end of this section.

Elements of $IN$ are pairs of predicates $(a, b)$ that cannot both be achieved in the same state of a planning domain. The decision to limit $IN$ to pairs of predicates is based on the observation that many constraints in the sorts of planning domains we have considered are binary. For example, in the warehouse planning world $WW$, objects of sort *crate*, *driver* and *truck* are only ever at one location in one state, objects of sort *truck* and *crane* are either unloaded or loaded, a *driver* is either on the floor or in the *truck* and so on. This is also the case for other frequently used planning domains such as the STRIPS robot planning world [19]. In addition, the utility of using invariants that constrain larger sets of predicates has been questioned because the cost of checking consistency is dependent on the size and number of the constraints (see for example: [19]). However, we see no *theoretical* reason why larger sets of predicates could not be considered. Our decision to limit $IN$ to pairs of predicates is solely for reasons of efficiency.

Our definition of $IN$ is based on a *ground* negative state invariant, $IN_g$ which is defined as follows:

$$\begin{aligned} IN_g = \{ & (i, j) \in preds_g \times preds_g \mid \\ & necessary\_clobber(i, j) \wedge necessary\_clobber(j, i)\} \end{aligned}$$

Members of $IN_g$ are pairs of ground predicates, where predicates are ground if all of their arguments are instantiated with constants. The set $preds_g$ contains all the predicates within a planning domain. It is assumed that predicate arguments are universally quantified over objects of some fixed sort. For example, the STRIPS domain predicate $in\_room(B, R)$ has two arguments $B$ and $R$ which are universally quantified over the sorts $box$ and $room$ respectively. So if the sort $box$ contains the object $box1$ and the sort $room$ contains the object $room6$, then the set $preds_g$ will contain the predicate $robot\_in(box1, room6)$.

The definition above tells us that for any pair of predicates $(i, j)$ in $IN_g$, there is no way to achieve $i$ and $j$ in the same state, because achieving $i$ will necessarily clobber $j$ and achieving $j$ necessarily clobbers $i$. We define this condition, where there is no way to achieve one predicate without clobbering another as follows:

$$\begin{aligned}&necessary\_clobber(a, b) = \\ &\quad \forall\, s \in seqs(OS) : a \in s.e^{+} \Rightarrow b \in s.e^{-}\end{aligned}$$

which states that $a$ necessarily clobbers $b$ if *every* sequence of operators that achieve $a$, clobber $b$ (the set $seqs(OS)$ and its post-conditions are defined on page 16). Finally, we can define membership of $IN$ as follows:

$$IN = \{(x, y) \in preds \times preds \mid \forall\, s \in \mathbb{P}\, SUBS : (x, y)_s \in IN_g\}$$

which states that a pair of predicates $(x, y)$ is a member of the negative state invariant, $IN$, so long as there exists a legal substitution, $s$, such that $(x, y)_s$ is a member of the ground negative state invariant. The definition assumes the presence of a set of legal substitutions, called $SUBS$, that bind variables within predicates to constants of the correct sort (this was defined on page 14).

If we consider the definition of $necessary\_clobber(a, b)$ we can see that it covers *all* sequences of operators that achieve $a$. So, to show that $necessary\_clobber(a, b)$ is true it tests that every way of achieving $a$, clobbers $b$. Since membership of $IN$ for a pair of predicates $(a, b)$ depends on both $necessary\_clobber(a, b)$ and $necessary\_clobber(b, a)$ being true this means that whenever $a$ is achieved $b$ is clobbered and whenever $b$ is achieved $a$ is clobbered. On the basis of this we contend that the definition of $IN$ is *sufficient* to capture all pairs of predicates that cannot be achieved in the same state.

To illustrate the operational definition of $IN$, we shall work through a brief example from the robot planning world $SW$ for the pair of predicates:

$$(closed(D), open(D))$$

From the definition of $IN$, the pair of predicates are in $IN$ if there is some substitution $s \in SUBS$ under which the pair of predicates are in $IN_g$. Lets suppose that there is only one substitution $s$ which is $\{door12/D\}$, then the pair $(closed(door12), open(door12))$ are in $IN_g$ if the following is true:

$$\begin{aligned}&necessary\_clobber(closed(door12), open(door12)) \wedge \\ &necessary\_clobber(open(door12), closed(door12))\end{aligned}$$

Lets look at each of these in turn. To show that $open(door12)$ is necessarily clobbered in the course of achieving $closed(door12)$ we start with the achievers of $closed(door12)$

which consists of one operator $\{close(D)\}$. The achievable pre-conditions of $close(D)$ are:

$$close(D).pre = \{open(D), robot\_at(D, R)\}$$

where $D$ is a variable that can be instantiated by objects of sort *door* and $R$ is a variable that can be instantiated by objects of sort *room*. The post-conditions of $close(D)$ are as follows:

$$close(D).e^+ = \{closed(D)\}$$
$$close(D).e^- = \{open(D)\}$$

There are two substitutions for the predicates in this operator, as follows:

$$s1 = \{door12/D, room1/R\}$$
$$s2 = \{door12/D, room2/R\}$$

so $necessary\_clobber(closed(door12), open(door12)$ can be shown to be true since under both substitutions $open(door12)$ is in the negative effects of the operator $close(door12)$. By a similar process we can show that the reverse also holds, that $open(D)$ necessarily clobbers $closed(D)$. Since there is no way to simultaneously achieve both predicates the pair $(closed(D), open(D))$ is a member of $IN$.

An $IN$ for the robot planning world $SW$ (defined in appendix A) is given below.

$$
\begin{aligned}
IN = \{ & \\
& (open(D), closed(D)), \\
& (at\_door(B, X, P), at\_door(B, X, Q)), \\
& (robot\_in(D), robot\_in(E)), \\
& (robot\_near(D), robot\_near(E)), \\
& (at\_door(B, Y, \_), at\_door(B, X, \_)), \\
& (at\_door(B, \_, X), in\_room(B, Y)), \\
& (in\_room(X, Y), in\_room(X, Z)), \\
& (robot\_at(\_, Y), robot\_at(\_, Z)), \\
& (robot\_at(\_, Y), robot\_in(Z)), \\
& (robot\_at(Z, X), robot\_at(Y, X))\}
\end{aligned}
$$

In the invariant, we use a number of conventions that govern the way that variables are represented. In keeping with the rest of the thesis we use the convention that capital letters denote variables. Pairs in the invariant are modelled as logical terms so the scope of variables is limited to predicate pairs, in other words they only have scope within elements of the invariant, not throughout the invariant. It is implicit that *different* variable letters denote different variables, so $X \neq Y$ wherever $X$ and $Y$ are different variable letters within a pair of predicates. The anonymous variable "_" in predicate arguments is used to signify that the instantiation of this variable is unimportant in the context of some pair of predicates in $IN$.

So the invariant states, for example, that $open(D)$ and $closed(D)$ can not be made simultaneously true in any state of this planning domain. Also, it is not possible to achieve a state where the *robot* is both $robot\_at(\_, Y)$ and $robot\_at(\_, Z)$, where $Y$ and $Z$ are variables of different objects of sort *room*.

As we stated earlier, the negative state invariant can be either user supplied or automatically generated from the set of operators $OS$ for a modelled planning domain. Both forms of invariant are useful because they can reveal problems in the specification of operators in $OS$ and as such they can form part of the validation of $OS$. For instance, if the domain modeller includes $(a, b)$ in $IN$ but it is possible to achieve $a$ and $b$ in some state of the modelled planning domain then the specification of the operators does not accord with the invariant and this suggests that the domain modeller has incorrectly specified the operators. On the other hand, the automatic generation may throw up inconsistent pairs of predicates that were unintended by the domain modeller. In both cases the domain modeller would need to reconsider the way that the set of domain operators were specified.

A limitation of the user supplied state invariants is that the domain modeller may not identify all of the pairs of predicates that are members of $IN$. For an automatically generated state invariant, there is more certainty that all pairs of predicates in $IN$ will be identified but there is no cross check on the way that the operators have been specified since the invariant will correspond to the operators. However, it does seem that the form of the automatically generated $IN$ is much easier for the domain modeller to validate than the operators themselves, so the automated state invariant may be useful to the domain modeller during operator validation.

During initial development of PRECEDE all the state invariants for the example planning worlds $SW$, $SW^2$ and $WW$ were supplied by the domain modeller but at a later stage of development a procedure was developed that could automatically generate the state invariants. This was used to generate the state invariant for $SW$ that was used in the empirical testing of $SW$ which is the subject of chapter 7. A brief description of the procedure for invariant generation is given in appendix A.3 and the state invariants that were used in the empirical tests are listed in appendices A.3, B.3 and C.3.

### 4.2.2 Inconsistent Predicates

Now we can define what it means for sets of predicates to be *inconsistent*, in terms of the negative state invariant, $IN$. We define this as a function, given by:

$$\begin{aligned}&inconsistent(a, b) = \exists\, s \in \mathbb{P}\, SUBS,\ \exists (i, j) \in IN :\\ &\qquad (i_s \in a_s \wedge j_s \in b_s) \vee (i_s \in b_s \wedge j_s \in a_s)\end{aligned}$$

which means that the input sets of predicates $a$ and $b$ are *inconsistent* if there is a pair of predicates $(i, j)$ in $IN$ such that under some substitution $i$ is a member of one of the input sets of predicates and $j$ is a member of the other input set of predicates. As before we assume the presence of the set of legal substitutions SUBS in the definition. If the two input sets of predicates are found to be *inconsistent* then this means that there is no way that *all* of the predicates can be made true in the same state.

To illustrate this, we'll consider an example from the robot planning world $SW$, where we test if the sets of predicates $p$ and $q$ given below, are $inconsistent(p, q)$:

$$p = \{in\_room(A, B)\}$$
$$q = \{in\_room(A, C), open(D), closed(E), robot\_at(D, F)\}$$

From the $IN$ given on page 49 we identify the pair of predicates:

$$(i, j) = (in\_room(X, Y), in\_room(X, Z))$$

If we now suppose that $s$ is a legal set of substitutions for the variables in the input sets of predicates and pair $i$ from $IN$:

$$s = \{big\_box/A, room3/B, room6/C, door12/D, door23/E, big\_box/X, \\ room1/F, room3/Y, room6/Z\}$$

then the sets of predicates are indeed *inconsistent* because $i_s \in p_s$ as follows:

$$in\_room(big\_box, room3) \in \{in\_room(big\_box, room3)\}$$

and $j_s \in q_s$ as follows:

$$in\_room(big\_box, room6) \in \{in\_room(big\_box, room6), open(door12), \\ closed(door23), robot\_at(door12, room1)\}$$

Using this definition of *inconsistent*, we can also show that all pairs of predicates in $IN$ are *inconsistent*. This is because for any $(x, y) \in IN$, there exists some element of $IN$ (itself!) and some substitution $s$ such that: $x_s \in \{x\}_s$ and $y_s \in \{y\}_s$.

### 4.2.3 PRECEDE: Goal Order Definition

The purpose of this section is to define the properties of $P$, the goal order that is output by PRECEDE. We define $P$ as follows:

$$P = \{(x, y) \in preds \times preds \mid achieve\_before(x, y)\}$$

Hence elements in $P$ are pairs of predicates $(x, y)$, from some modelled planning domain for which $x$ "can be achieved before" $y$ during planning. Now lets look at the meaning of "can be achieved before" with the *achieve_before* relation, which we define as follows:

$$\begin{aligned} &achieve\_before(x, y) = \\ &\quad necessary\_before(x, y) \lor \\ &\quad \exists\, z \in preds : necessary\_before(x, z) \land achieve\_before(z, y) \end{aligned}$$

which states the two conditions under which $x$ *can* be achieved before $y$. Either $x$ is "necessarily before" $y$ or $x$ is "necessarily before" some other predicate $z$ for which $achieve\_before(z, y)$ is true. Finally, we define what it means for some predicate to be "necessarily before" another as follows

$$\begin{aligned} &necessary\_before(x, y) = \\ &\quad \exists\, o \in OS : x, y \in o.pre \land \\ &\quad \forall\, a \in achievers(x) : y \notin a.e^{+} \land (inconsistent(a.pre, \{y\}) \lor y \in a.e^{-}) \end{aligned}$$

which states that $necessary\_before(x, y)$ is true under two conditions. The first requires that $x$ and $y$ must appear together in the pre-conditions of at least one operator in

$OS$. The second condition requires that all achievers of $x$ can't be achievers of $y$ and that either the pre-conditions of the achievers of $x$ are inconsistent with $y$ or that the achievers of $x$ clobber $y$.

From the definitions above, we can see that the relation *necessary_before* is at the heart of the goal order $P$. The relation is restricted to pairs of predicates that appear as operator pre-conditions because these are pairs of predicates that *must* be achieved in the same state, namely, for operator application[3]. For any pair of predicates $(x, y)$ that appear together as operator pre-conditions, if *necessary_before*$(x, y)$ is true, then from its definition and the definition of *inconsistent* (defined on page 50), $x$ cannot be achieved from *any* state without clobbering $y$. Hence the goals are ordered so that $x$ *must* be achieved before $y$ in any sequence of operators for some planning goal.

To illustrate the *necessary_before* relation, consider the example below taken from the warehouse planning domain $WW$, where the predicate *crane_at*$(S)$ must necessarily be achieved before *crane_loaded*$(C, S)$:

$$necessary\_before(crane\_at(S), crane\_loaded(C, S))$$

and suppose for this example that the negative state invariant for $WW$ includes:

$$IN = \{(crane\_loaded(X, Y), crane\_unloaded), \ldots\}$$

To show that *necessary_before* is true we must first show that both *crane_at*$(S)$ and *crane_loaded*$(C, S)$ appear as operator pre-conditions. This is satisfied because the following operators have *crane_at*$(S)$ and *crane_loaded*$(C, S)$ in their pre-conditions:

$$\{lift\_down(C, Sh), crane\_stack(C, S, Sh)\}$$

To show that *necessary_before* is true we must also show that none of the achievers of *crane_at*$(S)$ achieve *crane_loaded*$(C, S)$. Since there is only one achiever of *crane_at*$(S)$ with positive post-conditions as follows:

$$move\_crane(S_i, S).e^+ = \{crane\_at(S)\}$$

we know that *crane_loaded*$(C, S)$ cannot be achieved by any achievers of *crane_at*$(S)$.

Finally, we must show that the pre-conditions of the achievers of *crane_at*$(S)$ are *inconsistent* with the set containing *crane_loaded*$(C, S)$ or that the achievers delete *crane_loaded*$(C, S)$. We have already seen that there is only one achiever of the predicate *crane_at*$(S)$, the operator *move_crane*$(S_i, S)$. Its pre-conditions are as follows:

$$move\_crane(S_i, S).pre = \{crane\_at(S_i), crane\_unloaded\}$$

Now we can show that the following is true:

$$inconsistent(move\_crane(S_i, S).pre, \{crane\_loaded(C, S)\})$$

in other words, that the pre-conditions of *move_crane*$(S_i, S)$ are inconsistent with the set containing the predicate *crane_loaded*$(C, S)$. This is the case because there is

[3] We assume that for a consistent operator set $OS$, it must be possible to achieve the pre-condition states of each operator.

an element $(a, b)$ of $IN$ (in fact it is the only element shown above) such that $a_s$ is a member of the pre-conditions of $move\_crane(S_i, S)$ and $b_s$ is a member of the set containing the predicate $crane\_loaded(C, S)$.

This is true, under for example the substitution $s = \{crate1/C, s14/S, s10/S_i\}$, where $a_s$ is a member of the pre-conditions of $move\_crane(S_i, S)$ as follows:

$$crane\_unloaded \in \{crane\_at(s10), crane\_unloaded\}$$

and $b_s$ is a member of the set containing $crane\_loaded(C, S)$, under substitution $s$, as follows:

$$crane\_loaded(crate1, s14) \in \{crane\_loaded(crate1, s14)\}$$

So $crane\_at(S)$ is "necessarily before" $crane\_loaded(C, S)$ and this pair of predicates will be a member of the PRECEDE order $P$ for this modelled planning domain (this can be seen in the goal order $P$ that has been automatically generated for $WW$ which is shown on page 60).

A pair of predicates can also be in $P$ if there is a "path" between $x$ and $y$ in the *necessary_before* orders contained in $P$. In other words the orders in $P$ are *transitive* in the sense that:

$$necessary\_before(a, b) \wedge necessary\_before(b, c) \rightarrow (a, c) \in P$$

Any such order between predicates, such as $(a, c)$ in the example above, is *heuristic*, because we have not shown that in all cases $a$ must be achieved before $c$. For instance in the example above the order $(a, c)$ relies on the presence of $b$ and in situations where $b$ is not present the order between $a$ and $c$ is unknown. The ordering of the predicates so that $a$ can be achieved before $c$ in this case is *heuristic*. Consequently, we say that if *necessary_before* is true for a pair of predicates $(a, b)$ then $a$ *must* be achieved before $b$ and for other orders $(x, y)$ in $P$ we say that $x$ *can* be achieved before $y$.

An important feature of the *necessary_before* relation is that if the following are true:

$$necessary\_before(x, y) \wedge necessary\_before(y, x)$$

then we can conclude that the set of operators $OS$ are not internally consistent. Why is this ? Firstly, from the definition of *necessary_before* we know it must be possible to achieve $x$ and $y$ in the same state, since they are pre-conditions of at least one operator in $OS$. We also know that if *necessary_before* is true for both $(x, y)$ and $(y, x)$ then it will never be possible to achieve them both in the same state because achieving either one *necessarily* clobbers the other.

Now, recall that the negative state invariant contains pairs of predicates that cannot be achieved in the same state, so intuitively $x$ and $y$ satisfy the requirements of membership of $IN$. If this is the case then, from the definition of *inconsistent* (page 50), $x$ and $y$ are inconsistent because as we saw, all pairs of predicates in $IN$ are *inconsistent*. So, if it is the case that $necessary\_before(x, y)$ and $necessary\_before(y, x)$ then the pre-conditions of any operators that include $x$ and $y$ are also *inconsistent*. Therefore a useful by-product of PRECEDE compilation is a way in which inconsistent operators can be identified.

Finally, we summarise the properties of the goal orders in $P$. The order specified by $P$ is *partial* since some domain predicates can be unordered with respect to each other. In fact it can occur that $P = \{\}$ if all of the domain predicates are unordered.

- Orders in $P$ are transitive in the sense that:

  $$((pred_a, pred_b) \in P \wedge (pred_b, pred_c) \in P) \rightarrow (pred_a, pred_c) \in P$$

  or in other words, if $pred_a$ *must* be achieved before $pred_b$ and $pred_b$ *must* be achieved before $pred_c$ then we infer that $pred_a$ *can* be achieved before $pred_c$. Any orders that are part of the transitive closure of the set of orders are heuristic.

- Orders in $P$ are asymmetric in the sense that:

  $$(pred_a, pred_c) \in P \rightarrow (pred_c, pred_a) \notin P$$

  or in other words if $pred_a$ is ordered before $pred_c$ then $P$ cannot contain an order that $pred_c$ can be achieved before $pred_a$.

- Orders in $P$ are irreflexive in the sense that:

  $$(pred_a, pred_a) \notin P$$

  or in other words $P$ cannot contain an order that $pred_a$ must be achieved before itself.

### 4.2.4 PRECEDE: Generating Goal Orders

A system has been implemented that compiles an order over the set of predicates in a modelled planning domain. In this section we give an outline of this implementation, which is encapsulated in a procedure referred to as *precede*.

The procedure *precede* is defined below. It takes two inputs, the negative state invariant $IN$ and the set of domain operators $OS$ for a modelled planning domain. The output of the procedure is the goal order, $P$, whose properties were defined in section 4.2.3.

**procedure** $precede : \ OS \times IN \longrightarrow P$

1. $initialise(G)$
2. $for\ all\ x, y \in pred : \ x \neq y\ do$
   $if\ there\ is\ o \in OS : \ x \in o.pre \wedge y \in o.pre\ then$
   $if\ necessary\_before(x, y)\ then\ add\_directed\_edge((x, y), G)$
3. $strongly\_connected\_components(G, Scc)$
4. $update\_scc(G, Scc, P)$

$return(P)$

The first step of *precede* is to initialise $G$, a directed graph which will be used to represent *necessary_before* relations during processing. A directed graph is a pair $(A, N)$ where $A$ is a set of arcs in the directed graph, initially set to $\{\}$ and $N$ the set of nodes in the graph, also initially set to $\{\}$.

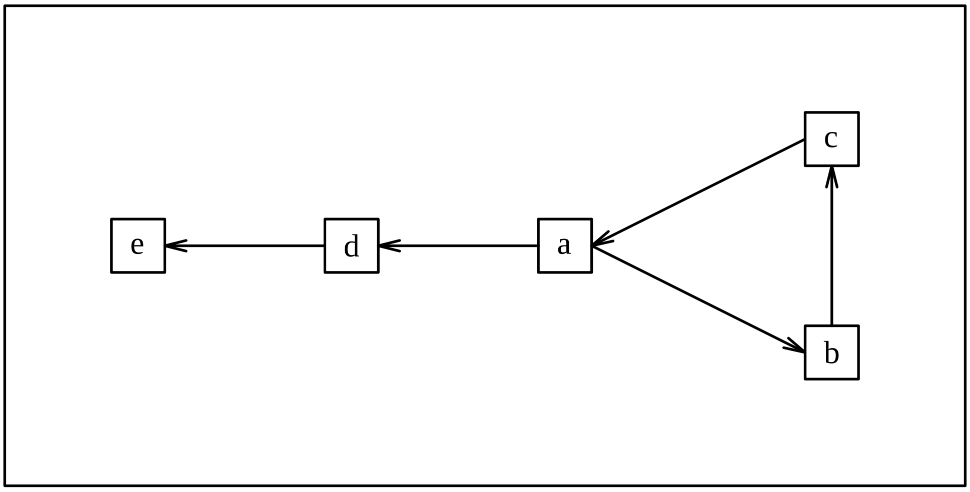


Figure 4.3: Example: Directed Graph generated by *precede*

Step 2 of *precede* considers each pair of predicates $(x, y)$ that appear as operator preconditions. It tests that $x$ and $y$ are not equal, and in so doing ensures that no reflexive orders can be added to $P$. Then for any pairs of predicates for which *necessary_before* is true, the directed graph $G = (A, N)$ is updated. It is updated in two ways, by adding arcs to $A$ and also by adding new nodes to $N$. The arcs in the graph are pairs $(a, b)$ where $a$ and $b$ are predicates for which *necessary_before*$(a, b)$ is true. The nodes $N$ in the graph are the set of predicates that appear in the arcs in $A$.

Once the directed graph $G$ has been constructed, the third step of *precede* identifies any *strongly connected components*, *Scc*, in the directed graph $G$. The properties of strongly connected components were defined on page 43 but one important feature is that there is a path from each node in the component to every other. An efficient algorithm for identifying the strongly connected components of a graph (linear with respect to the size of graph) can be found in for example [26] and was used in the implementation of *precede*. Before looking at what *precede* does with the strongly connected components that are identified we look at their significance.

Recall from the definition of $P$ (page 51) that orders in $P$ are transitive and asymmetric. The significance of identifying strongly connected components in $G$ is that the orders represented within them are asymmetric. For example, consider the graph depicted in figure 4.3 where the nodes in the graph are: $N = \{a, b, c, d, e\}$ and the arcs are:

$$A = \{(a, b), (b, c), (c, a), (a, d), (d, e)\}$$

The nodes $\{a, b, c\}$ form a strongly connected component in the graph, because there is a "path" from each of the nodes in the component to every other. This means that the orders in the strongly connected component are:

$$\{(a, b), (b, a), (a, c), (c, a), (b, c), (c, b)\}$$

These are all symmetric and therefore cannot be included in $P$ since all orders in $P$ must be asymmetric. This is because it is meaningless to have an goal order that says achieve $a$ before $b$ and also achieve $b$ before $a$.

The reason why the graph $G$ can contain "cycles" is because *necessary_before* is only tested between pairs of predicates. However, when pairs of *necessary_before* predicates are combined, the transitivity of the orders means that larger sets of predicates may be ordered with respect to each other. So even though *necessary_before* orders between pairs of predicates are necessary, the accumulation of these orders in $P$ may not be.

Consequently the final step of *precede* is to remove any cycles from the graph $G$. It does this by constructing a reduced graph with nodes and arcs as follows:

- any nodes that formed a strongly connected component in $G$ form a single node in the new reduced graph
- there will be an arc between any nodes in the reduced graph for which there is an arc between corresponding nodes in $G$

The reduced graph that is output is the PRECEDE goal order $P$. For the example $G$ that was given earlier, the output $P$ is as follows (and shown in the figure):

$P = (\{(\{a, b, c\}, d), (d, e)\}, \{\{a, b, c\}, d, e\})$

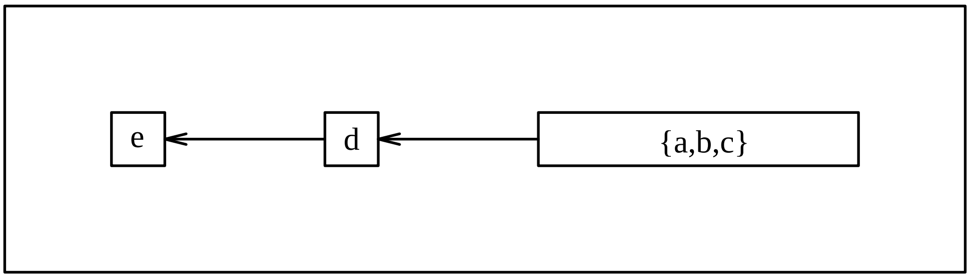


Figure 4.4: Example: Output of *precede*

### 4.2.5 Complexity of Generating Goal Orders

Compilation of a domain model with *precede* is done off-line, in the sense that the domain can be compiled before any planning has occurred in a given modelled planning domain. One consequence of this is that the time taken for the compilation is independent of "on-line" resources during planning. Nevertheless, the abstract complexity of compilation with *precede* is important since the time taken to compile a domain model should be "reasonable". If it was discovered that compilation was exponential, for example, with respect to a critical feature of the domain model, then we would have to restrict the size of domain model that could be compiled by *precede*. In this section we shall consider the cost of inconsistency testing for *necessary_before*, the size of the directed graph in which these relations are represented and the cost of identifying strongly connected components in the graph.

The cost of testing whether sets of predicates are inconsistent depends on the *number* and *size* of elements in $IN$. The size of each element in $IN$ is 2 since analysis was restricted to pairs of predicates. The number of elements in $IN$ depends on the following factors:

- the number of achievable predicates for the domain model (this is the set *preds* which is defined on page 15), let this be $n$

- the average number of constants of each sort of object in the domain model (for example the sorts of objects in the robot planning world include *box* of which there are three constants $box1$, $box2$ and $big\_box$), let this be $k$
- the average number of variable "slots" for each predicate, let this be $v$

Since elements in $IN$ can potentially be any pair of predicates, the maximum number of combinations of pairs is $n^2$. For each of these $n^2$ pairs of predicates the maximum number of ways in which they can be instantiated is $k^v$. So the maximum size of $IN$ is:

$$n^2 \times k^v$$

However, in practice it is unlikely for $IN$ to be even close to the maximum size since this means that every predicate is inconsistent with every other ! For example, in the robot planning world $SW$, where there are 8 predicates and an average of 2 common variables per pair, the maximum number of rules is: $8^2 \times 2^2 = 256$ yet the actual set $IN$ contains only 12 elements.

The size of the directed graph $G$ that is constructed by *precede* to represent necessary before relations between predicates is bounded by the number of predicates in *preds*. If the number of nodes in the graph is $n$ then the maximum number of arcs in the graph is $n^2$ since there can potentially be an arc from every predicate in the graph to every other.

Efficient algorithms exist for identifying the strongly connected components of a graph (see for example: [26]). The method that was used in the procedure *precede* identifies all strongly connected components of the graph by searching the graph in a depth first manner. Each node in the graph need only be visited once so the maximum complexity of this search is:

$$O(max(A, N))$$

which is linear in either the number of nodes or arcs in the graph, depending on which is the larger.

## 4.3 Static Evaluation of PRECEDE Compilation

In later chapters we consider the way that the use of a PRECEDE compiled domain model might improve the performance of a planning system in the class of DIPs (section 6.2.1). We then use this analysis to formulate objectives for a series of empirical tests (chapter 7). In this section our focus is how useful the goal orders appear to be. We start with an analysis of PRECEDE goal orders that have been compiled for the planning worlds $SW$, $SW^2$ and $WW$ and a comparison with hierarchies that have been generated by an implementation of the ALPINE algorithm. Then we assess the relation between PRECEDE and ALPINE, how far the PRECEDE goal orders are consistent with ALPINE hierarchies and why PRECEDE is less likely than ALPINE to "collapse" in certain modelled planning domains.

### 4.3.1 Example PRECEDE Goal Orders

In this section we give some examples of PRECEDE goal orders and compare them with some ALPINE hierarchies to assess how far the orders correspond. The example abstraction hierarchies have been generated using an implementation of the ALPINE algorithm, called ABGEN, which we have used in earlier comparative work (see for example [50]). This implementation generates hierarchies that are similar to those generated by ALPINE, given differences in representation of operators and the domain (see: appendix A.10 on page 168 for an example).

We shall start with a few examples from the robot planning world $SW$. As we have seen, PRECEDE goal orders $P$, are partial orders over the set of predicates for a modelled planning domain. For $SW$ this set is:

$$\begin{aligned} preds = \{ \\ & open(D), closed(D), next\_to(O1, O2), in\_room(O, R), at\_door(O, D, R), \\ & robot\_in(R), robot\_at(D, R), robot\_near(O)\} \end{aligned}$$

The goal order that is generated is as below and depicted in figure 4.5:

$$\begin{aligned} P = \{ \\ & (in\_room(O, R), at\_door(O, D, R)), \\ & (in\_room(O, R), robot\_in(R)), \\ & (robot\_in(R), robot\_at(D, R))\} \end{aligned}$$

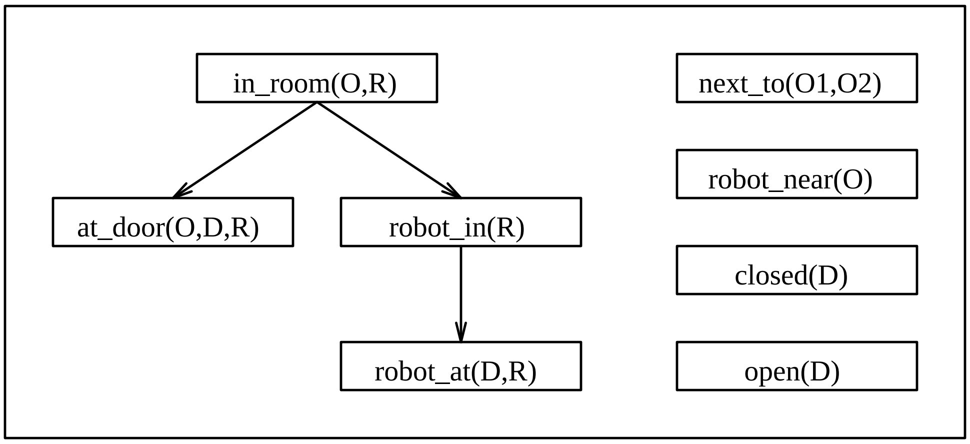


Figure 4.5: $SW$: PRECEDE partial order

So for example, for this world the predicate $in\_room(O, R)$ can be achieved before $robot\_at(D, R)$, but $P$ contains no order between $open(D)$ and $robot\_in(R)$.

Now we can show a few examples of problem specific hierarchies that have been generated by ABGEN and compare them against the goal order generated by PRECEDE. Consider first, the following planning goals, $G1$ and $G2$ (also depicted in figure 4.7):

$$\begin{aligned} G1 &= \{at\_door(big\_box, door45, room4), at\_door(box1, door35, room3)\} \\ G2 &= \{in\_room(box2, room2), robot\_in(room1), in\_room(big\_box, room3), \\ & \quad at\_door(box2, door23, room2)\} \end{aligned}$$

| | G1 | G2 |
|---|---|---|
| 1. | in_room(O,R) | in_room(O,R) |
| 2. | at_door(O,D,R) | robot_in(R) |
| 3. | closed(D),open(D) | at_door(O,D,R) |
| 4. | robot_in(R), robot_near(O) robot_at(D,R), next_to(O,O') | closed(D), open(D) |
| 5. | | robot_near(O), robot_at(D,R) next_to(O,O') |

Figure 4.6: Example Abstraction Hierarchies $SW$

The hierarchies that were generated by ABGEN for the two goals are as shown in table 4.6. In the table the numbered rows (1-5) are the levels of the abstraction hierarchy, level1 is the highest, level5 the lowest and the columns are labelled $G1$ and $G2$ for the two planning goals.

The hierarchies shown in the table are slightly different, which is to be expected since the hierarchies generated by ABGEN are problem specific. For example, the hierarchy for $G2$ has one more level in it, and the relative positions of $robot\_in(R)$ and $at\_door(O, D, R)$ are different, with respect to each other, in the two hierarchies.

How does the PRECEDE goal order correspond to these hierarchies ? The orders in $P$ are consistent with those in the ABGEN hierarchies. By this we mean that there are no orders $(a, b)$ in $P$ (meaning that $a$ can be achieved before $b$) for which $b$ is at a higher level than $a$ in either of the ABGEN hierarchies. For example, in $P$ the predicate $in\_room(O, R)$ is "before" both $at\_door(O, D, R)$ and $robot\_in(R)$ and this is reflected in both of the ABGEN hierarchies since it is in a higher level. There is no order for the predicates $at\_door(O, D, R)$ and $robot\_in(R)$ in $P$ because for some problems $at\_door(O, D, R)$ can be achieved before $robot\_in(R)$ and for other problems $robot\_in(R)$ can be achieved first.

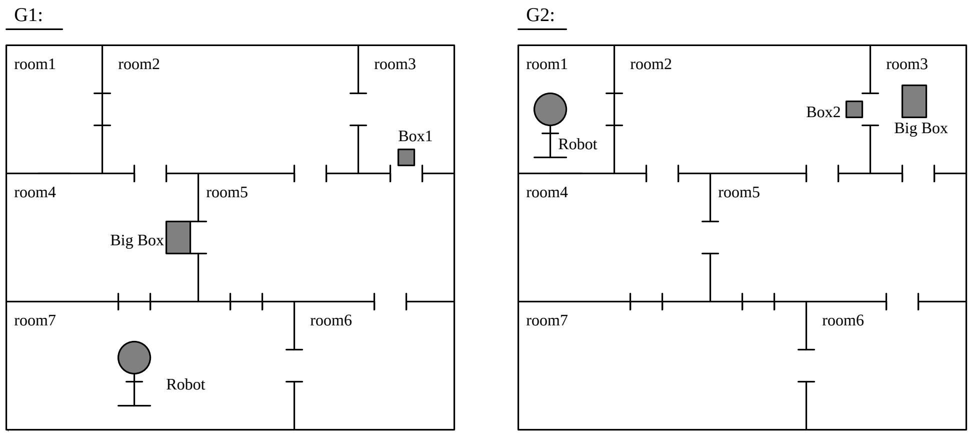


Figure 4.7: Example $SW$ Planning Goals

In fact, we can show that if $P$ contains an order $(a, b)$ then $b$ will never be higher than $a$ in any ABGEN hierarchy. This is an important feature of PRECEDE goal orders and is the subject of section 4.3.2.

Now we shall consider a few examples from the warehouse planning world $WW$. The

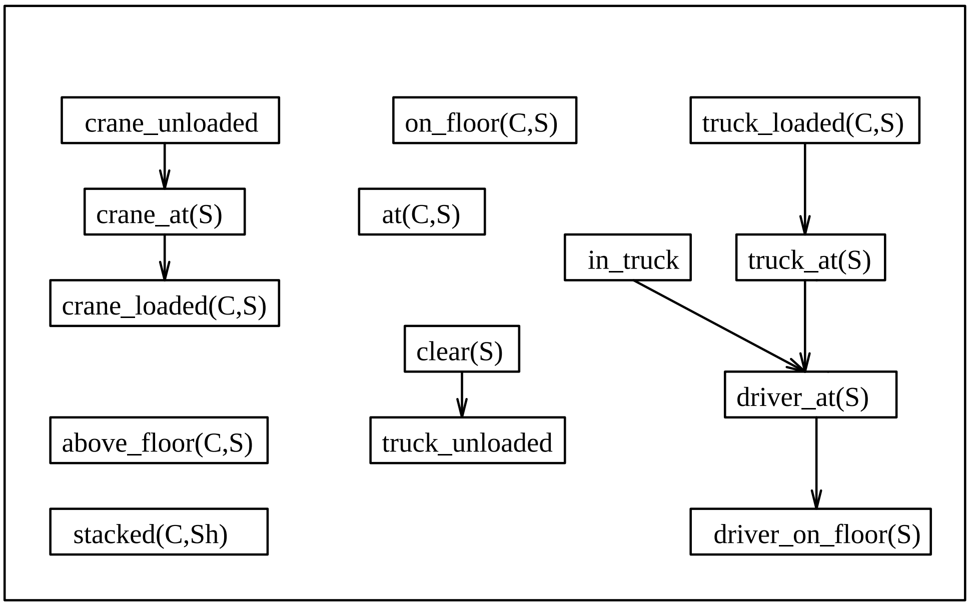


Figure 4.8: $WW$: PRECEDE goal order

set of predicates for this planning domain are:

$$\begin{aligned} preds = \{&stacked(C, Sh), crane\_unloaded, truck\_unloaded, on\_floor(C, S),\\ &driver\_at(S), crane\_loaded(C, S), crane\_at(S), above\_floor(C, S),\\ &at(C, S), driver\_on\_floor(S), in\_truck, truck\_at(S), truck\_loaded(C, S)\}\end{aligned}$$

The PRECEDE goal order $P$ is depicted in figure 4.8. It shows, for example, that $clear(S)$ can be achieved before $driver\_at(S)$ and that there is no ordering between $truck\_unloaded$ and $in\_truck(S)$.

Lets look at how this compares with some example hierarchies generated by ABGEN. For the following planning goals:

$$G1 = \{crane\_at(s6), on\_floor(crate2, s13), clear(s2)\}$$
$$G2 = \{clear(s12), on\_floor(crate3, s6)\}$$

the hierarchies that are generated by ABGEN are shown in table 4.9. We can see that the ABGEN hierarchies are "flat" since there are few levels in each of the hierarchies. Lets consider how they compare with the PRECEDE goal orders. Firstly, the PRECEDE orders look as if they might be more useful during planning because they contain some orders that are finer grained than for ABGEN. An example of this is the series of orders that lead from $truck\_loaded(C, S)$ to $driver\_on\_floor(S)$.

We can also observe that all of the orders in $P$ are consistent with the orders in the ABGEN hierarchies. For goal $G1$ the only order in the abstraction hierarchy places $crane\_at(S)$ at a higher level than $above\_floor(C, S)$ and $crane\_loaded(C, S)$. The same relative order is contained in $P$. For the goal $G2$, all of the predicates are at a higher level than $driver\_at(S)$ and this is consistent with the orders in the PRECEDE compiled partial order $P$.

| | G1 | G2 |
|---|---|---|
| 1. | crane_at(S) | above_floor(C,S), crane_unloaded, crane_loaded(C,S), stacked(C,Sh), truck_loaded(C,S), at(C,S), truck_unloaded, driver_on_floor(S), in_truck, clear(S), on_floor(C,S), driver_at(S) |
| 2. | above_floor(C,S), crane_unloaded, crane_loaded(C,S), stacked(C,Sh), truck_loaded(C,S), truck_unloaded, driver_on_floor(S), in_truck, clear(S), on_floor(C,S) | |

Figure 4.9: Example Abstraction Hierarchies $WW$

### 4.3.2 Relation between PRECEDE and ALPINE

In the previous section we gave a few examples that illustrated the correspondence between goal orders that are compiled by PRECEDE and abstraction hierarchies compiled by ALPINE. In this section we show that the PRECEDE goal orders compiled for a modelled planning domain will *always* be consistent with ALPINE abstraction hierarchies compiled for any problem that could be posed in that domain model.

Our aim is to show that if $necessary\_before(x, y)$ is true for a pair of predicates $x$ and $y$ then $y$ cannot be at a higher level in some ALPINE abstraction hierarchy, $H$, than $x$. This theorem is stated as:

$$necessary\_before(x, y) \Rightarrow level(H, x) \not< level(H, y)$$

Our starting point is a sufficient condition for $x$ to be at a higher (or the same) level than $y$ in some hierarchy $H$. This is given by:

$$\exists\, t \in seqs(OS) : x \in t.e^+ \wedge y \in t.e^- \Rightarrow level(H, x) \geq level(H, y) \qquad (A)$$

which states that if there exists some sequence of operators that achieve $x$ and clobber $y$ then $y$ can not be higher in an ALPINE hierarchy than $x$. This is based on Knoblock's Restriction 4.2 [36, pp 65], where all the "relevant" effects of an operator in some sequence (these are the positive effects of the sequence) are at the same or higher levels in an abstraction hierarchy than any non-relevant effects of the operator (the negative effects of the sequence). Now we can use condition A to prove that the theorem stated above is true. We shall do this by showing that if $necessary\_before(x, y)$ is true for some pair $x$ and $y$ then there does exist a sequence of operators that achieve $x$ and clobber $y$.

So we shall assume that $necessary\_before(x, y)$ is true for $x$ and $y$. From the definition of this condition that was given on page 51 we know that:

$$\forall\, o \in achievers(x) : y \notin o.e^+ \wedge (y \in o.e^- \vee inconsistent(o.pre, \{y\})$$

Now, using the operational definition of inconsistency, which was given on page 50, this can be re-stated as:

$$
\begin{aligned}
&\forall\, o \in achievers(x):\\
&\quad y \notin o.e^{+} \wedge\\
&\quad (y \in o.e^{-} \vee \forall\, s \in seqs(OS):\\
&\qquad\qquad (y \in s.e^{+} \Rightarrow s.e^{-} \cap o.pre \neq \{\}) \wedge (s.e^{+} \supset o.pre \Rightarrow y \in s.e^{-}))
\end{aligned}
$$

which states the two conditions that must be true if $x$ is necessarily before $y$. The first states that $y$ cannot be in the positive effects of any operator that achieves $x$. The second condition states that for all ways of achieving $x$ the result is that $y$ is clobbered. There are two ways that this can happen for some achiever of $x$, either the operator directly clobbers $y$ or all of the sequences of operators that achieve its pre-conditions also clobber $y$ (and any sequences that achieve $y$ clobber at least one of the pre-conditions of the achievers of $x$).

Using the Universal Instantiation Rule (or $\forall$-elimination rule) we let $O$ be some arbitrary element of the set $achievers(x)$. This set is assumed non-empty since we assume that the set of operators $OS$ is consistent and that there is at least one operator to achieve each goal in a modelled planning domain. Using $O$ we now have two conditions that must both be true for $x$ and $y$ if $necessary\_before(x, y)$ is true. The first condition is given by:

$$
y \notin O.e^{+} \tag{B}
$$

and the second condition is given by the following disjunction:

$$
\begin{aligned}
&y \in O.e^{-} \vee\\
&(\forall\, s \in seqs(OS):\\
&\quad (y \in s.e^{+} \Rightarrow s.e^{-} \cap O.pre \neq \{\}) \wedge (s.e^{+} \supset O.pre \Rightarrow y \in s.e^{-}))
\end{aligned} \tag{C}
$$

We shall concentrate on the second term in the disjunction which we have labelled as condition $C$. Condition C is itself a disjunction where the first term is:

$$
y \in O.e^{-} \tag{D}
$$

and the second term is:

$$
\begin{aligned}
&\forall\, s \in seqs(OS):\\
&\quad (y \in s.e^{+} \Rightarrow s.e^{-} \cap O.pre \neq \{\}) \wedge (s.e^{+} \supset O.pre \Rightarrow y \in s.e^{-})
\end{aligned} \tag{E}
$$

If $necessary\_before(x, y)$ is true then either condition $D$ or condition $E$ must be true for $x$ and $y$.

Now we can show that in both cases we can construct a sequence of operators that satisfy the left hand side of condition A that was given earlier. Firstly, lets consider condition $D$. If this is true then we can construct a sequence of operators, that consists only of $O$ (the achiever operator for $x$). If we call this sequence $M$, then we have that:

$$
y \in M.e^{-}
$$

from condition $D$ itself. Also we we have:

$$y \notin M.e^+$$

from condition $B$. Hence, if condition $D$ is true, then the left hand side of condition A is satisfied under the substitution $\{M/t\}$.

Lets now consider the second case, which is given in condition $E$. Using the Universal Instantiation (or $\forall$-elimination) rule we let $S$ be some arbitrary element of $seqs(OS)$ that is an achiever for $O$, such that:

$$S.e^+ \supset O.pre$$

which means that the sequence $S$ achieves all the pre-conditions of $O$. We also know that $O$ is an achiever operator for $x$. So we know that we can construct a sequence, let it be called $M$, by appending $O$ to the end of sequence $S$ and that this sequence is valid (all operator pre-conditions are achieved, as defined on page 16) and achieves $x$. Hence, if condition $E$ is true then we have a sequence $M$ (formed $S$ and $O$) that satisfies the left hand side of condition A given above, under the substitution $\{M/t\}$. $M$ satisfies condition A because:

$$y \in M.e^-$$

since $S$ deletes $y$ (from the second part of the conjunction in condition $E$) and $O$ doesn't add $y$ (from condition $B$). The other part of condition A is satisfied by $M$ because:

$$x \in M.e^+$$

since $O$ is an achiever of $x$.

Therefore, in both cases $D$ and $E$ we have constructed a sequence of operators that satisfy the following sufficient condition for ALPINE hierarchies:

$$\exists\, t \in seqs(OS) : x \in t.e^+ \wedge y \in t.e^-$$

which follows from $necessary\_before(x, y)$ and hence we can conclude that

$$necessary\_before(x, y) \Rightarrow level(H, x) \geq level(H, y)$$

or in other words if $necessary\_before(x, y)$ then $y$ cannot be at a higher level in an ALPINE hierarchy than $x$, which is the correspondence between PRECEDE goal orders and ALPINE abstraction hierarchies that we wished to show.

### 4.3.3 The Collapsing Problem Revisited

Finally, we consider why PRECEDE goal orders do not suffer from the same collapsing problems as ALPINE. This is primarily because PRECEDE goal orders are based on necessary interactions between goals, unlike ALPINE orders which are based on possible interactions between goals. We know that PRECEDE goal orders are based on necessary interactions since from the definition of the negative state invariant $IN$ pairs of predicates $(a, b)$ are members of $IN$ if *all* sequences of operators that achieve one, clobber the other and vice versa.

As we saw earlier a number of researchers have argued that the reason for collapsing in automatically generated abstraction hierarchies is the way that a planning domain has been represented (see for example: [60] and [23]). Another factor, noted by Knoblock, is that ALPINE hierarchies are based on *possible* interactions between goals [36, pp 127]. Since this set is a superset of the necessary interactions, this means that ALPINE considers more interactions than will actually occur in the context of solving some particular planning problem.

PRECEDE is unlike ALPINE in that it outputs goal orders that are based on necessary interactions between predicates. It is our contention that this makes the goal orders less likely to collapse. The reason for this is that if there exists a necessary order, $necessary\_before(x, y)$, between a pair of predicates then those predicates must always be achieved in that order. For there to be no order between $x$ and $y$, in other words for them to collapse, then $necessary\_before(y, x)$ must also be true but this can not be the case for a consistent set of operators.

## 4.4 Summary

PRECEDE is a new compilation method that compiles a goal order $P$ which is based on necessary interactions between predicates. It uses information about inconsistent states of a domain, in the form of a negative state invariant which can be either supplied by the domain modeller or automatically generated from the set of domain operators $OS$. In this chapter we precisely defined the form of the PRECEDE goal order $P$ and overviewed an implementation that generates a goal order that satisfies these properties.

An important feature of PRECEDE goal orders is the way they correspond to ALPINE abstraction hierarchies, in particular if $P$ identifies that $x$ can be achieved before $y$ then $y$ can never be higher in an ALPINE abstraction hierarchy than $x$. Also, we contend that PRECEDE goal orders are less susceptible to "collapse" because the orders are based on necessary interactions between goals.

The static evaluation of PRECEDE goal orders suggests that they will be useful in improving planner performance. In the empirical tests, which are the subject of chapter 7 we would expect to see improvement in planner performance at least as great as that with ALPINE.

# Chapter 5

# PROPOSE

In chapter 3 we reviewed a range of compilation methods and one of the conclusions was that there is a need for off-line compilation methods that can generate useful macro operators direct from the model of a planning domain without the need for training problems. The subject of this chapter is a novel compilation method, called PROPOSE, which we developed in response to this requirement.

PROPOSE compilation outputs iterative macro operators and enumerations for operator bindings direct from a model of a planning domain. These can then be used during planning for selecting operators and bindings of operators. Because they are compiled direct from the domain model without the need for training examples, the compilation is not biased towards any particular problem distribution.

The aim of this chapter is to introduce and define PROPOSE. It starts with a brief look at the motivation for the work and the general ideas that led to the development of PROPOSE (section 5.1). This is followed by a discussion of the use of *sort* information as a means of abstraction during compilation (section 5.2) and the use of the positive state invariant $IP_{sort}$ during compilation (section 5.2). Then in section 5.3.1 we define the form of the iterative macro operators, the sequences of enumerations that are compiled and this is followed by an outline implementation for generating PROPOSE macros and enumerations (section 5.4.1). Finally, in section 5.6 we statically analyse the output of PROPOSE compilation to get some sense of what the macro's and enumerated sequences look like and how they compare to macros that are output by other compilation methods.

## 5.1 Motivation

The general motivation of the work that led to the development of PROPOSE is the need for compilation methods that can acquire macros that are "useful" to a DIP. By useful we mean that when the DIP is bound with a compiled domain model, (e.g. some original domain model plus compiled macro operators), the performance of the planning system is improved.

In section 3.3 we reviewed a number of compilation methods that generate macro operators and we saw that the majority of methods acquire macros from planning traces.

However a problem with this approach is that this can unduly bias the compiled planning system to particular distributions of problems, namely those that it has acquired macros from. There are a few examples of methods that do not require planning examples, such as the approach of Etzioni with the STATIC algorithm for production of control heuristics [21]. Hence our motivation was to develop a method that could acquire useful macros without the need for example planning traces.

Our work with PROPOSE was inspired by our idea that macro operators can be acquired from a model of a planning domain by analysing the *sorts* of objects of the domain and the operators that can be applied to each of the different sorts. We will discuss the nature of sorts in more detail in section 5.2 but for now they can be thought of as similar to *types* in programming languages. As an illustration, the sorts for the robot planning world $SW$, include *room*, *door* and *box* and the objects of sort *room* include $room1$ and $room2$ (see appendix A.4 for a full list). Different operators are applicable to different sorts of objects, for instance the operator *close* is applicable to all objects of sort *door*. Using this sort information, it is possible to identify the possible "situations" that the different sorts of objects can occupy in a modelled planning domain. Then macros can be acquired that span the possible start and end situations for all the sorts of objects in the domain model.

A further idea that has been integrated into PROPOSE is that once iterative macros have been identified the predicate arguments of sorts that are changed by the iteration can be enumerated. These "enumerated sequences" are defined in section 5.3.2 but for now we will just motivate them with a brief example. Suppose that for sort *truck* in the warehouse world, a possible start situation is $truck\_at(A)$ and an end situation is $truck\_at(Z)$ (where $A$ and $Z$ are instantiated with different objects of sort *space*). An iterative macro for this situation might consist of a sequence of the operator $drive(S, S')$ which repeatedly moves the *truck* from one *space* to another. In this case PROPOSE enumerates sequences of instantiations of sort *space* that could be used to "unwind" this iterative macro during planning.

In PROPOSE we have developed a compilation method that assembles sequences of operators (macros), which can contain iterative parts, that are abstract solutions spanning the possible start and end situations for different sorts of objects in the domain. It also enumerates sequences of instantiations of the iterative macros. Of course this approach assumes a model of planning that uses sorts but we feel that this is intuitively valid and certainly it is the case for planning domains that can be found in the literature, such as the robot planning domain in [62] or the machine shop scheduling domain in [36].

## 5.2 Positive state invariants: $IP_{sort}$

PROPOSE uses information about valid states of a modelled planning domain, supplied in the form of positive state invariants for each "sort" in the model. These state invariants are said to be positive because they specify the alternate situations that objects of different sorts can occupy in valid states of the modelled planning domain. PROPOSE uses them to generate the set of possible start and end situations, or *task configurations*, that could arise in any planning problem for a given sort of object. It is this set of task configurations for which PROPOSE compiles macros and enumerated sequences and which we define in sections 5.3.1 and 5.3.2. In this section we set the scene by discussing the nature of sort information assumed by PROPOSE, the positive state invariants for each sort, $IP_s$, and finally the set of task configurations for each sort, $TC_s$.

A sort, $S$, is a set of objects where $s \in S$ is read as *s is an object of sort S*. The sort of an object defines the "behaviour" of the object, so that for each object of a sort $s \in S$ its behaviour is defined by the behaviour of $S$, which we'll refer to as behavioural inheritance. By the behaviour of a sort we mean the operators that can be applied to it and the situations that objects of the sort can occupy. So for example in the robot planning world $SW$, objects of the sort *door* can be opened or closed by the application of the operators $open(Door)$ and $close(Door)$ (where $Door$ is a variable that can be bound to any object of sort *door*). This behavioural inheritance of sorts is useful because it allows us to abstract away detail by reasoning about the behaviour of a sort rather than all the objects of that sort.

PROPOSE assumes that the domain modeller has collected objects of the same kind together into sorts and that this is supplied as part of the domain model. As an example, the sorts for the robot planning world $SW$ are given below, the sorts for the other planning worlds are listed in appendices B.4 and C.4 on pages 174 and 204.

$$box = \{box1, box2, big\_box\}$$
$$door = \{door12, door23, door24, door25, door35, door45, door47, door57, door67\}$$
$$room = \{room1, room2, room3, room4, room5, room6, room7\}$$
$$robot = \{robot\}$$

PROPOSE also expects the domain modeller to supply a positive state invariant for each sort $s$ in the domain, called $IP_s$. The set of all the state invariants is collectively referred to as $\mathcal{IP}$. Each invariant is a set of sets of predicates representing a disjunction of conjunctions of predicates. So an invariant $IP_s$, for some sort $s$, represents the range of alternative "situations" that objects of this sort can occupy. Every valid state of the modelled planning domain must contain a legal substitution of exactly *one* member of $IP_s$, for every domain object of sort $s$.

For example, the set of state invariants for the robot planning world $SW$ are as follows:

$$\begin{aligned}
IP_{box} &= \{\{in_room(B,R), at_door(B,D,R)\}, \{in_room(B,R), next_to(B,B')\},\\
&\quad\ \{in_room(B,R)\}\}\\
IP_{robot} &= \{\{robot_in(R), robot_at(D,R)\}, \{robot_in(R), robot_near(B)\},\\
&\quad\ \{robot_in(R)\}\}\\
IP_{door} &= \{\{open(D)\}, \{closed(D)\}\}
\end{aligned}$$

where $B$ and $B'$ are variables of sort *box* and $R$ and $D$ are variables of sort *room* and *door* respectively. Since the invariants specify the alternative situations that objects of each sort can be in, we know for example, from invariant $IP_{box}$, that objects of sort *box* can either be in a room and at a door, or in a room and next to another box or simply in a room.

It is worth noting that the positive state invariants could be generated from the domain model if all sort information was explicit in predicate arguments. But, in the modelled planning domains that we are using as examples, some sort information is implicit in the predicate name. For example, the $SW$ predicate $robot\_in(Room)$ has sort information implicit in the predicate name. This could be re-represented, for example as $in(Robot, Room)$, where all the sort information is explicitly represented as predicate arguments. An advantage of this form of representation is that the number of objects of the sort *robot* could be increased without changing the representation. However, we decided not to re-represent the domain in this way because we wanted to avoid engineering the model of the planning domain to fit the requirements of PROPOSE.

Finally, we can define the set of *task configurations* for a modelled planning domain. We shall use $\mathcal{TC}$ to refer collectively to the set of task configurations for some domain and we shall use $TC_s$ to refer to a task configuration of some sort $s$. As we mentioned earlier the positive state invariants are used to generate the set of all possible start and end situations, that could occur in a planning problem, for an object of sort $s$. A task configuration for some sort $s$ is simply a pair $(i, g)$ where $i$ and $g$ are drawn from the state invariant $IP_s$ and represent some start and end situations for objects of sort $s$. For example, task configurations for the sort *robot* in the planning world $SW$ include:

$$(\{robot\_in(R), robot\_at(D, R)\}, \{robot\_in(X), robot\_at(Y, X)\})$$

where the start situation for the *robot* is in some *room* and at a *door* and the end situation also has the *robot* in some *room* at a *door*. For some input planning problem either the *room* or the *door* (or both) in the start and end situations of the task configuration will be bound to different objects, otherwise the end situation would be satisfied by the start situation !

Below is another example task configuration for some object of sort *robot* where the start situation has the *robot* in some *room* and near to some *box* and in the end situation it is in some other *room*.

$$(\{robot\_in(R), robot\_near(B), \{robot\_in(Z)\})$$

The set of task configurations for a sort $s$, denoted $TC_s$, is defined as the composite or product set of the alternatives specified in the invariant, as follows:

$$TC_s = IP_s \times IP_s$$

Intuitively $TC_s$ defines the set of all possible start and end situations for an object of sort $s$ in any possible planning problem.

## 5.3 PROPOSE

So far in this chapter we have discussed the motivation for PROPOSE and the nature of the positive state invariants that are used by PROPOSE during compilation. The purpose of this section is to precisely define the properties of the output of PROPOSE compilation. This consists of sets of iterative macros $\mathcal{M}$ which we define in section 5.3.1 below and sets of enumerated sequences $\mathcal{EM}$ whose definition is the subject of section 5.3.2 on page 71.

### 5.3.1 PROPOSE: Macro Operator Definition

The first part of PROPOSE compilation is the generation of the sets of iterative macros for each of the sorts in a modelled planning domain. The set of iterative macros for some sort $s$ is referred to as $M_s$ and we'll use $\mathcal{M}$ to refer to the set of all $M_s$ that are compiled for a domain. $M_s$ for some sort $s$, is given by:

$$M_s = \{((i, g), \overline{M}_s(i, g)) \mid i, g \in TC_s\}$$

So elements in $M_s$ are pairs of the form $((i, g), \overline{M}_s(i, g))$, where $(i, g)$ is an element of $TC_s$ and where $\overline{M}_s(i, g)$ is the set:

$$\overline{M}_s(i, g) = \{l \in (OS \cup OS^*)^* \mid pre_s(hd(l)) = sort_s(i) \wedge join_s(l, g) \wedge invar_s(l)\}$$

From the definition (which uses the set notation $X^*$ to denote the set of all sequences of elements from the set $X$) we can see that each iterative macro in the set $\overline{M}_s(i, g)$ is a sequence of elements, where each element is either an operator from the set $OS$ or a sequence of operators from $OS$. To illustrate the form of macros in $\overline{M}_s(i, g)$ we shall use an example from the warehouse planning world $WW$ (this is defined in appendix C on page 198), for the sort *crate* and the following task configuration:

$$(i, g) = (\{on\_floor(C, S), at(C, S)\}, \{on\_floor(C, S'), at(C, S')\})$$

There is one iterative macro in the set for this task configuration, which is the following sequence (the other macros for this planning domain can be found in appendix C.6 on page 204):

$$m = [load(C, S, S_1), [drive\_load(C, S_1, S_2)]^*, unload(C, S_2, S')]$$

where $load(C, S, S_1)$ is the first element in the macro, $unload(C, S_2, S')$ is the last element and the element $[drive\_load(C, S_1, S_2)]^*$ is a sequence or iterative part within the macro. For iterative macro parts we use the notation $[a, b, \ldots]^*$ to denote the fact that the sequence of operators within it can be repeated 0, 1 or more times.

From the definition of $\overline{M}_s(i, g)$ above, we know that macros in the set $\overline{M}_s(i, g)$ must satisfy three conditions. We shall now consider each of these conditions in turn using the example macro $m$ as an illustration. The first condition that the macros must satisfy is:

$$pre_s(hd(l)) = sort_s(i)$$

where $pre_s$ and $sort_s$ are functions that we define below and $hd$ is a sequence function that returns the head element from a sequence. The function $pre_s$ accepts as input an element of a macro and returns predicates of sort $s$, as follows:

$$pre_s(x) = if\ x \in OS\ then\ sort_s(x.pre)\ elseif\ x \in OS^*\ then\ sort_s(hd(x).pre)$$

We define $sort_s$ as a function that, given a set of predicates $P$, returns the predicate names of members of $P$ that are of sort $s$ (where $name(x)$ is the name of the predicate $x$):

$$sort_s(P) = \{name(x) \mid j \in IP_s \wedge x \in j \wedge x \in P\}$$

So this first condition on macros in $\overline{M}_s(i,g)$ requires that all predicates of sort $s$ in the pre-conditions of the first element of the macro are equal to the set of predicates in $i$ (recall that $i$ is the start situation for the current task configuration). In other words, the pre-conditions of the first operator in the macro must be achieved by $i$. We can see that this condition is true of the example macro $m$ since the pre-conditions of sort *crate* for $load(C,S,S_1)$ are $on\_floor(C,S)$ and $at(C,S)$ which is the same as $i$.

The second condition that macros in $\overline{M}_s(i,g)$ must satisfy is that the positive effects of sort $s$ for an operator are the same as the pre-conditions of sort $s$ for the next operator in the macro and where $g$ gives the final pre-condition. In other words operators in the macros must "join" and we define this condition as follows:

$$\begin{array}{l} join_s(l,g) = \\ \quad if\ len(l) = 1\ then\ post_s(hd(l)) = sort_s(g) \\ \quad else\ join_s(tl(l),g) \wedge post_s(hd(l)) = pre_s(hd(tl(l))) \end{array}$$

where the functions $pre_s$ and $hd$ are as defined above and $tl$ and $len$ are sequence functions that return the tail and the length of the sequence, respectively. The function $post_s$ accepts as input an element from a macro and returns domain predicates of sort $s$ as follows:

$$\begin{array}{l} post_s(x) = \\ \quad if\ x \in OS\ then\ sort_s(x.e^+)\ elseif\ x \in OS^*\ then\ sort_s(last(x).e^+) \end{array}$$

where *last* is a function that returns the last element in a sequence. We can see that the join condition is satisfied for the example macro $m$ that was given earlier. If we start with $unload(C,S_2,S')$, the last operator in the macro, its positive effects of sort *crate* are $on\_floor(C,S')$ and $at(C,S')$ and these are the same as $g$ from the task configuration. The next element of the macro $m$ that we consider is the last operator in the iterative part of $m$ which is $[drive\_load(C,S_1,S_2)]^*$. Its positive effects of sort *crate* are $truck\_loaded(C,S_2)$ and $at(C,S_2)$ which are the same as the pre-conditions of $unload(C,S_2,S')$, so they "join". Finally, the positive effects of sort *crate* for the first operator in the macro, $load(C,S,S_1)$ are $on\_floor(C,S)$ and $at(C,S)$ which are equal to the pre-conditions of the head operator from the iterative part of the macro.

The final condition that macros in $\overline{M}_s(i,g)$ must satisfy is that any sequences of operators within them, the "iterative part", must satisfy a *loop invariant*. We define this loop invariant in terms of two conditions. The first states that the positive effects of sort $s$ of the last operator in the iterative part of the macro must be the same as the predicates of sort $s$ in the first operators' pre-conditions. In other words each application of the iterative part achieves its own pre-conditions of sort $s$. The second condition of the loop invariant is that the iterative component must be *effective* in the sense that it must change the truth values of predicate instances of sort $s$. This condition ensures that the macro doesn't contain sequences of operators that achieve nothing ! We define the loop invariant as follows:

$$\begin{array}{l} invar_s(l) = \forall\, x \in elems(l) : x \in OS^* \Rightarrow \\ \quad sort_s(hd(x).pre) = sort_s(last(x).e^+) \wedge \exists\, p \in sort_s(hd(x).pre) : p \in x.e^- \end{array}$$

where the functions $sorts_s$, $hd$, $last$ are as defined earlier and the function $elems$ simply returns the set of elements in a sequence.

We can see that for the example macro given earlier, where the iterative part is $[drive\_load(C, S_1, S_2)]^*$, both of the conditions in the loop invariant are satisfied. The first condition is satisfied because for sort *crate* the pre-conditions of the iterative part are the same as the positive post-conditions (in both cases they are instances of the predicates named *truck_loaded* and *at*). The second condition of the loop invariant is satisfied because the iterative component is effective since it changes the truth value of instances of the predicate named *truck_loaded* which is of sort *crate*.

### 5.3.2 PROPOSE: Enumerated Sequence Definition

The second part of PROPOSE compilation is the generation of the sets of enumerated sequences that can be used when unwinding the iterative parts of macros. The set of enumerated sequences for some sort $s$ is referred to as $EM_s$ and we shall use $\mathcal{EM}$ to refer to the set of all $EM_s$ that are compiled for a modelled planning domain.

Each $EM_s$ contains sequences of values that could be used to "unwind" iterative parts of macros in each $M'_s$. For example, suppose that the following is an iterative part of a macro for some task configuration in the warehouse world $WW$:

$$[drive(X, Y)]^*$$

if an actual planning problem required moving the *truck* from say $s1$ to $s10$ (see appendix C on page 198 for more detail on the layout of the warehouse) then this iterative part could be "unwound" using the sequence of values $[s1, s2, s6, s9, s10]$ to form:

$$[drive(s1, s2), drive(s2, s6), drive(s6, s9), drive(s9, s10)]$$

Unlike the iterative macros in $M'_s$, which are compiled for all the sorts of a modelled planning domain, $EM_s$ are compiled for selected sorts that are "static" and whose values are changed by the iterative part of some macro. Before we define the properties of these $EM_s$ we shall start by considering these two conditions for enumeration.

We say that a sort $s$ is "static", if there are predicates that have arguments of sort $s$, whose truth value cannot be changed by the application of any operator in $OS$. Within the model of planning that we are using predicates whose truth value cannot be changed are part of the environmental information for the domain. In the FM system this is supplied in the set $ES$. So for example, in the warehouse planning world $WW$, the environmental information includes the following (this is given in full on page 202):

$$\{next(s1, s2), next(s2, s1), next(s9, s10), next(s10, s9), next(s6, s2), next(s2, s6), \\ next(s6, s10), next(s10, s6), \ldots\}$$

which are instances of the predicate *next* whose truth values cannot be changed by application of operators in $OS$. Since the arguments of *next* are of sort *space*, this sort is said to be static. This reflects the fact that for this modelled planning domain, the relation of each *space* is fixed, so $s1$ is always next to $s2$ and so on.

Later in this section when we define $EM_s$ the static sort information will be re-represented as a set of binary relations, called $R_s$, where there is a binary relation $(a, b) \in R_s$ for every pair of objects of sort $s$ that appear in a predicate instance in $ES$. So for example, the set of instances of *next* that are given above will be re-represented as follows:

$$R_{space} =$$
$$\{(s1, s2), (s2, s1), (s9, s10), (s10, s9), (s6, s2), (s2, s6), (s6, s10), (s10, s6), \ldots\}$$

The second condition on the sorts $s$ for which PROPOSE generates $EM_s$, is that there is some iterative macro part that changes the way that predicate arguments of sort $s$ are instantiated. To illustrate this consider the following iterative part of some macro, of sort *truck*, from the warehouse world $WW$:

$$x = [drive(S, S')]^*$$

Both the pre- and post-conditions of $x$ include the predicate of sort *truck* that is named *truck_at* which has one argument of sort *space*. The iterative macro part $x$ is assumed to satisfy the loop invariant and hence we know that it changes the truth value of instances of this predicate. The sort of the argument that it changes is *space* so the second condition for enumeration is satisfied. Since the sort *space* is also "static" the first condition for enumeration is also satisfied. So the set $\mathcal{EM}$ for this domain model will contain $EM_{space}$.

Now we can turn to the definition of the sets of enumerated sequences. Each set $EM_s$, for some sort $s$, that satisfies the enumeration conditions, is given by:

$$EM_s = \{((a, b), \overline{EM}_s(a, b)) \mid a, b \in s\}$$

So elements in $EM_s$ are pairs of the form $((a, b), \overline{EM}_s(a, b))$ where $a$ and $b$ are objects of the sort $s$ and where $\overline{EM}_s(a, b)$ is the set:

$$\overline{EM}_s(a, b) = \{e \in s^* \mid$$
$$hd(e) = a \wedge last(e) = b \wedge enum(e, R_s)\}$$

where each element in $\overline{EM}_s(a, b)$ is a sequence of objects of sort $s$ (the notation $X^*$ is used to represent the set of sequences of elements of the set $X$). For each sequence $e$ the head and last elements of $e$ (accessed using the functions *hd* and *last* respectively) must correspond to $a$ and $b$ respectively. Also, it must be the case that either the head and tail elements of $e$ are in the set $R_s$ or that the condition $enum(e, R_s)$ is satisfied, where $R_s$ is a binary relation over the set objects in sort $s$ as defined earlier. The condition *enum* is defined as follows:

$$enum(a, R_s) = len(a) > 1 \Rightarrow ((hd(a), hd(tl(a))) \in R_s \wedge enum(tl(a), R_s))$$

which tests that the pair formed from the first two elements in the sequence are in $R_s$ and then recursively tests that the tail of the sequence also satisfies *enum*.

Finally, we note that although we have defined $EM_s$ where $s$ is some sort which satisfies the enumeration conditions (that it is static and that some iterative macro part changes the instantiations of predicate arguments of sort $s$), it is possible that $s$ can be a set of sorts. This is the case for the set of sorts $\{door, room\}$ (for the robot planning world $SW$) which are both static and are the sorts of predicate instances that are changed by the following iterative macro part:

$$x = [gothrudoor(D, R'), gotodoor(D', R')]^*$$

This iterative macro part changes the truth value of predicate instances of *robot_at*. The pre-conditions of $x$ include $robot\_at(D, R)$ and the post-conditions include the predicate $robot\_at(D', R')$ where $D$ and $D'$ are variables that can be instantiated to different values of sort *door* and where $R$ and $R'$ can be instantiated to different values of sort *room*. Hence, the sorts that it changes are *door* and *room*.

The set of binary relations $R_{door,room}$ is obtained from the set of environmental information, $ES$, so that there is a relation in $R_{door,room}$ for any pair of arguments of sort *room* or *door* in predicates in $ES$. For example, for $connect(room1, room2, door12)$ there are the pairs $(room1, door12)$ and $(room2, door12)$. We will discuss this example further in section 5.4.2 when we outline an implementation that enumerates the sets of sequences $\mathcal{EM}$.

Examples of the sets $\mathcal{EM}$ for the planning worlds $SW$, $SW^2$ and $WW$ are listed in appendices A.7, B.7 and C.7.

## 5.4 PROPOSE: An implementation

We have implemented a system that compiles a set of macro operators and a set of enumerated sequences for a modelled planning domain. In this section we give an outline of this implementation which is encapsulated in a procedure referred to as *propose*.

The procedure takes as input the set of operators $OS$, environmental information $ES$ and a set of positive state invariants $\mathcal{IP}$ (as defined in section 5.2) for some modelled planning domain. It outputs sets of macro operators $\mathcal{M}$, as defined in section 5.3.1 and sets of enumerated sequences $\mathcal{EM}$, as defined in section 5.3.2. We define *propose* as follows:

**procedure** *propose* : $OS \times ES \times \mathcal{IP} \longrightarrow \mathcal{M} \times \mathcal{EM}$
1. $propose\_macro(OS, ES, \mathcal{IP}, \mathcal{M})$
2. $propose\_enum(OS, ES, \mathcal{IP}, \mathcal{M}, \mathcal{EM})$

$return(\mathcal{M}, \mathcal{EM})$

so we can see that the procedure *propose* consists of the procedure *propose_macro* which we describe in the section below and *propose_enum* which we describe in section 5.4.2.

### 5.4.1 PROPOSE: Generating Macros

The procedure *propose_macro* generates a set of macros $\mathcal{M}$ that satisfy the conditions specified in section 5.3.1. It takes as input a domain model consisting of $OS$, $ES$ and the user supplied positive state invariant $\mathcal{IP}$. Its output is the set of propose macros $\mathcal{M}$. We define it as follows:

$$\begin{array}{ll} \textbf{procedure } \mathit{propose\_macro} : & OS \times ES \times \mathcal{IP} \longrightarrow \mathcal{M} \\ \quad 1. & \mathit{construct\_graphs}(OS, ES, \mathcal{IP}, \mathcal{G}) \\ \quad 2. & \mathit{construct\_macros}(\mathcal{G}, \mathcal{M}) \\ \quad \mathit{return}(\mathcal{M}) & \end{array}$$

So there are two main steps to *propose_macro*, firstly a set of graphs $\mathcal{G}$ are constructed and then these are used to construct the set of macros $\mathcal{M}$. Lets now consider the first step of *propose_macro* which is the construction of a set of directed graphs[1], one for each sort in the modelled planning domain, with the procedure *construct_graphs*. We'll refer to the directed graph for each sort $s$ as $G_s$ and $\mathcal{G}$ will be used to refer collectively to the set of all $G_s$.

The nodes of each directed graph $G_s$ are the elements of $IP_s$ and arcs in $G_s$ are triples of the form $(o, a, b)$, where $o$ is an operator name that "labels" the arc, $a$ is the tail of the arc and $b$ the head. Arcs in $G_s$ satisfy the following condition:

$$(o, a, b) \in G_s \Rightarrow \exists\, o \in OS : pre_s(o) = a \wedge post_s(o) = b$$

where $pre_s$ and $post_s$ are as defined in the section 5.3.1. So arcs in $G_s$ can be seen as representing the way that operators change the "situation" of objects of sort $s$.

As an illustration we will consider the graph $G_{robot}$ from the robot planning world $SW$, where the positive state invariant is:

$$\begin{array}{l} IP_{robot} = \\ \{\{robot\_in(R), robot\_at(D,R)\}, \{robot\_in(R), robot\_near(B)\}, \{robot\_in(R)\}\} \end{array}$$

The graph $G_{robot}$ that is output by *construct_graphs* is depicted in figure 5.1 where the nodes are enclosed in boxes and the arcs are labelled with operator names enclosed in double edged boxes with rounded corners. At each node is a set of predicates and an abbreviation for these, either $a$, $b$ or $c$, enclosed in parentheses. Arrows on the arcs represent their direction, so for example the figure shows that there is an arc in $G_{robot}$ from $\{robot\_in(R)\}$ to $\{robot\_in(R), robot\_at(D,R)\}$ and this is labelled with the operator $gotodoor(D,R)$. Also, there is an arc with arrows at both ends between the node $\{robot\_in(R), robot\_at(D,R)\}$ and the label $gotodoor(D,R)$. This represents the fact that the head and the tail of the arc are the same node, or in other words the form of the arc is:

$$(gotodoor(D,R), \{robot\_in(R), robot\_at(D,R)\}, \{robot\_in(R), robot\_at(D,R)\})$$

Now we can outline the second step of the compilation of the set of macros $\mathcal{M}$ which is encapsulated in the procedure that we call *construct_macros*. This takes as input the set of directed graphs $\mathcal{G}$ that are output by *construct_graphs* and outputs the set of macros $\mathcal{M}$. The basic procedure for each $M_s \in \mathcal{M}$ is to do a depth first search of the directed graph for that sort $s$ and to record the "paths" between every pair of nodes

[1]These were defined during the discussion on the motivation for PRECEDE on page 43.

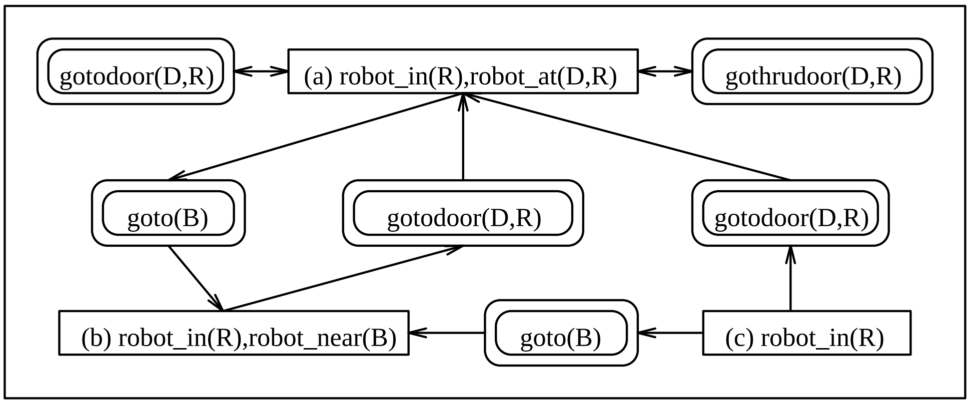


Figure 5.1: $G_{robot}$: sample output from *propose_graphs*

in the graph. Before looking at the depth first search in more detail we'll first consider the nature of the nodes in each graph and the paths or sequences of arcs that connect them.

The nodes in any $G_s$ are the elements of the state invariant $IP_s$, representing the alternate situations that objects of that sort can occupy. This means that the set of all pairs of nodes in $G_s$ is the set of task configurations $TC_s$ for that sort ($TC_s$ was defined on page 68 as the product set of $IP_s$). So for example, the three nodes in $G_{robot}$, shown as single square boxes in figure 5.1 are the elements of $IP_{robot}$. Hence for this sort, *construct_macros* will generate macros for each of the 9 pairs of nodes in the graph (a node can be paired with itself).

Arcs in the directed graph $G_s$ represent the way that operators change the situation of objects of sort $s$. So for example, when $gotodoor(D, R)$ is applied to a state where the situation of an object of sort *robot* is $\{in\_room(R)\}$ then the situation of that *robot* is changed to $\{in\_room(R), at\_door(D, R)\}$. In fact, the graph $G_s$ represents *all* the ways that operators can change the situation of objects of sort $s$. Hence macros in $M_s$ are simply the operators that label sequences of arcs between nodes in $G_s$.

So the set of macros for some task $(a, b)$ in $TC_s$ are simply the set of paths between this pair of nodes in $G_s$. These can be identified by doing a depth first search of $G_s$ which allows paths between pairs of nodes to be identified (see for example: [26]). However, the efficiency of these search algorithms assumes that nodes need only be visited once during the search. But we want to be able to identify "loops" in the graph to form the iterative macro parts and this can mean visiting nodes more than once.

The example graph $G_{robot}$ shown in figure 5.1 contains a number of loops which are paths from the node back to itself. Below we list a number of these loops and for simplicity we just list the sequences of operators that label the paths. These loops can be identified with a depth first search of the graph that only need visit each node in the graph once.

$[gothrudoor(D, R)]$

$[goto(B), gotodoor(D, R)]$

$[gothrudoor(D, R), gotodoor(D, R), goto(B), gotodoor(D, R)]$

$[gothrudoor(D, R), gotodoor(D', R)]$

$[gotodoor(D, R), gothrudoor(D, R')]$

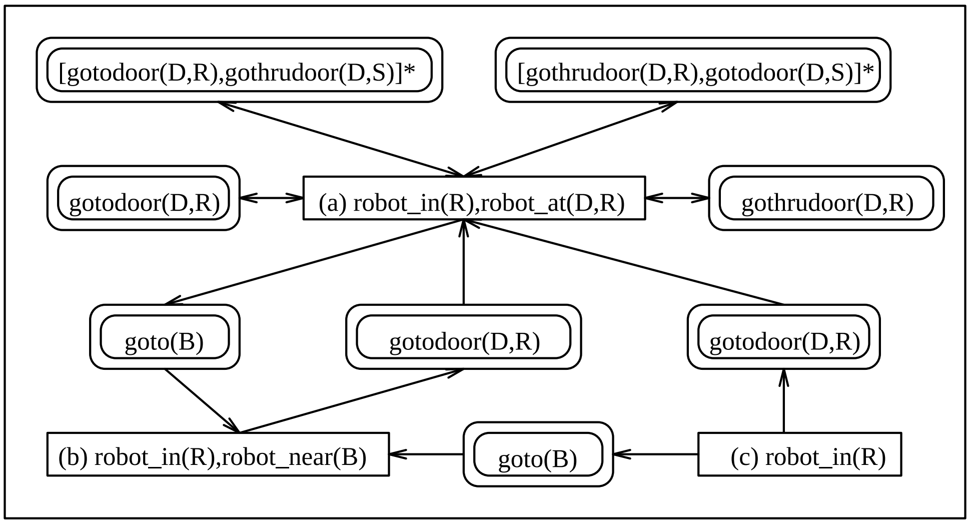


Figure 5.2: $G'_{robot}$ after loop updating

but the only loops that satisfy the invariant $invar_s(l)$ (see page 70) are as follows:

$$[gothrudoor(D, R), gotodoor(D', R)]^*$$
$$[gotodoor(D, R), gothrudoor(D, R')]^*$$

Our approach is to "process" the directed graphs $G_s$ in $\mathcal{G}$ and update them to form new graphs, called $G'_s$, where any loops from $G_s$ that satisfy the loop invariant have been added explicitly as arcs, labelled with the sequence of operators that formed the loop.

Figure 5.2 shows the updated graph $G'_{robot}$ after the loops have been processed. It contains the same nodes as $G_{robot}$ but the set of arcs has been expanded to include the two valid loops which were given above. This updated graph can now be searched and iterative macros generated using an algorithm that need only visit each node in the graph once.

The depth first search that is performed by *construct_macros* proceeds as follows. First each node in $G'_s$ is marked as "not visited", then a node that has not been visited is selected to start searching from. This node $a$ is the "current" node and it is marked as "visited" and will not be considered again during the search. The set of arcs that have the current node $a$ as the tail (for an arc $(o, a, b)$ $a$ is the tail and $b$ is the head) are then used to update the set $M_s$. Then the search "descends" by selecting as the current node tails of arcs in $a$ that are "not visited". The search terminates when all nodes in the graph have been visited.

Below, we illustrate the depth first search through the graph $G_{robot}$ but first we briefly outline the way that the graph $M_s$ is updated. For some current node $a$ and some arc $(o, a, b)$ the graph $M_s$ can be updated in the following ways:

1. For any $M_{robot}(X, a)$, new macros are added to $M_{robot}(X, b)$, where $X \neq b$[2], by *appending* $o$ to any macros in $M_{robot}(X, a)$.
2. For any $M_{robot}(b, Y)$ new macros are added to $M_{robot}(a, Y)$, where $Y \neq a$[2], by

[2]This ensures that no new "loops" are added to $M_s$, since all valid loops were "processed" earlier.

*prepending* $o$ to the sequences in $M_{robot}(b, Y)$.

3. New macros of the form $\{o\}$ are added to $M_{robot}(a, b)$

Now we shall illustrate the way that a "processed" (loop updated) graph $G_s$ is searched in the process of generating $M_s$. To simplify the example by avoiding long sequences of predicates to refer to nodes in the graph, we'll use the following abbreviations (which are shown in parentheses in the nodes of $G_{robot}$ and $G'_{robot}$):

$$a = \{robot\_in(R), robot\_at(D, R)\}$$
$$b = \{robot\_in(R), robot\_near(D, R)\}$$
$$c = \{robot\_in(R)\}$$

The first step of the procedure initialises the set $M_{robot}$ for each task configuration in $TC_{robot}$, as follows:

$$M_{robot} = \{((a, a), \{\}), ((a, b), \{\}), ((a, c), \{\}), ((b, a), \{\}), \ldots, ((c, c), \{\})\}$$

We'll suppose for this example that the search starts from node $b$ in $G'_{robot}$. This is marked as "visited" and the set of arcs that lead from $b$ to other nodes are assembled. From the graph (figure 5.2) we can see that there is only one arc from $b$, labelled with operator $gotodoor(D, R)$ for which the head is node $a$. The only way that the initialised $M_{robot}$ can be updated is by adding a new macro to $\overline{M}_{robot}(b, a)$ as follows:

$$M_{robot} = \{((a, a), \{\}), \ldots, ((b, a), \{[gotodoor(D, R)]\}), \ldots, ((c, c), \{\})\}$$

The search now "descends" and the next node to be considered is $a$. It is marked as "visited" and all arcs that lead from it are assembled. This set is as follows:

$$\begin{aligned}
A = \{ & \\
& (gothrudoor(D, R), a, a), \\
& (gotodoor(D, R), a, a), \\
& (goto(B), a, b), \\
& ([gothrudoor(D, R), gotodoor(D, R)]^*, a, a), \\
& ([gotodoor(D, R), gothrudoor(D, R)]^*, a, a)\}
\end{aligned}$$

We can now look at how $M_{robot}$ is updated using the arcs in $A$. The first way that it can be updated is by adding new macros to $\overline{M}_{robot}(X, a)$, for any macros in $\overline{M}_{robot}(X, a)$, but the only candidates are for $\overline{M}_{robot}(a, a)$ so no updates of this type are made, since all loops have already been processed. The second update condition is to add new macros to $\overline{M}_{robot}(a, Y)$, using any macros in $\overline{M}_{robot}(b, Y)$ where these new macros are formed by appending any $o$ from arcs $(o, x, y) \in A$. There is one candidate macro, which is

$$\overline{M}_{robot}(b, a) = \{[gotodoor(D, R)]\}$$

This is updated by adding new macros where the labels from the arcs in $A$ are appended to macros in $\overline{M}_{robot}(b, a)$. The updated $\overline{M}_{robot}(b, a)$ is as follows:

$$\overline{M}_{robot}(b, a) = \{$$
$$[gotodoor(D, R), gothrudoorD, R)],$$
$$[gotodoor(D, R), [gothrudoor(D, R), gotodoor(D, R)]^*], \ldots\}$$

Note that the macro below is not added to $\overline{M}_{robot}(b, a)$.

$$[gotodoor(D, R), [gotodoor(D, R), gothrudoor(D, R)]^*]$$

because it repeats the operator $gotodoor(D, R)$. Also excluded is the macro:

$$[gotodoor(D, R)]$$

This is redundant because it is contained in the macro below, since the iterative part of the macro may be repeated 0, 1 or more times.

$$[gotodoor(D, R), [gothrudoor(D, R), gotodoor(D, R)]^*]$$

The final way that $M_{robot}$ is updated is that a macro is added for each of the arcs in $A$. It contains arcs from $a$ to $a$ and these are used to update $M_{robot}(a, a)$ as follows:

$$\overline{M}_{robot}(a, a) = \{$$
$$[gothrudoor(D, R)],$$
$$[gotodoor(D, R)],$$
$$[[gothrudoor(D, R), gotodoor(D, R)]^*],$$
$$[[gotodoor(D, R), gothrudoor(D, R')]^*]\}$$

$A$ also contains arcs from $a$ to $b$ which are used to update $\overline{M}_{robot}(a, b)$ as follows:

$$\overline{M}_{robot}(a, b) = \{[goto(B)]\}$$

At this stage the only node left not visited is node $c$ so the search selects this as the current node and this generates macros including the following:

$$\overline{M}_{robot}(c, a) = \{[gotodoor(D, R)]\}$$
$$\overline{M}_{robot}(c, b) = \{[goto(B)]\}$$

The full set of macros for *robot* and the other sorts of the robot planning world are listed in appendix A.6 and the sets of macros for the other worlds $SW^2$ and $WW$ are given in appendices B.6 and C.6.

### 5.4.2 PROPOSE: Enumerating Sequences

As we saw in the definition on page 73 the procedure *propose* is specified in terms of *propose_macro* that generates the set of macros $\mathcal{M}$ which we outlined in the last section and *propose_enum* that enumerates the set $\mathcal{EM}$. In this section we outline the main steps in this procedure *propose_enum*.

It takes as input, the set of macros $\mathcal{M}$, for all the sorts in the modelled planning domain, which were output by *propose_macro* and $ES$, environmental information for the domain. The output is the set of enumerated sequences $\mathcal{EM}$.

**procedure** $propose\_enum : \mathcal{M} \times ES \longrightarrow \mathcal{EM}$

1. $enum\_sorts(\mathcal{M}, ES, S)$
2. $enum\_sequences(ES, S, \mathcal{EM})$

$return(\mathcal{EM})$

We can see that the first first step of the procedure is, *enum_sorts*, which identifies static sorts that are arguments of predicate instances which are changed by some iterative macro part. In section 5.3.1 we saw some examples of these, they include the sort *space* for $WW$ and the set of sorts $\{door, room\}$ for the robot planning world $SW$.

The main step of *propose_enum* is the enumeration of the sequences, $EM_s$, for each $s$ in the set output by *enum_sorts*. These are enumerated by searching, depth first, through the set of binary relations, $R_s$ for the sort $s$. To illustrate this search we shall use an example from the robot planning world $SW$ for the set of static sorts $\{door, room\}$ where the binary relation is as follows:

$$R_{door,room} = \{ (room1, door12), (door12, room1), (room2, door12), (door12, room2), (room2, door23), (door23, room2), (room3, door23), (door23, room3), \ldots, (room6, door67), (door67, room7), (room7, door67), (door67, room7)\}$$

The procedure *propose_sequences* which generates the sets of enumerated sequences $EM_s$ is simply a depth first search of the set of binary relations, in other words a graph, $R_s$. The search is essentially the same as that used in *construct_macros*, with slight differences in the way that the sets $EM_s$ are updated because the values that are added to sets in $EM_s$ are nodes in the graph $R_s$. Another difference is that pairs in $EM_s$ of the form $((a, a), s)$ are not permitted since this would be a sequence of values that could be used to unwind an iterative macro where its start and end values are the same. Finally, the enumerated sequences are the set of shortest paths through the binary relation $R_s$. For example, the pairs in $EM_{door,room}$ include the following, for start value $door67$ and end value $room3$:

$$\overline{EM}_{door,room}(door67, room3) = \{ [door67, room7, door57, room5, door53, room3], [door67, room6, door56, room5, door53, room3]\}$$

Examples of the sets of enumerated sequences for this planning world are listed in appendix A.7 and the sequences for the other planning worlds $SW^2$ and $WW$ are in appendices B.7 and C.7.

## 5.5 Complexity of PROPOSE compilation

In the previous section, we gave an outline of the implementation that we have developed which compiles $\mathcal{M}$ and $\mathcal{EM}$ for a modelled planning domain and is encapsulated in a procedure which we called *propose*. This was decomposed into *propose_macros* which outputs $\mathcal{M}$ and *propose_enum* which outputs $\mathcal{EM}$. In this section we consider the complexity of these parts of compilation with PROPOSE.

We have seen in the previous sections how the set of PROPOSE macros, $\mathcal{M}$, and enumerated sequences, $\mathcal{EM}$, are compiled direct from the domain model without the need for example planning problems. So the compilation method can be seen as off-line because it is independent of planning time, this is unlike methods, such as ALPINE [36], where time taken for compilation must be included in planning time. Nevertheless, the abstract complexity of the compilation method is important since if the searches required are exponential, with respect to some critical feature of the domain model, then this may limit the size of domain models that be compiled in "reasonable" time. In addition, the "size" of the compiled output, such as the number of macros for a particular task configuration, has a bearing on the utility of planning with the compiled domain model.

### 5.5.1 Complexity of Generating $\mathcal{M}$

The complexity of generating $\mathcal{M}$ depends on the procedures in *propose_macro*, which are *construct_graphs* and *construct_macros*. For the construction of the directed graphs $G_s$ in $\mathcal{G}$ we can see that this is linear with respect to both the number of elements in $IP_s$ for a sort $s$ and the number of operators in the set $OS$.

For the construction of the sets of $M_s$ in $\mathcal{M}$ the complexity depends on the loop processing and the construction of the macros. Identification of loops and construction of the sets of macros for some sort $s$ uses a depth first search whose complexity depends on the number of nodes in the graph $G_s$, we'll call this $N$, and the number of arcs in $G_s$, which we'll call $A$. The complexity of the depth first search (see for example [26]) which is used in *construct_macros* is:

$$O(max(N, A))$$

The size of $N$ for some sort $s$ is equivalent to the number of elements in the positive state invariant for that sort, $IP_s$. Also, there is an upper bound on the size of $A$ because in the worst possible case there will be an arc from every node to every other node or in other words, $|A| = N^2$. However, there will not be this many arcs in practice since operators within our planning model are *deterministic* in other words they are functions that map to a single state.

Finally, we need to consider the number of macros that are compiled by PROPOSE for some sort $s$, since this is important for the utility of using the compiled macros. The number of macros depends on the number of task configurations in $TC_s$, which in turn depends on the number of nodes $N$ in the graph. This is because tasks are pairs of the form $(x, y)$ where $x$ and $y$ can be any nodes in the graph (including $x = y$). The number of macros also depends on the average number of nodes *adjacent* to each node in a graph $G_s$, where $b$ is said to be adjacent to $a$ if there is an arc from $a$ to $b$ in the graph. This is similar to the average branching factor of the graph, since it is the average number of nodes that can be reached from any one node, we'll refer to it as $B$.

It turns out that the total number of macros for each $M_s$ is exponential with respect to $B$. Hence, the maximum number of macros that could be generated is:

$$O(N^2 \times 2^B)$$

However, this will be acceptable if the size of $B$ can be kept low. Factors that influence the size of $B$ include the number of states that operators map to and from and the number of situations that objects of different sorts can occupy. We have already seen that operators within the model of planning that we are using (see: section 2.2) are deterministic in that they map to at most one state. One consequence of this is that the number of arcs in any $G_s$ never reaches the maximum. Another factor that limits the size of $B$ in the example planning worlds $SW$, $SW^2$ and $WW$ is that the number of situations that objects of sorts can occupy (in other words the number of elements in $IP_s$) is quite small, on average $IP_s$ contain three elements.

### 5.5.2 Complexity of Generating $\mathcal{EM}$

To assess the complexity of enumerating the set $\mathcal{EM}$, three features must be considered. These are firstly, the number of static sorts that are changed by iterative part of macros, secondly, the complexity of the depth first search which is used to explore the graph of binary relations for each static sort (or set of sorts, such as $\{door, room\}$), and finally, the total number of enumerated sequences that are generated.

The number of static sorts that can be "changed" by an iterative part of a macro (as discussed in section 5.3.2) depends on the number of *sorts* for a modelled planning domain. In the worst case, the total number of sorts (and sets of sorts) that can be changed is the number of elements in $\mathbb{P}\ sorts$. However, this would mean that all sorts in the modelled planning domain were static and the truth value of predicates of that sort could not be changed. In the example planning worlds that we are using as illustration, the number of static sorts are low. For example, in the warehouse planning world $WW$, the sorts are:

$$\{crate, driver, truck, crane, space, shelf\}$$

but the only static sort that is "changed" by iterative macros is *space*. We contend that the complexity of the number of sorts is acceptable when the number of static sorts whose values are enumerated is low.

The complexity of enumerating depends on the complexity of the depth first search that is used to explore the binary relation $R_s$, which relates together objects of sort $s$. This search is similar to that used for *propose_macros*, which has already been considered in the previous section. We summarise the complexity of searching the graph, $R_s$, with nodes $N$ and arcs $A$ as:

$$O(max(N, A))$$

Finally, we must consider the complexity of the number of enumerations that could be generated. This depends on the number of nodes, $N$, and the average number of nodes adjacent to each node, in other words the average branching factor $B$ in the graph $R_s$. As with the complexity of the number of macros in $M_s$ (see section 5.5.2) this turns out to be exponential with respect to the average branching factor. So the maximum number of enumerations that could be generated is:

$O(N^2 \times 2^B)$

However, this describes the worst case situation when each node in $R_s$ is adjacent to every other node. We contend that if $B$ is relatively low, then the number of sequences that are enumerated will be acceptable and one way that the size of $B$ can be restricted is when nodes in $R_s$ are not adjacent to every other. It turns out that in the modelled planning domains that we have looked at the average branching factor of graphs $R_s$ is low. For example, consider the graph $R_{door,room}$, which was discussed at the end of section 5.4.2. The average branching factor for $R_{door,room}$ is 2 and this reflects the fact that on average each door is connected to two rooms and each room is connected to two doors.

## 5.6 Static Evaluation of PROPOSE Compilation

So far in this chapter we have defined the output of PROPOSE compilation and outlined an implementation that generates $\mathcal{M}$ and $\mathcal{EM}$. In this section our aim is to analyse the macros and enumerated sequences in $\mathcal{M}$ and $EM$ to assess how "useful" they appear. This will help in formulating the empirical tests of PROPOSE that are the subject of chapter 7. We begin this section with an example of how the use of PROPOSE compiled macros and enumerated sequences can be used during planning. Then we consider two aspects of the utility of using these compiled domain models. The first is how far the macros and enumerated sequences appear to be "correct" (section 5.6.2) and secondly, the relation between the size of a modelled planning domain and the output of PROPOSE (section 5.6.3).

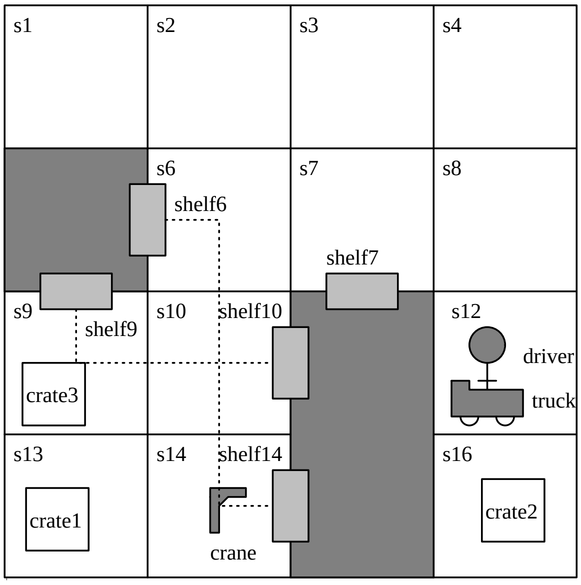


Figure 5.3: Example $WW$ initial state

### 5.6.1 Example: using $\mathcal{M}$ and $\mathcal{EM}$ during planning

In chapter 6 we will select a planner in the class DIP and show how its plan generation engine can be changed so that it can be bound with a PROPOSE compiled domain model and in chapter 7 we will present results of empirical tests with this planning application. However, in this section we preempt this slightly by presenting an example of how macros and enumerated sequences can be used during planning and show the difference in performance that can result from PROPOSE compilation.

The example planning problem is taken from the warehouse planning world $WW$, where the goal condition is given below and the initial state is as shown in figure 5.3.

$$\{truck\_loaded(crate2, s14), at(crate2, s14)\}$$

From the condition above we know that in the goal state, *crate*2 will be on the *truck* and at location $s14$ and from the initial state, *crate*2 is on the floor at location $s16$. Hence the task configuration that is appropriate for *crate*2, an object of sort *crate* is a pair as follows:

$$(\{on\_floor(C, S_i), at(C, S_i)\}, \{truck\_loaded(C, S_g), at(C, S_g)\})$$

since in the initial state *crate*2 is on the floor at location $s16$ and in the goal state it is loaded on the *truck* at location $s14$.

The set of macros that were compiled by PROPOSE for sort *crate* include the following macro for this task configuration:

$$[load(C, S_i, S_1), [drive\_load(C, S_1, S_g)]^*]$$

The iterative component of this macro is $[drive\_load(C, S_1, S_2)]^*$ and contains the name of a single operator that can be repeated 0, 1 or more times. It can be repeated because firstly, each application of it establishes its own pre-conditions (of sort *crate*) and secondly, it changes the truth values of predicate instances of sort *crate*, for example, before it is applied $at(C, S_1)$ is true and after it is applied $at(C, S_1)$ is false (for more on this see the loop invariant on page 70).

During planning, a domain independent planner can use this macro to help select operators to achieve goals during planning (this is the subject of section 6.2.2). Also, the enumerated sequences in $\mathcal{EM}$ can be used to "unwind" the iterative part of the macro $[drive\_load(C, S_1, S_2)]^*$. To do this the "start" and "finish" values for the unwinding must be identified. This depends on the sorts that are changed by the iterative component and the instantiations of predicate arguments of those sorts from the planning problem. The static sort that is changed by the iterative macro is *space*, since the pre-conditions include $truck\_loaded(C, S_1)$ and the post-conditions include $truck\_loaded(C, S_2)$, where $S_1$ and $S_2$ are variables that can be instantiated to *different* objects of sort *space* and where $C$ must be instantiated to the same object of sort *crate*. The "finish" value for the enumeration is $s14$ since this would achieve the goal of $truck\_loaded(crate2, s14)$. The "start" value of the enumeration is the *space* after *crate*2 has been loaded onto the truck, which is $s12$. So the enumerated sequence for sort *space* and values $(s12, s14)$ is retrieved from $\mathcal{EM}$. This sequence is $[s12, s8, s7, s6, s10, s14]$ and it is used to unwind the iterative part of the macro as follows:

$$drive\_load(crate2, s12, s8), drive\_load(crate2, s8, s7), drive\_load(crate2, s7, s6),$$
$$drive\_load(crate2, s6, s10), drive\_load(crate2, s10, s14)$$

The use of PROPOSE compiled macros and enumerated sequences in a manner similar to that sketched out above can improve the performance of a planning algorithm by a factor of two. For example, for the following problem:

$$G = \{crane\_at(s6), on\_floor(crate2, s13)\}$$
$$I = \{crane\_at(s14), clear(s9), clear(s7), clear(s13), clear(s1), clear(s6),$$
$$clear(s16), clear(s3), clear(s8), clear(s4), truck\_at(s2), driver\_at(s2),$$
$$at(crate3, s12), on\_floor(crate3, s12), at(crate2, s10), on\_floor(crate2, s10),$$
$$at(crate1, s14), on\_floor(crate1, s14), in\_truck, truck\_unloaded, crane\_unloaded\}$$

an example planning algorithm from the class DIP (described in section 6.1) specialises 293 partial plans and takes 67 CPU seconds to generate a plan for this problem. The same planning algorithm, which has been changed to accept a PROPOSE compiled domain model, requires only 28 CPU seconds and specialises only 145 partial plans to generate the same plan to achieve this planning goal. The results of pilot empirical tests such as this, suggest that PROPOSE compilation will be useful in improving the performance of planning systems in the class DIP.

### 5.6.2 The Correctness of PROPOSE compilation

In this section we consider how far the output of PROPOSE compilation, the iterative macros and enumerated sequences correspond to solution plans to planning problems.

Consider for example, the planning problem with the following goal:

$$G = \{on\_floor(crate1, s16), at(crate1, s16)\}$$

and initial state:

$$I = \{crane\_at(s14), clear(s12), clear(s9), clear(s7), clear(s13), clear(s10),$$
$$clear(s16), clear(s3), clear(s8), clear(s4), truck\_at(s6), driver\_at(s6),$$
$$at(crate3, s2), on\_floor(crate3, s2), at(crate2, s1), on\_floor(crate2, s1),$$
$$at(crate1, s14), on\_floor(crate1, s14), in\_truck, truck\_unloaded, crane\_unloaded\}$$

A plan that solves this problem is:

$$Plan = [drive(s6, s10), load(crate1, s14, s10), drive\_load(crate1, s10, s6),$$
$$drive\_load(crate1, s6, s7), drive\_load(crate1, s7, s8),$$
$$drive\_load(crate1, s8, s12), unload(crate1, s12, s16)]$$

The problem goal $G$ describes a goal state for $crate1$ an object of sort $crate$, namely that it be on the floor at location $s16$. The initial state of $crate1$ is on the floor at location $s14$ so the task configuration for this object of sort $crate$ is:

$$(\{on\_floor(C, S_i), at(C, S_i)\}, \{on\_floor(C, S_g), at(C, S_g)\})$$

A macro for this task configuration is as follows:

$$m = [load(C, S_i, S_1), [drive\_load(C, S_1, S_2)]^*, unload(C, S_2, S_g)]$$

and we can see that all of the operators in *Plan*, except for $drive(s6, s10)$, correspond to the operators in this macro. Suppose that we start with the "last" operator in $m$ ($unload(C, S_2, S_g)$) which corresponds to the last operator in *Plan*, under the substitution $\{crate1/C, s12/S_2, s16/S_g\}$. The element in $m$ that precedes this is the iterative part which corresponds to the sequence of *drive_load* operators in *Plan* from $s10$ to $s6$. Finally, the first element in $m$ corresponds to the operator $load(crate1, s14, s10)$ in *Plan*, under the substitution $\{crate1/C, s14/S_i, s10/S_1\}$.

The first operator in *Plan* is $drive(s10, s6)$ which does not appear in the macro $m$. This is because operations on the sort *truck* are not part of the task configuration for the goal $\{on\_floor(C, S_g), at(C, S_g)\}$ which is of sort *crate*.

The purpose of this brief example was to show that that PROPOSE macros and enumerated sequences reflect the structure of solution plans generated during planning. We contend that *all* macros generated for a task configuration for some sort are *correct* in that they contain a sufficient set of operators to achieve the goal of the task configuration for some object of that sort. This was illustrated in the example, where the operators in the PROPOSE macro $m$ were sufficient to move an object of sort *crate* (in this case *crate*1) from its initial situation to the goal situation but they did not contain any operators for changing the situation of objects of sort *truck* in isolation.

The contention that PROPOSE macros are correct is based on the way that they are generated from the set of directed graphs, $\mathcal{G}$, that were output by *construct_graphs* (see page 74). For some sort $s$, the nodes of a directed graph $G_s$ are the elements of the positive state invariant $IP_s$ for that sort. Each pair of nodes in $G_s$ represent a possible start and end situation that objects of that sort can occupy. Directed arcs in the graph are labelled with operators whose pre-conditions of sort $s$, are at the tail of the arc and whose post-conditions, of sort $s$ are at the head of the arc. In other words arcs in $G_s$ show how operators change the situation of objects of sort $s$. Hence, a sequence of operators that label a path through some $G_s$, from node $a$ to node $b$ show one way in which the situation of some object of sort $s$ can be changed from situation $a$ to situation $b$. The set of paths through the graph show all the ways that the situations of objects of sort $s$ can be changed.

### 5.6.3 Relation between domain model size and PROPOSE output

Finally, in this section we consider the relation between the size of a modelled planning domain and the output of PROPOSE compilation, since this is important when assessing the utility of the compiled domain model. PROPOSE has been used to compile the planning worlds $SW$, $SW^2$ and $WW$ and the sets of macros and some of the enumerated sequences for these domains are listed in the appendices on pages 146, 175 and 204

Table 5.6.3 contains information on the relative size and structure of the modelled planning domains $SW$, $SW^2$ and $WW$, for comparative purposes. Column **ops** gives the number of operator instances, **preds** the number of predicate instances, and **states** the approximate number of world states.

| | ops | preds | states | $\mathcal{TC}$ | $\mathcal{M}$ | $\mathcal{EM}$ | sequences |
|---|---|---|---|---|---|---|---|
| SW | 207 | 208 | $7 \times 10^6$ | 18 | 26 | 272 | 300 |
| SW$^2$ | 414 | 240 | $10^9$ | 18 | 26 | 272 | 300 |
| WW | 482 | 290 | $2 \times 10^6$ | 28 | 45 | 156 | 264 |

Table 5.1: Planning Domains: size information

The columns labelled $\mathcal{TC}$ and $\mathcal{M}$ contain information about the number of macros that are output by PROPOSE. Column $\mathcal{TC}$, shows the number of task configurations for a modelled planning domain, so for example, the number of task configurations totaled over all the sorts of the warehouse world is 28. The column $\mathcal{M}$ shows the number of macros totaled across all the sorts and all the task configurations. We can see from this that for any task configuration that could appear as a part of a planning problem in $WW$, there are an average of two iterative macros that have been output by PROPOSE.

Finally, the right hand columns contain information about the number of enumerated sequences that are output by PROPOSE. The column $\mathcal{EM}$ gives the number of $EM_s$ in $\mathcal{EM}$ totaled across all the static sorts which are enumerated. For the warehouse world there are 156 such elements. The column **sequences** is the actual number of sequences for all of the elements of $\mathcal{EM}$ which for $WW$ is 246. Hence, for the warehouse world there are on average 2 sequences that can be used to unwind any iterative part of a macro.

The relation between the size of domain model and the size of the output that is compiled by PROPOSE suggests that it will be useful. This supports the conjecture that the size of $\mathcal{M}$ and $\mathcal{EM}$ will be acceptable, at least for the planning domains that we are using as illustration.

## 5.7 Summary

PROPOSE is a new compilation method that compiles sets of iterative macro operators and enumerated sequences that can be used to unwind the iterative parts of the macros. It uses information about valid states of a modelled planning domain, supplied by the domain modeller in the form of positive state invariants on objects of different sorts in the domain. In this chapter we defined the properties of the sets $\mathcal{M}$ and $\mathcal{EM}$ and outlined an implementation encapsulated in a procedure called *propose* which outputs them.

PROPOSE has been used to compile the example planning domains $SW$, $SW^2$ and $WW$. The results of static evaluation of PROPOSE macros and enumerated sequences for these domains lead us to expect that they will be useful in improving planner performance. Therefore we would expect to see improvement in planner performance as a result of planning with a PROPOSE compiled domain model, in the empirical tests that are the subject of chapter 7

# Chapter 6

# Planning with PRECEDE and PROPOSE

Development of PRECEDE and PROPOSE was motivated by the need for off-line methods of compiling domain models in ways that are useful for choosing between different partial plan specialisations. These choices concern which outstanding predicate goals to achieve next, which operator to use to achieve each goal and how to bind variables in the selected operators. As we saw in chapter 4 PRECEDE compiles a partial order over predicates in a modelled planning domain which is based on necessary interactions between the predicates. PROPOSE, the subject of chapter 5, compiles iterative macro operators and sequences of values that can be used to unwind them.

The purpose of this chapter is to show how PRECEDE and PROPOSE compiled domain models can be used during planning. We have selected an example planning system, called *MEA* and we shall show how its plan generation algorithm has been changed to accept and use PRECEDE and PROPOSE compiled domain models. This implementation will also enable us to test the effect of the compilation methods on the performance of *MEA*, which is the subject of chapter 7.

We begin the chapter by looking in some detailed at the *MEA* algorithm (section 6.1) so that the effect of PRECEDE and PROPOSE on the algorithm can be appreciated. Then in section 6.2 we look at how the algorithm is changed to reason about PRECEDE and PROPOSE compiled domain models. For comparison, we also consider the effect of two comparable methods, ABGEN (our implementation of the ALPINE algorithm [49]) and CHAR [50]. In section 6.2.5 we consider how the different compilation methods can best be combined in a compilation function. Finally, in section 6.3 we consider the amount of search reduction that might result from the domain model compilation.

## 6.1 An example planning system: *MEA*

PRECEDE and PROPOSE are general compilation methods that can be applied to any member of the class of planners that we are interested in (the class of DIP's was defined on page 7). We will consider the generality of the compilation methods further in section 9.1 but for the remainder of this chapter (and for the empirical tests in chapter 7) we will focus on one particular DIP so that we can evaluate the effect of PRECEDE and PROPOSE on its planning performance.

The example DIP that we have selected is called *MEA*. It is part of the *FM* system, an integrated architecture for studying planning and learning, that we introduced in chapter 2 and which we have been using throughout the thesis as an illustration. *MEA* is a goal directed planner, that uses a variation of Means End Analysis (first introduced in [58]) to guide search. It assumes that it can build up plans that solve conjunctive goal problems from linear plans for the individual goals in the conjunction. This is often referred to as the linearity assumption and is a feature of many planners in the tradition of STRIPS [22]. As we saw in section 2.3.2.1 one consequence of assuming linearity is that it can lead to considerable increases in efficiency although this is at the cost of completeness. So *MEA* loses completeness for gains in efficiency but it is correct in the sense that all solution plans that it generates do solve the input planning problem. The relative efficiency of plan generation was one of the reasons why we selected *MEA*, since if PRECEDE and PROPOSE compilation could improve the performance of an already quite efficient planning system then it seemed reasonable to assume that it might also improve the performance of less efficient DIP's.

The main purpose of this section is to outline the *MEA* plan generation algorithm. It is a post-hoc formalisation of the algorithm that was first described in [46] so the terminology has been slightly changed although the system is still essentially the same.

Below we give an outline of the *MEA* plan generation algorithm. It takes as input a planning problem of the form $(I, G, OS, ES)$ where $I$ and $G$ are the initial state and the goal condition for the problem, $OS$ is the set of domain operators and $ES$ the environmental information. Termination occurs when a partial plan which solves the problem is found or planning fails because there are no more ways to specialise the partial plans, or planning is abandoned when some resource limit is reached. The function returns on success, with a solution plan $S$ else it returns either *failed* or *abandoned*.

**procedure** $MEA : (I, G, OS, ES) \longrightarrow S$

1. $initialise(I, G, PP)$
2. $until\ solution(S)\ or\ fail\ or\ abandon\ do:$
   - $get\_next(PP, pp)$
   - $specialise\_plan(pp, pp')$
   - $queue\_plans(PP, pp')$

$return(S)$

During plan generation *MEA* reasons about partial plans, which were defined in section 2.2.4 as 4-tuples of the form: $pp(Os, Ts, Ps, As)$, where $Os$ is a set of *totally instantiated* operators from $OS$ plus the special operators *goal* and *current*, $Ts$ is a *total* order on operators in $Os$, $Ps$ is the set of *unachieved* pre-conditions of operators in $Os$ and $As$ is the set of *achieved* pre-conditions of operators $Os$ (for more details of partial plan representation see section 2.2.4 on page 17). Two important features of partial plan representation in *MEA* are firstly, that operators in a partial plan are always fully instantiated and secondly, that operators are always totally ordered with respect to each other.

The first step of $MEA$ is to initialise the partial plan queue, $PP$. This takes $I$ and $G$ from the specification of the problem and outputs $PP$ containing the partial plan $pp_i$. The form of this initial partial plan was explained in section 2.2.4.

The second step is the main step of $MEA$ and it is repeated until one of the termination conditions is met. It terminates on success when the priority queue of partial plans $PP$ contains a partial plan $pp$ that has no pre-conditions left to be achieved. This means that all goals have been achieved and the solution plan to the current planning problem is the sequence of operators in the special operator *current* (we will discuss this further later in this section). Termination also occurs when there are no more ways to specialise the set of partially developed plans or planning is abandoned when some resource limit is reached.

During planning, the set of partial plans is maintained by $MEA$ as a priority queue $PP$ where the partial plan at the front of the queue will be specialised next (it is retrieved using the procedure $get\_next(PP, pp)$). The general heuristic that $MEA$ uses to assign priority to a partial plan is based on the number of outstanding goals it contains, in other words the number of pre-conditions in $Ps$. The plans are ordered so that the partial plan with the *fewest* unachieved pre-condition goals is at the front of the priority queue $PP$ and at the end of the queue are the partial plans with the most outstanding pre-conditions.

Now we can look at the specialisation of partial plans which is the main step of $MEA$. The function *specialise_plan* is defined below. It takes as input a partial plan $pp$ and it returns a set of partial plans $pps$ that are specialisations of $pp$.

**procedure** $specialise\_plan : pp \longrightarrow pps$

1. $pps = \{\}$
2. $for\ all\ g \in earliest\_goal(pp)\ do:$
3. $\quad for\ all\ o \in i\_achievers(g)\ do:$

$\qquad pp' = add(pp, o)$

$\qquad if\ earliest\_goals(pp') = \{\}\ then:\ pp'' = advance(pp')$

$\qquad pps = append(pps, pp'')$

$return(pps)$

The first step of *specialise_plan* is to initialise the set of partial plans in $pps$ to $\{\}$. The next step of the procedure is to get the "earliest goals" of the partial plan $pp$ which is being specialised. The earliest goals of a partial plan are the set of outstanding pre-conditions of the "earliest operator" in the partial plan. The earliest operator in the partial plan is the operator that immediately follows the special operator *current*. Given that the order $Ts$ is a *total* order over the partial plan operators in $Os$, and that *current* is always the lower bound of this order, then the earliest operator is the earliest operator from the set of domain operators $OS$. For example, in the partial plan below the earliest operator is $o_1$ and the earliest goals are $\{a\}$.

$$pp(\{current, goal, o_2, o_1\}, \{(current, o_1), (o_1, o_2), (o_2, goal), \ldots, (current, goal)\},$$
$$\{(a, o_1), (b, o_2), \ldots\}, \{\ldots\})$$

Once the earliest goals have been identified, the next step of *specialise_plan* gets the set of $i\_achievers(e)$ for each $e$ in the earliest goals. Then it creates a new partial plan for each instantiated achiever operator. To illustrate the way that these new partial plans are created we'll use an example. Suppose that the partial plan *pp*, given above is being specialised. In this partial plan there is only one earliest goal, which is the goal $a$. If the $i\_achievers(a)$ is the set: $\{o_3, o_4\}$ then two new partial plans formed by "adding" these operators (using the procedure *add*) to *pp*, as follows:

$$pp_x(\{current, goal, o_2, o_1, o_3\}, \{(current, o_3), (o_3, o_1), (o_1, o_2), \ldots)\},$$
$$\{(x, o_3), (b, o_2), \ldots\}, \{(a, o_3), \ldots\})$$
$$pp_y(\{current, goal, o_2, o_1, o_4\}, \{(current, o_4), (o_4, o_1), (o_1, o_2), \ldots, \},$$
$$\{(y, o_4), (b, o_2), \ldots\}, \{(a, o_4), \ldots\})$$

where $pp_x$ is the partial plan formed by "adding" $o_3$ to *pp* and $pp_y$ is the partial plan formed from adding $o_4$ to *pp*. When a new partial plan is formed by adding an operator then all the four elements of the partial plan $pp(Os, Ts, Ps, As)$ are changed. *Os* is updated by appending the operator, in this case $o_3$ or $o_4$ to the set. The temporal order *Ts* is updated and the new operator becomes the earliest operator in the plan. For example, in $pp_x$ operator $o_3$ has been "slotted" in to the temporal order immediately after *current* and before $o_1$. Any outstanding pre-conditions of the new operator are added to *Ps*, for $pp_x$ operator $x$ is an outstanding pre-condition of $o_3$ and so has been added to *Ps* (a pre-condition is "outstanding" if there is no operator in the partial plan that achieves it and this will be reflected by an entry in *As*). Finally, *As* in the new partial plan reflect the goals that have been achieved and so the *As* for $pp_x$ contains $(a, o_3)$ which denotes that pre-condition $a$ is achieved in this partial plan by operator $o_3$.

If it happens that a new partial plan is generated which has no outstanding earliest goals, then the current state of this partial plan is advanced. This is an important aspect of *MEA* plan generation, because the assumption is that there is a linear order in which goals can be achieved so once an operator that has no outstanding pre-conditions is found it can be applied and the current state of the world advanced. The assumption is that all remaining goals can be achieved from this advanced state. As we saw in 2.3.2.1 this assumption is not always the case for some problems and the goals cannot be achieved in a linear order. In the rest of this section we will sketch out the way that the state is advanced and how the *current* operator is updated.

Initially, a partial plan has just the two special operators *current* and *goal*. In this initial partial plan, the post-conditions of *current* are the predicates in the initial state description for the current problem and these are all reflected in *As* (as we saw in section 2.2.4). When the state of the partial plan is advanced, the earliest operator in the partial plan, is *applied* to the current state. The resulting state is the new current state, and this is used to update the set of achieved pre-conditions of the partial plan, in *As*. Also the operator that is applied is removed from the set in *Os* and is appended to the sequence of operators in the *current* operator. So the purpose of the *current* operator is to record the sequence of operators that have been applied to the initial state to achieve the current state.

Lets consider as an example, the situation in the following partial plan for the warehouse planning world *WW*:

$$pp_n(\{current, goal, unload(crate1, s7, s3), drive\_load(crate1, s8, s7),$$
$$load(crate1, s7, s8)\},$$

$$\{(current, load(crate1, s7, s8)), (load(crate1, s7, s8),$$
$$drive\_load(crate1, s8, s7)), \ldots\},$$
$$\{(truck\_loaded(crate1, s8), drive\_load(crate1, s8, s7)), \ldots\},$$
$$\{\ldots\})$$

where the earliest operator is $load(crate1, s7, s8)$ and because it has no outstanding pre-conditions it is applied to advance the state of $pp_n$. The first change is to update the set of operators in $Os$ to reflect the fact that the earliest operator has been applied. This is done by removing $load(crate1, s7, s8)$ from $Os$. So, the new $Os$ after $load(crate1, s7, s8)$ is applied is:

$$pp_n.Os = \{current, goal, unload(crate1, s7, s3), drive\_load(crate1, s8, s7)\}$$

Also the $current$ operator is updated to include $load(crate1, s7, s8)$. This operator was defined (see page 12) as a 3-tuple of the form $(current, oseq, e^+)$ where $current$ is the name of the operator, $oseq$ is the sequence of operators that have been applied so far to advance the state and $e^+$ is the current state of the planning world. We will suppose that the state of the partial plan $pp_n$ has not been advanced and hence $oseq$ is $\{\}$ and $e^+$ contains the set of predicates that describe the initial state of the modelled planning domain. After applying $load(crate1, s7, s8)$ to advance the state, $current$ becomes:

$$(current, \{load(crate1, s7, s8)\},$$
$$\{truck\_loaded(crate1, s8), truck\_at(s8), clear(s7), at(crate1, s8), \ldots\})$$

The temporal order is now changed by removing $load(crate1, s7, s8)$ which means that the earliest operator in $Ts$ becomes $drive\_load(crate1, s8, s7)$ as follows:

$$pp_n.Ts = \{(current, drive\_load(crate1, s8, s7)), (drive\_load(crate1, s8, s7),$$
$$\ldots), \ldots\}$$

The set of achieved pre-conditions, $As$, is updated to include all predicates that are achieved by applying $load(crate1, s7, s8)$, so $As$ becomes:

$$pp_n.As = \{(truck\_loaded(crate1, s8), current), (clear(s7), current), \ldots\}$$

Finally, since the intersection of the sets $As$ and $Ps$ must be disjoint, it may be necessary to change the set of unachieved pre-conditions, $Ps$, to reflect changes in $As$. Any pre-conditions that are achieved as a result of applying $load(crate1, s7, s8)$ are removed from $Ps$ and any pre-conditions that are clobbered must be added to $Ps$.

$$pp_n.Ps = \{(truck\_loaded(crate1, s7), unload(crate1, s7, s3)),$$
$$(truck\_at(s6), goal)\}$$

It can happen that the new "advanced" partial plan has no earliest goals and if this occurs then the partial plan is advanced again. In fact the partial plan is advanced until the new partial plan contains an earliest operator with outstanding pre-conditions, in other words until there are goals to be achieved or until the partial plan contains only the special operators $current$ and $goal$. In this situation the operator $current$ contains a sequence of operators that achieve all of the problem goals. So the solution plan that is described by an *MEA* partial plan is the sequence of operators contained in *current*.

This is different to some planners in the class of DIP, such as TWEAK [9] and NLP [45], where the operators in *Ts* are partially ordered and a solution plan is any linearisation of the operators in *Ts* that is consistent with the partial order. Also, these DIP's tend not to advance the state of partial plans during planning.

#### Summary

*MEA* is a relatively typical example of planners in the class of DIP systems. We have selected it as the planning system that we will use for the evaluation of PRECEDE and PROPOSE compilation on the basis of the following reasons:

- the performance of *MEA* is already quite good since it includes heuristics, such as an assumption of linearity, that help to reduce search during planning. We assume that if PRECEDE and PROPOSE compilation can improve the performance of an already quite efficient planning system then it might also improve the performance of less efficient DIP's.
- the good performance of *MEA* means that it can be used to solve relatively hard problems
- the performance of *MEA* has already been tested with other learning and compilation methods such as CHAR [50]

Plan generation in *MEA* can be viewed as a search through a space of partially developed plans, where partial plans describe a totally ordered set of fully instantiated operators from *OS* and the special operators *current* and *goal*. During planning *MEA* advances the current state of the modelled planning domain as planning goals are achieved. A problem is solved by *MEA* when the current state has been "advanced" to one which achieves all the goals in the conjunctive planning goal. In this situation the solution plan is the sequence of operators that have been applied to advance the current state. This sequence of operators are stored in the special *current* operator.

## 6.2 Compiled Planning Applications

Within the model of compilation that was described in section 3.2, a domain model, compiled or not, is bound with a plan generation engine to form a planning application for run-time execution. In this section we look at the planning applications that are formed by binding *MEA* with domain models that have been compiled with PRECEDE and PROPOSE and how the action of *MEA* is changed to use the compiled domain models. For comparison we also consider two related methods, ALPINE [36] and CHAR [46].

### 6.2.1 Planning with *MEA* and PRECEDE

We refer to the planning application formed by binding *MEA* with a PRECEDE compiled domain model as $MEA_3$ (the derivation of this name is explained further in section 6.2.5). The top-level planning algorithm for $MEA_3$ is virtually the same as the algorithm for $MEA$ given on page 88. The only difference is that $MEA_3$ has an extra input parameter, the global partial order $P$, compiled from the domain model. The main difference between the two planning applications lies in the way that partial plans are specialised.

### Plan Specialisation

$MEA_3$ specialises partial plans by achieving the earliest goals in an order corresponding to the PRECEDE goal order. Where $MEA$ specialised a partial plan by adding operators to achieve all of the outstanding earliest goals, $MEA_3$ achieves the outstanding goals that come earliest in the PRECEDE goal order. For example, if the partial plan to be specialised is as follows:

$$pp_i(\{\ldots, o_i, o_j\}, \{(current, o_i), (o_i, o_j), \ldots\}, \{(a, o_i), (d, o_i), (g, o_i), \ldots\}, \{\ldots\})$$

then its earliest goals are $\{a, d, g\}$ (the unachieved pre-conditions of the earliest operator in the partial plan). If we now suppose that the goal order compiled by PRECEDE for this domain is:

$$P = \{(a, b), (b, c), (a, c), (a, d), (a, g), (h, i), \ldots\}\}$$

this means for example that goal $a$ can be achieved before $b$, $c$, $d$ and $g$. The strategy of $MEA_3$ is to achieve goals in an order corresponding to the order in $P$. For partial plan $pp_i$ this means achieve goal $a$ for $o_i$ before $g$ and $d$. If we suppose that there is only one achiever operator for $a$, let it be $o_k$, then the specialised partial plan that is output is:

$$\begin{aligned} pp_j(&\{\ldots, o_i, o_j, o_k\}, \{(current, o_k), (o_k, o_i), (o_i, o_j), \ldots\}, \\ &\{(x, o_k), (d, o_i), (g, o_i), \ldots\}, \{(a, o_k), \ldots\}) \end{aligned}$$

There is only one earliest goal in $pp_j$ which is $x$ for operator $o_k$. So when this partial plan is specialised this goal will be achieved first. The goals $\{d, g\}$ from $pp_i$ can be achieved in one of two ways, either they are achieved as a positive side-effect of achieving some other goals (they are achieved by applying other operators to advance the state) or they are achieved once all the "descendent" goals from $o_k$ have been achieved and they have become the earliest goals in the partial plan.

## 6.2.2 Planning with *MEA* and PROPOSE

Now we shall look at the way that $MEA$ has been changed to accept and use a domain model that has been compiled by PROPOSE. We refer to the planning application that results from binding $MEA$ with a PROPOSE compiled domain model as $MEA_4$. $MEA_4$ accepts a domain model that includes sets of macro operators in $\mathcal{M}$ and sets of enumerated sequences in $\mathcal{EM}$ (these sets were defined in sections 5.3.1 and 5.4.2.

The top-level algorithm of $MEA_4$ is almost the same as $MEA$ (defined on page 88) although $MEA_4$ accepts extra input parameters, $\mathcal{M}$ and $\mathcal{EM}$. The main difference between the algorithms lies in the procedure *specialise_plan* (defined on page 89) where operators are selected for goal achievement and then instantiated. $MEA_4$ uses the sets of macros in $\mathcal{M}$ to select achiever operators and it uses the sets of enumerated sequences in $\mathcal{EM}$ to instantiate operators within iterative macro components.

During planning $MEA_4$ maintains a set of iterative macros for all the task configurations that are "part" of an input planning problem. For example, if an input planning problem for the warehouse world $WW$ contains the following goal condition $G$ and initial state $I$:

$$G = \{on\_floor(crate1, s16), at(crate1, s16), \ldots\}$$
$$I = \{truck\_loaded(crate1, s14), at(crate1, s14), \ldots\}$$

then the task configuration for the part of the problem that is related to *crate*1 (an object of sort *crate*) is:

$$(\{truck\_loaded(C, S), at(C, S)\}, \{on\_floor(C, S'), at(C, S')\})$$

The set of macros for this task configuration is given below (the sets of macros for this modelled planning domain are listed in full in appendix C.6).

$$\overline{M}_{crate}(\{truck\_loaded(C, S), at(C, S)\}, \{on\_floor(C, S'), at(C, S')\}) = \{ [[drive\_load(C, S, S')]^*, unload(C, S', S'')]\}$$

This tells us that the last operator in any plan for this task configuration will be an *unload* operator. Also plans for this task configuration may consist of just the *unload* operator and they are preceded by some sequence of 0, 1 or more *drive_load* operators (from the iterative component $[drive\_load(C, S, S')]^*$).

Lets now consider the way that $MEA_4$ selects operators and bindings to achieve a goal. To illustrate this we shall use an example where the following partial plan is to be specialised:

$$pp_x(\{current, goal, d, a, b, c\}, \{(current, a), (a, b), (b, c), (c, d), (d, goal), \ldots\}, \{(z, a), \ldots\}, \{(x, current), (y, d), \ldots\})$$

In $pp_x$ the temporal order $Ts$ (the second element in $pp_x$) is a total order over the operators in the partial plan. If we exclude the special operators *current* and *goal* this sequence is as follows:

$$[a, b, c, d]$$

The earliest goal of partial plan $pp_x$ is $z$ (the only pre-condition of the earliest operator in the plan, $a$). To be able to show how $MEA_4$ selects an operator to achieve this goal we need to identify the relevant macros and task configuration for the partial plan $pp_x$. We know that the goal context for this sequence of operators is $y$ since operator $d$ (the last in the sequence) achieves $y$ which is a pre-condition of the *goal* operator (from the pair $(y, d)$ in the fourth argument of $pp_x$). We'll suppose that this predicate is of sort $s$ and that the initial situation for this object is $x$. Hence, the context of this sequence of operators is the task configuration $(x, y)$. Lets now suppose that the macros for this task configuration are as follows:

$$\overline{M}_s(x, y) = \{[o_1, [o_2]^*, a, b, c, d], [o_1, d]\}$$

Now we can return to the way that $MEA_4$ uses these macros to achieve the goal selected from $pp_x$, in this case the earliest goal $z$. Achievers for $x$ are selected from macros in the set $M_s(x, y)$ where the sequence of operators in the partial plans' temporal order (excluding *goal* and *current*) match some "tail" of the macro. For example, for the task configuration $(x, y)$ the macros are as follows (numbered $m_1$ and $m_2$ for ease of reference):

$$m_1 = [o_1, [o_2]^*, a, b, c, d]$$
$$m_2 = [o_1, d]$$

and the sequence of operators in $pp_x$ are $[a, b, c, d]$. We can see that this sequence matches the "tail" of macro $m_1$ that results when the elements $o_1$ and $[o_2]^*$ have been removed. There is no match with the other macro $m_2$. So the element that is selected from $m_1$ is the element that immediately precedes $[a, b, c, d]$, which in this case is $[o_2]^*$ and this operator is used as the achiever for goal $x$. It should be noted that the matching process can "handle" repeated sequences of actions in a partial plan. For example if the sequence of operators was:

$$[o_2, o_2, \ldots, o_2, a, b, c, d]$$

Then this would still match the tail of macro $m_1$. In this case the "next" element to be selected (either $o_1$ or $[o_2]^*$) will depend on the relevant enumerated sequences that are being used to unwind this iterative component (we return to this towards the end of the section).

Finally, we consider how $MEA_4$ uses the enumerated sequences to instantiate operators within iterative components of macros (it uses the same strategy as $MEA$ to instantiate single operators that appear in the iterative macros). The first time that an iterative component is selected from a macro (as was the case when $[o_2]^*$ was retrieved from $m_1$ earlier), $MEA_4$ also retrieves the relevant sets of enumerated sequences for this component. These values are then used to instantiate the iterative component as it is unwound. As an example, consider what happens when the iterative component $[drive\_load(C, S, S')]^*$ is selected from the following macro:

$$m = [drive\_load(C, S, S'')]^*, unload(C, S'', S')]$$

This component changes the value of the static sort *space* and hence values of *space* form the start and end values for the enumeration. If we suppose that the start value is $s14$ and the end value is $s12$ then these values are used to retrieve the sets of sequences of instantiations for sort *space*. There is only one enumerated sequence for these start and end values, as follows:

$$[s14, s10, s6, s7, s8, s12]$$

and these values will be used to instantiate $[drive\_load(C, S, S')]^*$ when it is selected. When the iterative component has been unwound using all the values in the enumerated sequence the sequence of operators in the partial plan (excluding *current* and *goal*) will be as follows:

$$[drive\_load(crate1, s14, s10), drive\_load(crate1, s10, s6),$$
$$drive\_load(crate1, s6, s7), drive\_load(crate1, s7, s8),$$
$$drive\_load(crate1, s8, s12), \ldots]$$

At this stage $MEA_4$ no longer selects the iterative component from the macro since it has been completely unwound.

A final observation is that whenever the sequence of operators (excluding *goal* and *current*) in a partial plans' temporal order do not "match" any of the relevant macros then $MEA_4$ reverts to the strategy that is used by *MEA* for selecting achiever operators and instantiating them.

### 6.2.3 Planning with *MEA* and ABGEN

ABGEN is our adaptation of the ALPINE algorithm [37] that we have reported in [48, 50]. It generates abstraction hierarchies similar to those generated by ALPINE (the correspondence of the methods is discussed in appendix A.10 on page 168).

For the purposes of comparison we have implemented a version of *MEA* that has been changed to use an ABGEN compiled domain model, in a planning application that we refer to as $MEA_1$. In fact the changes that were made to *MEA* are very similar to those that were made for $MEA_3$, the planning application formed from binding *MEA* with a PRECEDE compiled domain model. The top-level algorithm $MEA_1$ is the same as *MEA* except that it accepts an extra input parameter, a problem specific abstraction hierarchy. This is used in much the same way as the PRECEDE partial order, to order the earliest goals in a partial plan to be achieved first and to place partial plans in the priority queue.

The most noticeable difference between *MEA* with ABGEN ($MEA_1$) and *MEA* with PRECEDE ($MEA_3$) is that for $MEA_1$, compilation must be done at the start of each planning session because the hierarchies are problem specific. As we saw for PRECEDE, compilation need only be done once for any modelled planning domain.

### 6.2.4 Planning with *MEA* and CHAR

CHAR is an experience based learning mechanism for planning systems that is described in detail elsewhere (see for example [46] and [50]). It is a method that produces heuristics for operator selection during planning through analysis of past successful planning traces and a combination of deductive and inductive learning techniques. The basic change to the *MEA* algorithm to accommodate CHAR's heuristic rules is that choice during plan specialisation is cut down as a result of the use of the induced heuristics.

The heuristics that are output by CHAR are not directly comparable with PROPOSEs' iterative macros. However, the approaches are related since both CHAR and PROPOSE affect the choice of operators during planning. In addition, the effect of CHAR heuristics on the performance of *MEA* has already been demonstrated [46] and so this provides an interesting point of comparison for $MEA_4$.

We shall refer to the planning application, formed when *MEA* is bound with a CHAR compiled domain model as $MEA_2$.

### 6.2.5 Summary

In this section we have defined ways that the *MEA* algorithm can be changed to use domain models that have been compiled with different compilation methods. These are only meant to be illustrations and are not the only ways that the different forms of compiled domain knowledge could be used.

The different planning applications that we have considered result from binding the plan generation engine *MEA* with domain models compiled by:

- ABGEN: the planning application $MEA_1$
- CHAR: the planning application $MEA_2$
- PRECEDE: the planning application $MEA_3$
- PROPOSE: the planning application $MEA_4$

We can also create planning applications by binding *MEA* with domain models that have been compiled by a combination of compilation methods. We will combine compilation methods that affect *different* partial plan specialisation choices. These specialisation choices are the choice of which outstanding goal to achieve next, which operators to use to achieve the selected goal and which operator bindings to use. We do not combine compilation methods that affect the same plan specialisation choices because either they contain the same information, in which case nothing can be gained by combining them or they contain different information, in which case there is no information available to reconcile the differences. For example, there is a correspondence between the goal orders generated by ABGEN and PRECEDE which mean that there is little to be gained by combining the compilation methods.

A number of different planning applications were formed by binding *MEA* with domain models compiled by several combinations of compilation methods. These are as follows:

- ABGEN and CHAR: the planning application $MEA_5$
- ABGEN and PROPOSE: the planning application $MEA_6$
- PRECEDE and CHAR: the planning application $MEA_7$
- PRECEDE and PROPOSE: the planning application $MEA_8$

The combination of off-line compilation with PRECEDE and PROPOSE in planning application $MEA_8$ is of particular importance to this thesis. It is our contention that when used individually to compile a domain model both PRECEDE and PROPOSE will result in improved planning performance. We also contend that the two methods are complementary and their combination will improve planning performance by more than either method in isolation.

## 6.3 Analysis of Search Reduction

The amount of search reduction that results from using a compiled domain model, depends in part on the size of the search space that is traversed by the original planner. In this section, we look at the search space traversed by *MEA* and then we consider how the use of PRECEDE goal orders and PROPOSE macros can affect search during planning.

The search space traversed by a planner can be modelled in terms of two factors, the *depth* of the search, called $d$, and the average *branching factor* of the search, called $b$ [41]. The depth of the search for a problem depends on the number of operators in the solution plan for the problem. The average branching factor of a search for planners in the class of DIP's is the average number of partial plans that result from specialising a single partial plan. In turn, this depends on a number of factors including:

- the number of goals to be achieved in the partial plan
- the number of achievers for each goal
- the number of instantiations of each achiever

If a planner uses an exhaustive search strategy, such as breadth-first search, then the size of the search in the worst case is (from Korf [41]):

$$O(b^d)$$

*MEA* does not exhaustively search the search space for a problem. Instead, it breaks down a conjunctive goal problem into a series of smaller problems which it assumes can be solved individually and then combined. The result of this is that the search for the problem becomes the sum of the searches for each of the smaller problems. The search space that will be traversed in the worst case now depends on the average branching factor and the depth for each sub-problem (for a sub-problem $p$ we write this as $d(p)$). In other words, for some planning problem where $G$ is the conjunctive goal $\{g_1, g_2, \ldots, gn\}$ then the size of the search in the worst case becomes:

$$O(b^{d(g_1)} + b^{d(g_2)} + \ldots + b^{d(g_n)})$$

*MEA* uses a heuristic that always selects the *earliest_goals* in a partial plan to achieve next (as defined on page 6.1). Also it is order dependent, so for example if the earliest operator in a plan is $pushthrudoor(Robot, Box, D, Room')$ and the earliest goals from this operator are:

$$\{at\_door(Box, D, Room), in\_room(Box, Room), near(Robot, Box), open(D)\}$$

then the strategy of *MEA* is to select the first goal from the set to achieve next. So in this case, the first goal selected will be $at\_door(Box, D, Room)$. This heuristic means that the average branching factor of *MEA*'s search is less than if all outstanding pre-condition goals of a partial plan were considered.

Lets now look at the way that the use of PRECEDE compiled goal orders in $MEA_3$ affects search during planning. As we have seen, one of the factors that influence the size of the average branching factor, and hence the size of the search, is the number of goals that can be achieved for each partial plan. We also saw how the branching factor of search with *MEA* is reduced because it heuristically selects the earliest goals in a partial plan to achieve first and it arbitrarily selects one goal to achieve based on the order of the set of earliest goals. The effect of PRECEDE goal orders on *MEA* is that rather than arbitrarily selecting one goal to achieve next, the compiled goal order $P$ is used to select which of the earliest goals to achieve next. To illustrate this consider the following example partial plan:

$$pp(\{current, goal, o_3, o_2, o_1\}, \{(current, o_1), (o_1, o_2), (o_2, o_3), \ldots\},$$
$$\{(a, o_1), (b, o_1), (c, o_1), (d, o_2), (e, o_2), (f, o_3), \ldots\}, \{\ldots\})$$

where the earliest outstanding pre-condition goals in $pp$ are:

$$earliest\_goals(pp) = \{a, b, c\}$$

*MEA* considers the earliest goals in the partial plan and arbitrarily selects the first of these goals to achieve next. In this case this is the goal $a$. Now if we suppose that the PRECEDE goal order $P$ for this domain is:

$$P = \{(c, b), (c, a), (b, a), \ldots\}$$

where $c$ can be achieved before $b$ and $a$ then $MEA_3$ will select $c$ from the earliest goals of $pp$ as the goal to be achieved first when specialising the partial plan $pp$.

Of course the amount of reduction in search depends on the number of orders that are contained in the goal order $P$. In the best case, the PRECEDE goal order $P$ would be a total order over all pre-condition predicates. The worst case would occur if PRECEDE compilation identified no orders between pre-condition subgoals. In this case search with $MEA_3$ would be similar to the search for *MEA* since it would revert to the strategy of arbitrarily selecting goals to achieve.

Finally, lets look at how PROPOSE compiled macros and enumerated sequences in the planning application $MEA_4$ affect search during planning. As we saw earlier in the section, two factors that affect the average branching factor of search are the number of achiever operators for each goal and the number of ways that each operator can be instantiated. $MEA_4$ uses PROPOSE macros to select and instantiate achiever operators during plan generation.

Once an outstanding goal has been selected, *MEA* then generates the set of instantiated achievers for the selected goal (the set of *i_achievers* was defined on page 15). The strategy of $MEA_4$ is different to this since it uses (as far as possible) the iterative macros, for task configurations in the problem goal, to identify achiever operators. It will also use the values in any relevant enumerated sequences to select instantiated achiever operators.

In the best case this can reduce the number of instantiated operators to achieve a selected goal from a partial plan to a single operator. This will occur when there is one relevant iterative macro for a particular task configuration and there is only way of unwinding the iterative component in the macro. As an illustration of this, consider the warehouse world problem as shown in figure 6.1.

For the problem shown in the figure there is only one task configuration for the sort *truck* as follows:

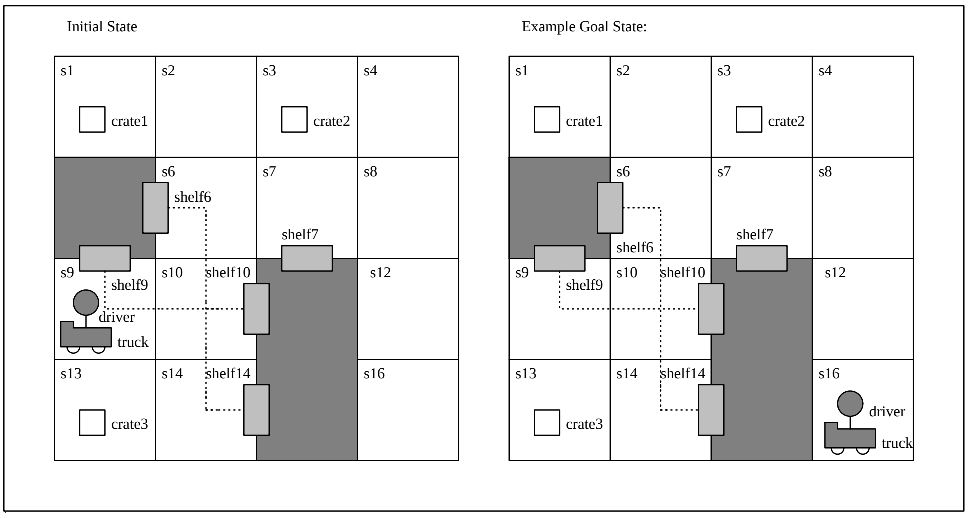


Figure 6.1: $WW$: example problem

$$(\{truck\_at(s9), truck\_unloaded\}, \{truck\_at(s16), truck\_unloaded\})$$

and there is only one macro in the set of macros for this task configuration (this is taken from the set $\mathcal{M}$ for this modelled planning domain which are listed in appendix C.6). This is as follows:

$$M_{truck}(\{truck\_at(s9), truck\_unloaded\}, \{truck\_at(s16), truck\_unloaded\}) = \{$$
$$[[drive(S, S')]^*]\}$$

which contains one element, with an iterative component made up of one operator $drive(S, S')$. The iterative component changes the value of sort *space* and in this problem its start value will be $s9$ (from the initial situation of *truck*) and the end value will be $s16$ (from the goal situation of *truck*). So when the first operator for the goal is selected the whole macro can be instantiated, using $s9$ and $s16$ as the start and end values respectively. There is only one enumerated sequence for these values which is:

$$[s9, s10, s6, s7, s8, s12, s16]$$

$MEA_4$ will now use these values to instantiate the iterative macro part whenever it is selected as an achiever operator in the context of this task configuration for sort *truck*. So, in this case the consequence of using PROPOSE iterative macros and enumerated sequences is to reduce to one, the number of ground achiever operators to be considered.

In the worst case the size of search with the planning application may be larger for $MEA_4$ than for $MEA$. This can occur when the total temporal order of operators in a partial plan does not correspond to any of the iterative macros that are relevant for the task configurations contained in the planning problem. In this situation, search with $MEA_4$ may be increased because it must revert to the strategy for operator selection and instantiation used by $MEA$. This is a consequence of the way that the $MEA$ algorithm has been extended so that it can accept a PROPOSE compiled domain model and not the PROPOSE compilation itself. We will discuss this further in section 8.2.2.

So our expectation is that $MEA_4$ will improve performance on some problems in the sample and for most of the other problems the performance of $MEA_4$ will be much the same as *MEA*. It is possible that performance will be degraded and we will consider this when we look at the results of the empirical tests.

## 6.4 Summary

So that we could test the effect of PRECEDE and PROPOSE compilation on planning performance an example planner was selected from the class of generative planners that we are interested in. The planner that was chosen is called *MEA* and is relatively typical of the class of DIP. It contains heuristics which make it quite efficient and this was one of the prime reasons why it was chosen since if PRECEDE and PROPOSE compilation improve its performance then they might also improve the performance of less efficient DIP's. Another factor was that the performance of *MEA* has already been tested with other learning and compilation methods [50].

We are interested in a number of different planning applications that can be formed by binding *MEA* with different domain models. For each of the example modelled planning domains $SW$, $SW^2$ and $WW$ we can compare the performance of *MEA* when bound with the uncompiled model of these planning domains and domain models that have been compiled by PRECEDE, PROPOSE, ABGEN, CHAR. We can also compare the performance of *MEA* when bound with domain models that have been compiled by selected combinations of these methods.

Analysis of the effect of compiled domain models on search during planning suggests that the use of PRECEDE and PROPOSE compilation will result in improved planning performance. We would also expect the effects of PRECEDE and PROPOSE to be complimentary. In the next chapter (chapter 7) we present the results of empirical tests of the performance of the different planning applications.

# Chapter 7

# Empirical Evaluation

Earlier, we defined compilation for planning as a process that transforms a model of a planning domain into a form that can be more efficiently reasoned about by a generative planning system. This requirement, that the compilation be useful, is important since the cost of using compiled knowledge can be high. This is the utility problem that was observed for example by Minton [52] and which we discussed in section 3.3.4.

The results of the earlier evaluation of PRECEDE and PROPOSE suggest that the compiled domain models that they produce will be useful for improving planning. In section 4.3 we proved that PRECEDE goal orders are always consistent with the orders in ALPINE abstraction hierarchies [37] and in section 6.2.1 we showed how the use of PRECEDE goal orders could reduce search during planning. In section 5.6 we considered the feasibility of using the PROPOSE iterative macros during planning. The purpose of this chapter is to gather empirical evidence of the effect of compilation with PRECEDE and PROPOSE on the performance of the planner *MEA* which was selected and discussed in chapter 6.

We start the chapter by considering the objectives of the tests of PRECEDE and PROPOSE (section 7.1). This is followed by a discussion of what performance metrics to use in assessing the utility of the compilation methods (section 7.2). In section 7.3 we describe the design of the tests and this is followed in section 7.4 by the results of the tests. Finally, we discuss the results of the tests overall and assess how far the results that were obtained support the objectives (section 7.5).

## 7.1 Objectives

We have tested the performance of the basic planner *MEA* and a number of planning applications formed by binding *MEA* with domain models compiled by a variety of different compilation methods (section 6.2.5) and the results of the tests will be discussed in section 7.4. We summarise the objectives of these empirical tests as follows:

1. The utility of compilation with ABGEN (our implementation of the ALPINE algorithm) and CHAR (a variation of the heuristic acquisition first discussed in [46]) has been demonstrated [50]. We expect that the current tests will replicate these earlier results. We are interested in the performance improvement of ABGEN and CHAR since we wish to compare this with the improvement with

PRECEDE and PROPOSE compilation. As we discuss later (points 3 and 4) it is our contention that PRECEDE and PROPOSE will perform at least as well as these related methods.

2. In section 4.3.2 we showed the correspondence between PRECEDE goal orders and ABGEN abstraction hierarchies. In the tests we want to show that the use of PRECEDE goal orders in $MEA_3$ improves performance by at least as much as the use of ABGEN abstraction hierarchies in $MEA_1$.

3. The analysis in section 4.3.1 suggested that PRECEDE compiles useful goal orders for the warehouse world $WW$, unlike the rather flat hierarchies compiled by ABGEN for this modelled planning domain. Consequently, we want to show that performance of the planning application $MEA_3$, using PRECEDE goal orders, is better than $MEA_1$, using ABGEN hierarchies, for this model of the warehouse planning domain.

4. Analysis of the macro operators and enumerated sequences (section 5.6) suggested that they might be useful in improving planner performance. The aim of the tests of PROPOSE is to see whether they improve planner performance on the sample of problems that are tested. We will also compare the results against the results for CHAR, because although not directly comparable the methods are related since they both affect operator choice during planning.

5. We want to confirm that the effect on planning performance of PRECEDE and PROPOSE is complementary. If it is then planning application $MEA_8$, where the domain model has been compiled by PRECEDE and PROPOSE, will perform better than $MEA_3$ and $MEA_4$.

## 7.2 Performance Metrics

To assess the impact of compilation methods on the performance of planners in the class DIP we need to find suitable performance metrics. Candidate performance metrics for planning include:

- the amount of CPU time required to solve a series of problems
- the number of partial plans specialised in the course of solving a series of problems

However, as illustrated by Segre and colleagues [64], neither of these performance metrics is totally satisfactory. Although their criticisms were directed specifically at evaluations of EBL such as [65] and [53] they also appear applicable to the learning and compilation methods that we are interested in. There are problems with taking absolute measures of cumulative CPU time since they can be machine dependent and sensitive to the way that compilation methods and planning algorithms have been implemented. Measuring the number of partial plans that are specialised during planning can also be misleading since the cost of specialising a partial plan can be sensitive to how the domain model has been compiled. Because of this we have decided to use both cumulative CPU time and partial plans as performance metrics for the tests and to compare performance across these two measures.

Further problems arise from the need to impose *resource limits*, regardless of which performance metric is being used, in order to guarantee planner termination. The problem is that performance figures are dependent on the resource limit that is imposed and changes in the resource limit can change the amount of improvement in performance attributed to different compilation functions (as demonstrated in [64]). For example, with a very low resource limit (either on CPU or partial plans) the basic planner may only be able to solve a few problems, while some planning application with a compiled domain model can solve significantly more. In this cas the improvement in performance is attributed to f partial plans that are specialised during planning can the compilation function. However, increasing the resource limit might increase the number of problems that the basic planner can solve but leave the number solved by the compiled planner unchanged. If this is the case the degree of improvement attributed to the compilation depends on the resource limit that is imposed. This makes it difficult to draw conclusions about the amount of improvement caused by particular compilation methods.

One way that this problem can be alleviated is to *control for correctness* [64]. For our purposes this would mean removing from a problem sample all problems that cannot be solved by any planning application. This means that planner performance is compared only for problems that all the planning applications can solve. However, this means that the results will not show which compilation methods improve the *resource limit competence* of the planner [64] in other words the number of problems that the planner can solve.

Consequently, we have decided to use two types of problem samples in the empirical tests of PRECEDE and PROPOSE, referred to as *uncontrolled* and *controlled*. An uncontrolled problem sample is a randomly generated set of problems and a controlled problem sample is a randomly generated set of problems where all problems that the basic planner *MEA* could not solve (within a resource limit of 2000 partial plans specialised) are removed from the sample. So in a sense this sample is only "partly" controlled since we know that *MEA* can solve all of the problems but the resource limited competence of the compiled planning applications on the problem sample is unknown. The intention of controlling the sample in this way was to bias performance in favour of *MEA* and allow us to more confidently attribute performance improvements for the compiled planning applications to the compilation.

Two other performance metrics that will be measured are the total number of operators that are used in solution plans and the total number of problems that are solved from a problem sample. The number of operators that are used in plans gives us a (very rough) indication of the quality of a plan, where the fewer the number of operators in a plan the higher the quality of the plan[1]. The total number of problems solved by a planning application is another useful metric since we want to be able to see how far the resource limited competence of the basic planner is changed by compilation. In summary the metrics that will be measured in the tests are:

- $c$: cumulative CPU time
- $p$: cumulative number of partial plans specialised
- $o$: cumulative number of operators
- $s$: number of problems that were solved

[1] It should be noted that all plans generated by *MEA* are correct in the sense that they solve the input planning problem, as mentioned earlier in section 6.1.

## 7.3 Design of Empirical Tests

As we saw in section 6.2 there are a number of different compilation methods and combinations of compilation methods that we will be considering. The basic planning engine that we will be using in the tests is *MEA* and we will also use this term to refer to the basic planning application that is formed by binding *MEA* with some non compiled model of a planning domain.

The compiled planning applications, numbered $MEA_1$ through to $MEA_8$ are formed from binding the planning engine *MEA* with the corresponding compiled domain model. These different combinations are summarised in table 7.1. The columns to the right of the table are labelled ABGEN, CHAR, PRECEDE and PROPOSE and rows are labelled with the names of the different planning combinations and compilation functions. Ticks in a row denote that the planning application uses a domain model that has been compiled with that compilation method. So for example, *MEA* is the planning application formed by binding the basic plan generation engine with the uncompiled domain model and $MEA_7$ is the planning application formed by binding *MEA* with a domain model that has been compiled by CHAR and PRECEDE.

| Planner | Compiler | **ABGEN** | **CHAR** | **PRECEDE** | **PROPOSE** |
|---|---|---|---|---|---|
| $MEA$ | – | | | | |
| $MEA_1$ | $C_1$ | $\surd$ | | | |
| $MEA_2$ | $C_2$ | | $\surd$ | | |
| $MEA_3$ | $C_3$ | | | $\surd$ | |
| $MEA_4$ | $C_4$ | | | | $\surd$ |
| $MEA_5$ | $C_5$ | $\surd$ | $\surd$ | | |
| $MEA_6$ | $C_6$ | $\surd$ | | | $\surd$ |
| $MEA_7$ | $C_7$ | | $\surd$ | $\surd$ | |
| $MEA_8$ | $C_8$ | | | $\surd$ | $\surd$ |

Table 7.1: Summary: planning and compilation combinations

It was decided to use the planning worlds $SW$, $SW^2$ and $WW$, that we have used as illustration in earlier chapters, for the empirical tests. We decided to limit the tests to these domains, since they seemed sufficiently complex and different for our purpose of demonstrating the effect of compilation with PRECEDE and PROPOSE. In addition we wanted to avoid the painstaking task of validating and modelling of any additional planning domains, especially since these worlds had been substantially tested already. On the other hand, well known toy planning worlds such as the blocks planning world and tower of hanoi (as used in for example [36]) were considered too simple to draw any conclusions from (we discuss this further for PROPOSE in section 8.2.2). Full details of the modelled planning domain that were used can be found in appendices A, B and C but here we summarise them as follows:

- **SW**: this is a seven room, ten door STRIPS like planning domain defined in [46], with a robot capable of moving boxes, opening doors and so on
- **SW**$^2$: a two robot domain similar to $SW$ but with two robots, called ann and albert, who both have the capability of the single robot. Typical solutions to tasks in this world combine the resources of ann and albert, and a considerable

| | **ops** | **preds** | **states** | **rules** |
|---|---|---|---|---|
| SW | 207 | 208 | $7 \times 10^6$ | 27 |
| $\mathrm{SW}^2$ | 414 | 240 | $10^9$ | 27 |
| WW | 482 | 290 | $2 \times 10^6$ | 53 |

Table 7.2: Planning Domains: size information

increase in complexity arises from the reasoning involved in how best to utilise the two robots (see table 7.3)

- **WW**: this world is a refinement of the warehouse world used in [46]. An agent, crane, truck and crates populate a warehouse that requires shelves to be stacked. The warehouse is cramped and much of the planning problem is in moving crates to clear paths.

Table 7.3 contains some information on the relative size and structure of each of these worlds, for comparative purposes. Column **ops** gives the number of operator instances, **preds** the number of predicate instances, and **states** the approximate number of world states. The final column **rules** gives the combined number of parameterised rule and axiom schemas that were posed by the modeller to specify relations between literals, and to specify inconsistent states.

The samples of problems were randomly generated using an automatic task generator developed as part of the FM planning system. This allows the user to specify which planning world to generate problems for and the average number of goals in each of the conjunctive goal problems. The *uncontrolled* problem sample consists of 100 tasks which have been randomly generated and these are listed in appendices A.9.2, B.9.2 and C.9.2 for $SW$, $SW^2$ and $WW$ respectively. The controlled problem samples were assembled by first generating a random sample of problems, then removing all problems that $MEA$ could not solve. This process was repeated with randomly generated problems until a sample of 100 tasks that $MEA$ could solve was obtained. These are listed in appendices A.9.1, B.9.1 and C.9.1 for $SW$, $SW^2$ and $WW$ respectively.

All software was written in SICStus Prolog (version 2.1 #8) and all tests were run under UNIX on a dedicated SUN SPARC station 1+ with 16 MB RAM and a 500 MB internal hard disc.

There is a total of 54 tests, since there are 9 planning applications ($MEA$ and $MEA_1$ through $MEA_8$) to be tested on each of the 3 planning worlds and for each planning world there are 2 problem samples.

A test harness was implemented to drive each of the 54 tests. It takes as input:

- a model of one of the test planning domains optionally compiled with one of the compilation functions $C_1$ through $C_8$
- either a controlled or uncontrolled problem sample for the planning world

and it outputs:

- performance on each problem in the problem sample for $c$, $p$, $o$ and $s$
- cumulative performance over the entire problem sample for $c$, $p$, $o$ and $s$

An important feature of each test was that a new SICStus Prolog session was initialised for each new planning problem. This ensured that the size of the Unix process was the same at the start of each planning session within each test.

## 7.4 Results

In this section the results of the empirical tests are presented. For ease of presentation we have grouped together the results for each of the different planning worlds and within each group the results are discussed in turn firstly for the controlled and then for the uncontrolled problem samples.

For the controlled planning systems we present two tables of results. The first contains the results for all of the problems in the sample, where *MEA* could solve all of the problems. The second table summarises the results from the set of problems formed by removing any that could not be solved by all of the planning applications, in other words it is the sample of problems that could be solved by all of the planning problems.

The results are shown in table format and full listings of all the test results are given in the appendices. The results for $SW$ can be found in appendix A.8 on page 148, for $SW^2$ in appendix B.8 on page 178 and for $WW$ in appendix C.8 on page 209.

The tables of results in this section summarise the results obtained for the four performance metrics. These are marked $c$, $p$, $o$ and $s$ respectively on the left hand side of the rows in the table. The columns of the tables of results are marked as follows. The most left hand column *MEA* gives the results for the basic planner. The columns $MEA_1$, $MEA_2$, $MEA_3$ and $MEA_4$ summarise the results for the planning functions formed by binding MEA with a domain model compiled by either ABGEN, CHAR, PRECEDE or PROPOSE respectively. The four right hand columns give the results for combinations of pairs of compilation functions, $MEA_5$ for ABGEN and CHAR, $MEA_6$ for ABGEN and PROPOSE, $MEA_7$ for PRECEDE and CHAR, and $MEA_8$ for PRECEDE and PROPOSE.

### 7.4.1 $SW$: the robot world

#### Controlled Problem Sample

The results for this problem sample are summarised in table 7.3. From the table it can be seen that on both of the performance metrics $c$ and $p$, the basic planner *MEA* consistently consumes more time and specialises more partial plans than all of the compiled options which is what we would expect.

The average number of operators that are used by *MEA* are greater than for all of the other compilation options. This is interesting since it suggests that compilation might improve the *quality* of the plans that are generated, where the shorter the plan the higher its quality. However, the total number of operators must be considered in combination with the number of problems that were solved. We can see that although $MEA_5$ has the lowest cumulative number of operators, it also solves the fewest problems.

Interestingly $MEA_3$, $MEA_4$ and $MEA_8$ solve all of the problems in the sample, but use less operators than *MEA*. This suggests that PRECEDE and PROPOSE compilation,

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 1905 | 562 | 1126 | 546 | 991 | 442 | 630 | 483 | 624 |
| **p** | 15608 | 7688 | 8038 | 7254 | 10051 | 4910 | 7181 | 5041 | 6949 |
| **o** | 2022 | 1883 | 1914 | 1972 | 2016 | 1657 | 1874 | 1875 | 1982 |
| **s** | 100 | 99 | 94 | 100 | 100 | 91 | 99 | 95 | 100 |

Table 7.3: $SW$: Results for controlled problem sample

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 1480 | 397 | 883 | 387 | 782 | 341 | 440 | 344 | 438 |
| **p** | 11567 | 5406 | 6162 | 5125 | 7644 | 3682 | 4984 | 3617 | 4900 |
| **o** | 1521 | 1449 | 1576 | 1464 | 1514 | 1415 | 1428 | 1501 | 1464 |
| **s** | 81 | 81 | 81 | 81 | 81 | 81 | 81 | 81 | 81 |

Table 7.4: $SW$: controlled problem sample – all problems solved

both individually and in combination might increase solution quality.

$MEA_2$ does rather poorly, in that it only manages to solve 94 of the tasks but this can be explained since CHAR can occasionally produce chunks that are not useful, these were referred to by McCluskey as *bad* chunks [46, pp 63].

For the single compilation functions the related compilation methods consume roughly the same amount of CPU time and specialise a similar number of partial plans. Compilation that affects goal choice, encapsulated in planning applications $MEA_1$ and $MEA_3$ improved performance, measured against CPU, by a factor of approximately 4. The methods that affect operator choice, encapsulated in planning applications $MEA_2$ and $MEA_4$, improved performance by a factor of approximately 2. So it appears that compilation that affects goal choice leads to more improvement in performance in this modelled planning domain than the methods that affect operator choice.

If we now consider table 7.4 which shows the results for the sample of problems that all of the planning applications could solve (this is a sample of 81 problems as described in appendix A.9.1) we can see that this shows a similar pattern of results to those in table 7.3. We can see that compilation with PRECEDE and PROPOSE improve the performance of $MEA$ in terms of both the amount of CPU consumed and the number of partial plans that are specialised. We can also see that both PRECEDE and PROPOSE generate slightly shorter plans which suggests that solution quality can be improved by compilation for certain problem samples.

The effect of combining PRECEDE and PROPOSE compilation in application $MEA_8$ is complementary with respect to the number of partial plans that are specialised but not when measured against the amount of CPU that is consumed. This is a similar result to that shown in table 7.3 but it may be the case that the cost of specialising partial plans when a domain model has been compiled could be improved by optimising the way that the software in $MEA_4$ and $MEA_8$ has been implemented.

### Uncontrolled Problem Sample

In table 7.5 the results are summarised for the sample of problems that had not been controlled for correctness. As predicted the options $MEA_1$ and $MEA_2$ consistently speed up $MEA$. In the tests reported in [50] the speed up was by a factor of approximately 5 for $MEA_1$, by a factor of about 2 for $MEA_2$ and when combined the speed up was also about factor 2. In this set of experiments the speedup on CPU time was by about a factor of 4 for option $MEA_1$ and by about factor 2 for $MEA_2$. For both options on performance metric $p$ the speed up was about factor 2. This is to be expected since the matching costs per partial plan are more expensive for $MEA_2$ and hence it can take longer (consume more CPU time) than $MEA_1$ to specialise the same number of partial plans. It is interesting to note that $MEA_1$ solves one more task than $MEA$ but with considerably fewer operators, which suggests that in this world at least $MEA_1$ helps improve solution quality. We shall consider in the next few sections if this extends across the other modelled planning domains.

For planning application $MEA_3$, with PRECEDE, on performance metrics $c$ and $p$ the improvement over $MEA$ is comparable to $MEA_1$, with ABGEN. Slightly less CPU time is consumed and fewer partial plans are specialised by $MEA_3$ which initially suggests that it takes slightly less CPU time to specialise each partial plan than $MEA_1$. However,

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 2016 | 572 | 992 | 559 | 1076 | 450 | 634 | 576 | 639 |
| **p** | 16202 | 7768 | 8383 | 7415 | 10511 | 5007 | 7222 | 5639 | 7070 |
| **o** | 1947 | 1855 | 1863 | 1947 | 1941 | 1615 | 1846 | 1709 | 1957 |
| **s** | 96 | 97 | 91 | 98 | 96 | 88 | 97 | 87 | 98 |

Table 7.5: $SW$: Results for uncontrolled problem sample

we must remember that $MEA_1$ is problem specific and it generates an abstraction hierarchy for each problem that is input. Since the time taken by ABGEN to generate each hierarchy is included in the overall CPU figure for $MEA_1$ the overall CPU figures suggest that $MEA_3$ takes slightly longer to specialise each partial plan than $MEA_1$.

The performance of planning application $MEA_4$ with PROPOSE is comparable on the performance metrics $p$ and $c$ with $MEA_2$. From the table we can see that both planning applications consume roughly the same amount of CPU time. It appears that $MEA_4$ specialises more partial plans than $MEA_2$ however two factors explain this. Firstly, $MEA_4$ solves more problems than $MEA_2$ and secondly, cost of specialising a node with $MEA_2$ is greater than for $MEA_4$. In terms of solution quality, performance metric $o$, they are similar although $MEA_4$ solves slightly more problems than $MEA_2$.

Taken as a whole the results for the single compilation functions, are consistent with those for the controlled problem sample, they suggest that compilation methods that output goal orders that are used during planning to select *goals* are more effective in improving planning performance in the robot world $SW$ than compilation for *operator and binding choice*.

If we now consider the results measured against performance metric $c$, for the combined compilation methods we see an improvement overall of a factor of between 2 and 4 which is consistent with earlier results reported in [50]. The combination of compilation methods PRECEDE and PROPOSE in $MEA_8$ improves slightly in terms of the number of partial plans specialised but not in terms of the amount of CPU consumed. However, the overall performance of $MEA_8$ is good and if we compare the performance of it with $MEA_4$ in terms of $c$, its performance is nearly twice as fast. This clearly shows the way that the compilation methods PRECEDE and PROPOSE can complement each other. Also, the planning application $MEA_8$ has better coverage, in terms of $s$ than some of the combined planning applications, such as $MEA_5$ and $MEA_7$.

The combination of compilation methods PRECEDE and PROPOSE in planning application $MEA_8$ suggest that these compilation methods are complementary. The number of partial plans that are specialised by $MEA_8$ is slightly less than either $MEA_3$ with PRECEDE or $MEA_4$ with PROPOSE. The amount of CPU time consumed by $MEA_8$ is slightly more than $MEA_3$ but it is considerably less than for $MEA_4$. This demonstrates how combining the two compilation methods can lead to greater performance improvement than for single compilation methods.

#### Summary of results $SW$

The results show that all the individual compilation methods that were tested improve the performance of $MEA$. As expected the results for $MEA_1$ and $MEA_2$ do mirror our earlier results with ABGEN and CHAR. The results also show the expected correspondence between improvements in planning performance with ABGEN and PRECEDE. One of our objectives was to see how far the planning application $MEA_4$ that uses PROPOSE improves planning performance and the results show a consistent improvement in terms of the number of partial plans that are specialised and the amount of CPU time consumed.

The results for the combined pairs of compilation methods ($MEA_5$, $MEA_6$, $MEA_7$ and $MEA_8$) show a consistent improvement, measured in terms of CPU time, by a factor of roughly 4 and measured in terms of the number of partial plans specialised the improvement is a factor of roughly 3. Interestingly, the combination of PRECEDE and PROPOSE compilation in application $MEA_8$ appears to have the best coverage, in that it consistently solves the most problems.

Overall the results suggest that compilation methods which compile goal orders ($MEA_1$ and $MEA_3$) are more influential on planning performance in this modelled planning domain than compilation methods that output information for operator choice ($MEA_2$ and $MEA_4$). The results also clearly indicate that domain model compilation with PRECEDE and PROPOSE can have a useful effect on planning performance. The results are important since despite the good performance of the basic planning system $MEA$, in this modelled planning domain, the compilation methods PRECEDE and PROPOSE still demonstrate a clear improvement in planning performance.

### 7.4.2 $SW^2$: person world

#### Controlled Problem Sample

Table 7.6 summarises the results for this problem sample. The results reported in [50] show an improvement of about factor 2 for $MEA_1$, about factor 6 for a combined compilation function that included $MEA_1$ and $MEA_2$, but there was no appreciable improvement for compilation with $MEA_2$ for an uncontrolled problem sample in this two robot planning world $SW^2$. The results shown in table 7.6 are consistent with these results. We can see that $MEA_1$ improves performance, measured in CPU and partial plans, by a factor of about 2 over $MEA$. For $MEA_2$ the performance is slightly worse for both $c$ and $p$. However, this is not inconsistent with the earlier results since they were for an uncontrolled problem sample and as we saw in the discussion earlier in this chapter, controlled problem samples can bias against compilation since it can change the resource limit competence.

The results for compiled planning with $MEA_3$ (PRECEDE) are better than those for $MEA_1$ in terms of both the number of partial plans specialised and CPU consumed. The coverage is also similar, it solves the same number of problems, but uses slightly more operators.

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 6282 | 3150 | 7912 | 2653 | 4209 | 3188 | 3128 | 1563 | 1252 |
| **p** | 25249 | 13209 | 21146 | 12401 | 19276 | 11249 | 12267 | 8509 | 8777 |
| **o** | 1285 | 1217 | 1328 | 1247 | 1203 | 1283 | 1245 | 1290 | 1276 |
| **s** | 100 | 98 | 98 | 98 | 94 | 94 | 96 | 99 | 95 |

Table 7.6: $SW^2$: controlled problem sample

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 5161 | 481 | 5173 | 1868 | 3634 | 868 | 472 | 1111 | 855 |
| **p** | 20114 | 5799 | 14753 | 8871 | 15796 | 5582 | 5332 | 6143 | 6287 |
| **o** | 954 | 957 | 1014 | 967 | 1003 | 1090 | 1008 | 987 | 1043 |
| **s** | 86 | 86 | 86 | 86 | 86 | 86 | 86 | 86 | 86 |

Table 7.7: $SW^2$: controlled problem sample – all problems solved

The performance of $MEA_4$ (PROPOSE) supports the fourth objective, since it performs far better than $MEA$ and the other planning application that is compiled for operator choice, $MEA_2$, in terms of the number of partial plans specialised and the amount of CPU time consumed. However, the number of problems that $MEA_4$ solves is slightly less than for $MEA_2$.

The combination $MEA_8$ does appear to support our speculation that the result of combining a pair of compatible compilation methods will be an improvement in performance that is greater than either of the techniques on their own. For both CPU time and the number of partial plans that are specialised, the combination of the methods increases the performance improvement for the single compilation options.

Lets now consider the results for just those problems that all the planning applications can solve which are summarised in table 7.7 (this set of 86 problems are described in appendix B.9.1). We can see that the performance improvement for $MEA_3$ and $MEA_4$ is similar to the results which were summarised in table 7.6 where the compilation methods PRECEDE and PROPOSE improve performance in terms of both the amount of CPU time used and the number of partial plans that are specialised. The results also

|  | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 52295 | 11016 | 45557 | 26715 | 47010 | 7188 | 9989 | 23589 | 20540 |
| **p** | 106911 | 34560 | 91274 | 59324 | 97906 | 19811 | 30705 | 47862 | 47525 |
| **o** | 703 | 1620 | 1011 | 1353 | 759 | 1923 | 1646 | 1539 | 1318 |
| **s** | 53 | 90 | 67 | 79 | 55 | 94 | 89 | 86 | 74 |

Table 7.8: $SW^2$: uncontrolled problem sample

show the complementary nature of PRECEDE and PROPOSE since the performance of $MEA_8$, as measured against $c$ and $p$, is better than for either $MEA_3$ or $MEA_4$ where the compilation methods have been used individually.

For this problem sample we can also compare the number of operators used by the different applications since all applications solved the same problems. We can see that for all compiled planning applications, any performance improvements are at the cost of longer solution plans.

Finally, an interesting feature of these results is the considerable difference between the performance of $MEA_1$ on the controlled problem sample and on the sample of problems formed by removing any that couldn't be solved by any of the planning applications. For problems such as task number 12 and 59 (which $MEA_2$ takes 696 and 1116 CPU seconds respectively before abandoning) they distort the overall figures considerably and when they are removed from the sample the performance improvement resulting from ALPINE compilation appears greater.

### Uncontrolled Problem Sample

Table 7.4.2 summarises the results for this sample of problems. For the single compilation options $MEA_1$ and $MEA_2$ these results are consistent with the results reported in [50]. Option $MEA_1$ uses about 5 times less CPU and partial plans than the basic planner $MEA$. It is interesting to observe the high success rate of this compilation option, it can solve 90 out of the 100 random tasks whereas its nearest (single compilation) rival can only solve 79 tasks. For $MEA_2$ there is no real improvement in performance. There are a number of factors influencing this including the fact that it only learns from successful solutions, so if a task is abandoned, no learning occurs.

Compilation option $MEA_3$ (PRECEDE) reduces the time that is taken and the number of partial plans that are specialised by a factor of about 2. This is not quite as good as the amount of improvement with $MEA_1$, but when we take the results in combination with the results for the controlled problem sample it provides evidence for the second objective that it will improve performance by at least the same amount as $MEA_1$. This is because for that sample the amount of improvement with $MEA_3$ is greater than $MEA_1$, and for a controlled sample that is biased against the compiled options.

The results with compilation option $MEA_4$ (PROPOSE) show a reduction in the amount of CPU consumed and the number of partial plans specialised is reduced quite considerably. The number of problems successfully solved is quite low in comparison to, say, the number solved by $MEA_2$ (compiled by CHAR which is related to PROPOSE in so far as they both influence operator choice). However when the results are compared across the two problem samples, they do support the objective of showing that $MEA_4$ improves performance. It also interesting that the overall degree of improvement with $MEA_4$ is comparable to that for $MEA_2$. This is because performance on the controlled sample is far better with $MEA_4$ than $MEA_2$.

When we consider the combination of pairs of compilation methods, we can see that by far the best improvement is achieved by the combination of ABGEN and CHAR in $MEA_5$. Interestingly though, when PRECEDE and PROPOSE are combined in $MEA_8$ the overall amount of improvement is better than either of the compilation options on their own in either $MEA_3$ or $MEA_4$. This supports our objective of showing that the novel compilation methods PRECEDE and PROPOSE are complementary.

#### Summary of results for $SW^2$

Overall the results do show improved planning performance, in terms of the number of partial plans specialised and the amount of CPU time, when domain models have been compiled with PRECEDE and PROPOSE. Of the two, the amount of performance improvement appears slightly better with PRECEDE than with PROPOSE and the coverage of $MEA_3$, the planning application that uses PRECEDE goal orders is better than $MEA_4$, with PROPOSE.

Interestingly, PRECEDE is the best single compilation method on the controlled problem sample whereas ABGEN is far better for the uncontrolled problem sample where it is also able to solve more problems. Of the methods for operator choice PROPOSE had more impact than CHAR on the controlled problem sample but the effect on performance on the uncontrolled sample was roughly the same. Taken as a whole the results suggest that for this domain (as with $SW$), the compilation methods that influence goal choice, ABGEN and PRECEDE, have more effect on planner performance than CHAR and PROPOSE, methods that influence operator choice. Overall the results do show that PRECEDE and PROPOSE compilation can improve planning performance in this modelled planning domain.

### 7.4.3 $WW$: warehouse world

#### Controlled Problem Sample

Tables 7.9 and 7.10 summarise the results for this problem sample (full results and problem sample listings are given in appendix C). Table 7.9 gives the results from all the problems in the sample but table 7.10 excludes results from problems that could not be solved by *all* of the planning applications. In effect this makes a problem sample consisting of 73 problems that all the planning applications could solve. We will consider the results in each table in turn, starting with table 7.9.

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 8218 | 18060 | 3996 | 4757 | 7798 | 8251 | 17760 | 4821 | 4135 |
| **p** | 26308 | 39518 | 13497 | 14534 | 21956 | 20618 | 34503 | 12989 | 11938 |
| **o** | 913 | 844 | 882 | 836 | 908 | 822 | 873 | 786 | 862 |
| **s** | 100 | 96 | 93 | 97 | 98 | 91 | 96 | 87 | 97 |

Table 7.9: $WW$: controlled problem sample

The first observation is that the performance of $MEA_1$ (compilation with ABGEN) is very poor. For example, the amount of CPU time consumed is more than double that for $MEA$ (this result is similar in table 7.10 and 7.11). Earlier, in section 4.3 we discussed some of the reasons why ABGEN tends to generate flat abstraction hierarchies for some modelled planning domains, such as the warehouse world. Based on this we predicted that performance with $MEA_1$ would be poor in this world but the degradation in performance is a little surprising. Our explanation for these results is based on the order sensitivity of the basic $MEA$ plan generation algorithm. Why is this ? Although the ABGEN hierarchies are flat for this modelled planning domain this can still lead to a reordering of the set of outstanding pre-conditions. It may even be the case that the domain modeller designed the set of operators for this domain so that the pre-condition predicates were ordered in some "useful" way. If the pre-conditions are re-ordered then this would account for differences in performance between $MEA$ and $MEA_1$ with ABGEN. The effect of change of representation on planning performance and the efficacy of compilation (and learning) methods has been documented elsewhere (see for example [20]). Although this result is interesting we shall not consider it further here since our primary objectives in the tests is to demonstrate the effect of PRECEDE and PROPOSE on planning performance.

The planning option $MEA_2$ performs well on this problem sample, reducing both the total CPU time and the number of partial plans specialised by a factor of about 2.

Improvements of a similar scale are recorded for PRECEDE compilation in planning application $MEA_3$ but it manages to solve more problems than $MEA_2$. The results for PROPOSE compilation with $MEA_4$ also show improvement performance over the basic planner $MEA$. Interestingly this planning application manages to successfully solve more problems than any of the the other compiled planning applications (either single or combined).

For the combined planning functions, the performance improvement is greatest for planning application $MEA_8$. It consistently consumes less CPU time and specialises fewer partial plans. Also, the coverage of this planning application is better than all of the other combined planning applications, for example it solves 10 more problems than $MEA_7$. These results are consistent with our contention that PRECEDE and

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 4285 | 9882 | 2104 | 2873 | 3933 | 3794 | 8001 | 3371 | 2484 |
| **p** | 15682 | 23530 | 7859 | 8420 | 12916 | 11111 | 19213 | 8493 | 7256 |
| **o** | 631 | 621 | 660 | 624 | 640 | 651 | 630 | 655 | 640 |
| **s** | 73 | 73 | 73 | 73 | 73 | 73 | 73 | 73 | 73 |

Table 7.10: $WW$: controlled problem sample – all problems solved

PROPOSE are complementary compilation methods.

Lets now consider the results summarised in table 7.10. These results were obtained from just those problems in the controlled problem sample that could be solved by all of the planning applications (this set of 73 problems is described in appendix C.9.1). These results also show that compilation with PRECEDE and PROPOSE, applications $MEA_3$ and $MEA_4$ respectively, improve performance of the basic planning algorithm as measured in terms of the amount of CPU time consumed and the number of partial plans specialised.

Interestingly, the results show that performance is very good for $MEA_2$ (with CHAR) on a sample of problems that it can solve.

Since the figures in table 7.10 are just for problems that all the planning applications can solve we can compare the cumulative number of operators that were used by the different applications to successfully solve these 73 problems. The figures show that overall, application $MEA_1$, generates the shortest average solution plans, but takes twice as long as the original planner. It appears that the good performance of application $MEA_2$ for this problem sample is at the expense of longer plans, since it uses a total of 660 operators compared to the 631 of $MEA$. Application $MEA_3$ (with PRECEDE) uses slightly less operators than $MEA$ and $MEA_4$ uses slightly more. When they are combined there is no difference and the number of operators is the same for $MEA_8$ as for $MEA_4$.

The figures also show how PRECEDE and PROPOSE compilation can be complementary. When measured in terms of both cumulative CPU time and the number of partial plans specialised the combination of PRECEDE and PROPOSE compilation does lead to greater improvement in performance than either compilation method in isolation. Interestingly, planning application $MEA_8$ where compilation with PRECEDE and PROPOSE is combined, performs better than any of the other combined planning applications. In fact only planning application $MEA_2$ (with CHAR) performs better, when measured in terms of cumulative CPU time but not in terms of the number of partial plans specialised. $MEA_8$ specialises 7256 partial plans in the course of solving the 73 problems in this sample whereas $MEA_2$ specialises 7859 partial plans. The matching performed at each partial plan by $MEA_2$ is potentially far more expensive

| | $MEA$ | $MEA_1$ | $MEA_2$ | $MEA_3$ | $MEA_4$ | $MEA_5$ | $MEA_6$ | $MEA_7$ | $MEA_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **c** | 22428 | 32811 | 16424 | 14831 | 23190 | 26917 | 32045 | 10233 | 13607 |
| **p** | 52329 | 66006 | 43196 | 32811 | 47146 | 53519 | 59918 | 25332 | 27888 |
| **o** | 492 | 482 | 485 | 564 | 535 | 455 | 528 | 541 | 592 |
| **s** | 45 | 55 | 49 | 55 | 54 | 50 | 55 | 52 | 56 |

Table 7.11: $WW$: uncontrolled problem sample

than $MEA_8$ and this suggests that it may be possible to improve the impact of PRECEDE and PROPOSE on planner performance when measured in terms of cumulative CPU by optimising the software implementation of it.

### Uncontrolled Problem Sample

In table 7.11 the results are summarised for this problem sample. The results show a similar overall pattern to the results for the controlled problem sample that were discussed in the previous section. As for the controlled problem sample, performance with $MEA_1$ which uses ABGEN abstraction hierarchies is worse than the performance of the basic planner $MEA$. For compilation with CHAR in $MEA_2$ and with PRECEDE in $MEA_3$ the amount of performance improvement is roughly the same as for the controlled problem sample, the improvement is by roughly a factor of 2 over the basic planner $MEA$. However, it may be the case that the performance of application $MEA_3$ is slightly better than $MEA_2$ since it manages to solve more problems than $MEA_2$. For planning application $MEA_4$ where the domain model has been compiled by PROPOSE the results show an improvement in the amount of CPU time that is consumed and the number of partial plans specialised especially when it is noted that $MEA_4$ solves 10 more problems than are solved by the basic planner $MEA$.

Interestingly when the compilation methods are combined the biggest improvements result from combining PRECEDE with either CHAR, in application $MEA_7$ or PROPOSE, in application $MEA_8$. This supports our expectation that combining different compilation methods can increase the amount of improvement in planning performance. It also supports our contention that the methods PRECEDE and PROPOSE are complementary.

### Summary of results: $WW$

The results that we have presented show that in this modelled planning domain compilation with PRECEDE and PROPOSE, in planning applications $MEA_3$ and $MEA_4$ respectively can improve performance of plan generation, when measured against both the number of partial plans specialised and CPU time consumed. It also appears that for this modelled planning domain, compilation with PRECEDE improves planning performance by a greater degree than PROPOSE.

For both of the problem samples the results support our contention that the methods PRECEDE and PROPOSE are complementary in that when they are combined they result in greater performance improvements than for either method when applied independently. This is shown for both the cumulative number of CPU seconds that are consumed and the number of partial plans that are specialised. The results also show that combining PRECEDE and PROPOSE increases the number of problems that the planning application can solve (the resource limit competence [64]). For the controlled problem sample, PRECEDE and PROPOSE, both individually and compiled solved more problems than any of the other compiled planning applications. For the uncontrolled problem sample, the combination of PRECEDE and PROPOSE in application $MEA_8$ actually increased the number of problems that could be solved.

Finally, we note that at most, only half of the problems in the uncontrolled problem sample, could be solved by any of the planning applications. Recall that the resource limit was set at 2000 (in terms of the number of partial plans specialised). The results do suggest that the resource limit could have been more usefully set slightly higher for this modelled planning domain.

## 7.5 Summary

A number of different planning applications were formed by binding the plan generation engine of a domain independent planner with domain models that had been compiled by PRECEDE, PROPOSE, ABGEN and CHAR. These planning applications were then tested on a number of problem samples for three modelled planning domains. The main objective of these tests was to demonstrate the effect of PRECEDE and PROPOSE compilation, both individually and in combination, on the performance of a generative domain independent planner called *MEA*. We also wanted to replicate the results of earlier tests with established on-line and post-event methods and then show that PRECEDE and PROPOSE compare favourably to these established methods.

The performance of planning applications $MEA_1$, where the domain model is compiled by ALPINE and $MEA_2$ where the domain model is compiled by CHAR was reported elsewhere [50]. Overall the results for these planning applications in the current set of tests do appear similar to the earlier results.

The planning application $MEA_3$ was formed by binding the plan generation engine with a domain model compiled by PRECEDE. Overall the results showed that $MEA_3$ consistently specialises fewer partial plans and consumes less CPU time than the basic planner *MEA*. In the robot planning worlds, $SW$ and $SW^2$, the amount of improvement for $MEA_3$ was very similar to the improvement for planning application $MEA_1$ which uses ABGEN abstraction hierarchies to reduce search during planning. We had expected this result, since there is a close correspondence between PRECEDE goal orders and ABGEN abstraction hierarchies (as shown in section 4.3.2).

In addition, planning application $MEA_3$ improves planning performance in the warehouse world $WW$, a domain for which ABGEN is unable to generate useful abstraction hierarchies. This provides support for our argument that PRECEDE can generate useful goal orders for domains where ABGEN abstraction hierarchies suffer from the collapsing problem.

Planning application $MEA_4$ was formed by binding the basic planner with PROPOSE compiled domain models. This application consistently performs better than $MEA$ across all the modelled planning domains that were used in the tests. For example, in the robot planning domain $SW$, where the basic planner already performs well, $MEA_4$ consumes about half the CPU time across all the different problem samples. The extent of the performance improvement is not as large for the problem samples in the warehouse planning world but nevertheless $MEA_4$ performs better than the basic planner over the different problem samples (with one exception being the uncontrolled problem sample) and its problem coverage is good.

The combined effect of PRECEDE and PROPOSE on planning performance were tested in planning application $MEA_8$. The results show that $MEA_8$ consistently performs better than the PRECEDE planning application ($MEA_3$) and the PROPOSE planning application ($MEA_4$) in the modelled planning domains $SW^2$ and $WW$. In these domains the combined planning application consistently specialises less partial plans and consumes less CPU time during planning. For planning domain $SW$, the performance of planning application $MEA_8$ was roughly the same as $MEA_3$ since it consistently specialised fewer partial plans and consumed slightly more CPU time but the performance of $MEA_8$ was consistently better than $MEA_4$ in this domain.

Overall, the results do show that domain model compilation with PRECEDE and PROPOSE can be useful in improving the performance of generative domain independent planning systems. The results also show that the effect of compilation with PRECEDE and PROPOSE can be complementary in that combining the two methods can result in greater performance improvement than results from compilation by either method individually.

Performance improvement with PRECEDE compilation compares favourably to the on-line method embodied in ABGEN since the amount of performance improvement for $MEA_1$ (ABGEN) and $MEA_3$ (PRECEDE) is comparable in planning domains $SW$ and $SW^2$ and is far better for $MEA_3$ in $WW$. Also, the results show roughly similar amounts of performance improvement with PROPOSE in planning application $MEA_4$ and with the post-event learning method CHAR in planning application $MEA_2$. Hence, the results show that off-line compilation methods can compare favourably with established on-line and post-event methods of compilation.

# Chapter 8

# Related Work

In earlier chapters we evaluated the novel compilation methods PRECEDE and PROPOSE statically (sections 4.3 and 5.6) and empirically (chapter 7). This chapter can be seen as a third form of evaluation in which we consider the relation of PRECEDE and PROPOSE to other methods of compilation for planning. There are two aspects to this evaluation, first we consider the relation of our methods to other work. Then we consider the scope of the compilation methods PRECEDE and PROPOSE (section 8.2) to further clarify their relation to other work.

## 8.1 Related Work to PRECEDE and PROPOSE

We have already discussed specific research work that was important in the development of PRECEDE and PROPOSE (sections 4.1 and 5.1). Here, we re-consider some of these approaches but the emphasis is now on the similarities and differences with PRECEDE and PROPOSE. In this section we consider our compilation methods separately, starting with PRECEDE below and PROPOSE in section 8.1.2. Then at the end of the section we discuss some general observations that apply equally to both methods of off-line compilation.

### 8.1.1 PRECEDE: relation to other work

Work with PRECEDE was motivated by the need for a method of compiling goal orders that could be used to select which goal to achieve next during partial plan specialisation. The research work that is closely related to PRECEDE comes from a number of areas including state violation checking [14, 19] , abstraction generation [36], goal orders [33] and control rules [20] and we shall in turn consider the relation of each of these approaches to PRECEDE.

#### Checking for State Violation

We remember Dawson and Siklóssy's REFLECT system [14] because of the emphasis that was given to compilation or pre-processing and the effect that it can have on planning performance. REFLECT compiled a set of pairs of predicates ("incompatible assertions") that cannot both be true in a consistent state of the modelled planning domain.

PRECEDE uses information about inconsistent states of a modelled planning domain supplied in the form of a negative state invariant, which we call $IN$. This invariant can be either user supplied or automatically generated from $OS$, the set of operators for a modelled planning domain. We defined a condition that pairs of predicates in $IN$ must satisfy which was inspired by the informal outline that was given by Dawson and Siklóssy [14]. So there is a strong similarity between the set of "incompatible assertions" compiled by REFLECT and the negative state invariant $IN$ which is used by PRECEDE during compilation.

REFLECT uses the set of incompatible assertions during plan generation to select partial plans whose outstanding pre-conditions are consistent, but PRECEDE uses the negative state invariant $IN$ to identify goal interactions and form goal orders. This is done a priori direct from the domain model. The PRECEDE goal order can then be used during planning to select which outstanding pre-conditions to achieve next during partial plan specialisation.

Interestingly, the approach of Drummond and Currie (see for example [19]) appears similar to that of Dawson and Siklóssy. Their approach is to use *temporal coherence* (similar to consistent state checking with REFLECT's incompatible assertions or PRECEDE's negative state invariant) "on the fly" to speed up planner performance, by selecting partial plans with temporally coherent outstanding pre-conditions during plan generation. However their approach is different to the one embodied in PRECEDE since it is completely on-line and the inconsistent state information is user supplied and all other processing is performed on-line.

### Abstraction Generation

As shown in chapter 4, a useful side effect of the off-line compilation method PRECEDE is that the goal orders it outputs appear to overcome a limitation of some methods of automated abstraction generation in the tradition of ABSTRIPS [62] and more recently ALPINE [36].

We showed in section 4.3.2 (see page 61) that there is a correspondence between PRECEDE partial orders and ALPINE abstraction hierarchies. This guarantees that for any two predicates $a$ and $b$ such that $necessary\_before(a, b)$ is true, then $b$ can never be at a higher level in a problem specific ALPINE abstraction hierarchy than $a$. The results of empirical tests with PRECEDE and ABGEN (our implementation of the ALPINE algorithm) confirmed our expectation that off-line compilation with PRECEDE compares favourably, in terms of performance improvement, with ABGEN.

Despite this correspondence between the orders generated by ALPINE and PRECEDE there are a number of important differences between the methods. The first of these is that where ALPINE abstraction hierarchies are based on *possible* interactions between goals, PRECEDE goal orders are based on *necessary* interactions. We contend that because PRECEDE goal orders are based on necessary interactions between goals, they do not collapse in the same way as ALPINE hierarchies.

Another important difference between ALPINE and PRECEDE is that ALPINE generates *problem dependent* abstraction hierarchies whereas PRECEDE's partial orders are *problem independent.* ALPINE can create problem independent hierarchies although these are more susceptible to collapse than problem dependent ones. An important advantage of problem independent compilation (or off-line) is that the compilation can be done a priori direct from the domain model and hence is independent of on-line planning resources.

The output of ALPINE and PRECEDE compilation differs because PRECEDE outputs a partial order over predicates in the domain, whereas ALPINE's output is a total order over the set of predicates attached to a domain model. If there is more than one linearisation of the levels in an abstraction hierarchy then ALPINE uses heuristics to form a total order.

Finally, we note that ALPINE compilation is more dependent on the initial representation of a modelled planning domain than PRECEDE. This contention is based on the observation that ALPINE orders are based on possible interactions whereas PRECEDE orders are based on necessary interactions. This means that all interactions that PRECEDE identifies will occur regardless of the initial representation whereas ALPINE orders depend on the initial representation.

### Goal Orders

The crucial role of *goal ordering* on the amount of search performed during plan generation has been noted by many researchers. Our own tests with an implementation of the goal ordering technique of Cheng and Irani [33] support this argument [50]. Their approach offers a means whereby an ordering can be assigned to goals in a "top-level" conjunctive planning goal. Simply put, the method of compilation is to consider each $g$ in some conjunctive planning goal $G$ and to test whether $g$ can be achieved from some state in which the other goals in the set:

$$G' = \{G \setminus \{g\}\}$$

are all true, whilst not clobbering any of the goals in $G'$. If it can then the goals are ordered so that $G'$ are achieved before $g$. As part of the compilation, the consistency of the pre-conditions of all achievers of $g$ and $G'$ is tested.

However, the compilation method ignores precisely those interactions that PRECEDE uses during compilation, which are the ones that always occur amongst operator pre-conditions. It ignores them because only ground operators that don't clobber the goal set are considered. Yet this is precisely the situation that PRECEDE tests, namely, where there is a necessary interaction between a pair of goals that must be true in the same state. In section 8.2 we will consider how PRECEDE can be combined with this goal ordering approach of Cheng and Irani to provide goal orders for goals from both conjunctive planning goals and from outstanding pre-conditions in partially developed plans.

### Control Rules for Goal Order

STATIC [20] is a compilation method that generates control rules, a priori, direct from the model of a planning domain. In this section we consider the control rules it generates that can be used to order goals during plan generation.

It compiles control rules for goal ordering by analysing graphs (PSG's) that represent the relationships between goals and operators in the domain model. These are then used to identify necessary interactions between goals. So like PRECEDE, STATIC identifies necessary interactions between goals and then orders goals on the basis of these interactions. However there are a number of important differences between the approaches.

The first difference between STATIC and PRECEDE is to do with the necessary interactions that goal orders are based upon. STATIC analyses the PSG's it has constructed, to see if achieving some goal $g$ will necessarily clobber another goal $g'$. If this is the case then a control rule is formed so that the goals $g$ and $g'$ are ordered with respect to each other so that $g$ is achieved first. PRECEDE differs from this because interactions between pairs of goals $(x, y)$ are based on the *necessary_before*$(x, y)$ condition, which requires that $x$ necessarily clobbers $y$ and $y$ necessarily clobbers $x$ (defined in section 4.2.1).

Both compilation methods analyse binary goal interactions rather than n-ary ones because this helps curtail the cost of acquiring goal orders. However, STATIC compiles control rules that order pairs of goals with respect to each other whereas PRECEDE goal orders are an accumulation of necessary orders between pairs of predicates. As we saw in section 4.2.3 this means that PRECEDE goal orders can order larger sets of predicates with respect to each other than just the pairs of predicates that are ordered by STATIC.

### 8.1.2 PROPOSE: relation to other work

In this section we compare PROPOSE with some of the related work in this area that we discussed earlier in section 3. These are compilation methods that compile a modelled planning domain to help select operator and bindings during partial plan specialisation. The different areas that we subdivide the work into are static processing of control rules [20] and macro operator construction (such as: [67] and [65]).

#### Control Rules for Operator Choice

In this section we consider the acquisition of control rules for operator and binding selection in STATIC [20]. We'll start by comparing the actual method of compilation and then compare the output of the compilation.

Both methods begin by generating a graph that represents relationships between goals and operators in the problem space. For STATIC, each graph is rooted in a distinct domain predicate and is generated by symbolic backward chaining on the operators for the problem space, starting from the predicate at the root of the graph. The root node is connected via or-links to all operators whose effects match the literal and each operator is linked via and-links to goal nodes that are its pre-conditions. The graph is directed and acyclic, termination of the graph is guaranteed since predicates are only permitted to occur once in the graph.

PROPOSE graphs are not rooted, and instead of generating a graph for each predicate in the domain it generates a graph for each *sort* in the domain. We would expect that the number of sorts in a modelled planning domain will frequently be less than the number of predicates in the domain. For example, in the warehouse planning world $WW$, there are 14 achievable predicates in the domain model (the set *preds* defined on page 15) but there are only 6 different sorts.

Also, nodes in the PROPOSE graph, for some sort $s$, are members of the positive state invariant $IP_s$, so rather than single predicates at each node, as is the case for STATICS PSGs', there can be sets of predicates at each node. One consequence of this is that PROPOSE graphs may contain fewer nodes. Arcs in a PROPOSE graph, of sort $s$, are labelled with operator names and the node at the tail of an arc contains the operators pre-conditions, of sort $s$, and the node at the head of an arc contains the operators post-conditions, of sort $s$. The graph is directed and cyclic. Termination of construction of the graph is guaranteed because the procedure is non recursive and each operator in the set is only processed once.

We contend that PROPOSE generates less graphs than STATIC because graphs are generated for each sort in the domain rather than each goal predicate. In addition each of the PROPOSE graphs may be smaller (have fewer nodes) than STATIC's PSG's since nodes in PROPOSE graphs are sets of predicates rather than single predicates.

Now lets consider the differences between the output of the two compilation methods. STATIC compiles control rules for operator selection or rejection from graph nodes where the outcome of using a particular operator in the context of solving a particular goal is known. For example, a rejection rule is acquired whenever the addition of an operator results in a "goal-stack cycle", that is, the node has already appeared in the context of achieving the goal.

What PROPOSE does is to traverse the graphs, assembling iterative macros, which are paths through the graph for different pairs of situations that an object of some sort can occupy. PROPOSE will consider goal-stack cycles during traversal of the graphs. They are rejected if they fail the loop invariant, in other words if they aren't effective (see page 70) otherwise they will be incorporated into the macro as iterative components. Hence a key difference between the methods is that PROPOSE can recognise iterative sequences of operators that will be useful in the context of certain task configurations.

### Macro Operator Acquisition

PROPOSE compiles iterative macro operators for the set of different task configurations for some modelled planning domain. In this section our main purpose is to compare the iterative macros that are compiled by PROPOSE with macros that are compiled by other methods.

An important difference between PROPOSE and other methods for macro operator acquisition, such as experience based learning in [67], is that they rely on example planning problems to compile the macro operators. This means that they can be unduly biased towards particular problem distributions, namely the ones that the example problems came from. PROPOSE is an exception to this, since it can compile useful iterative macro operators direct from the domain model and the use of sort abstraction means that macro's can be compiled for each sort of object in the modelled planning domain.

How far do PROPOSE iterative macros compare to the macros compiled by other methods such as FERMI [67] and BAGGER2 [65] ? In figure 8.1 below, which is

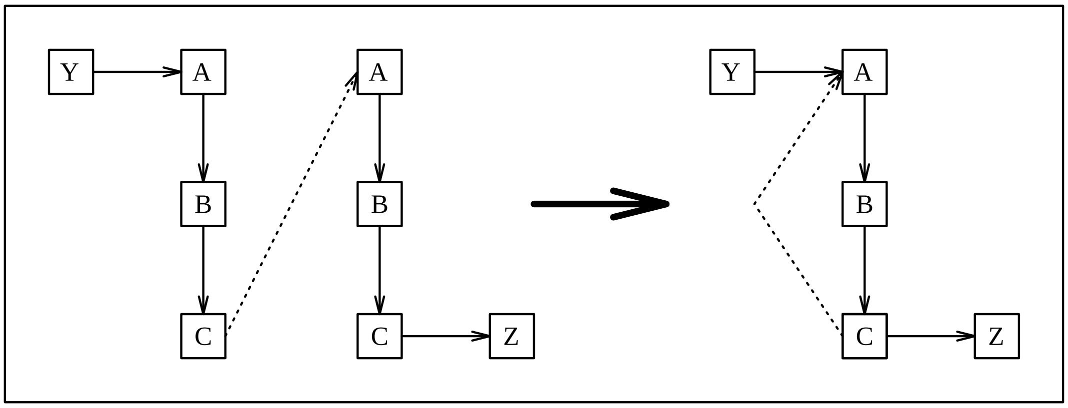


Figure 8.1: Formation of Iterative Macros

adapted from [67], the formation of iterative macro operators is shown. In the figure the nodes (labelled $A$, $B$, $C$ and so on) represent operators that appear in a solution plan for some planning problem. The directed arcs show the relative order of the operators in the plan, so for example, operator $Y$ comes before $A$ in the sequence of operators shown in the left hand part of the figure. This diagram shows how operators that can be repeated an arbitrary number of times in a plan (denoted by the dotted line) can be composed to form an iterative macro. So in this case, the sequence of operators $[A, B, C]$ can be repeated an arbitrary number of times. This is precisely the form of iterative macro that PROPOSE outputs, where iterative components within a macro can repeated an arbitrary number of times. Hence, we contend that PROPOSEs' iterative macro operators do correspond to those that appear in the machine learning literature.

In most machine learning techniques, the problem in composing iterative macros is how to generalise from a linear planning trace which contains repeated operator sequences, to an arbitrary number of applications of the repeated sequence of operators. This problem is frequently referred to as "generalising-to-n" [66]. PROPOSE provides a general solution to this problem, for sequences of operators that can be repeatedly applied to the same object of some sort, through analysis of the domain model prior to planning.

### Summary

One of the main differences between the compilation methods that we've developed and most of the compilation methods that were mentioned in this section is that PRECEDE and PROPOSE are used to compile a domain model off-line. For example, compilation of abstraction hierarchies with ALPINE [36] can be seen as on-line since the compilation is planning problem dependent and hence relies on current planning experience. Other methods such as macro operator acquisition (for example [67]) can be seen as post-event since they require the output of training planning sessions. A useful consequence of the planning problem independence of off-line compilation methods such as PRECEDE and PROPOSE is that they avoid bias towards particular problem distributions.

Another consequence of the problem independence of PRECEDE and PROPOSE is that the compilation method need only be applied to a domain model once. This is in contrast to other methods (see for example [33] and [36]) where the domain model is compiled separately for every new planning problem that is encountered. Although the cost of this problem specific compilation may be small, especially when compared with the overall cost of plan generation, the compilation nevertheless still consumes on-line resources.

Finally, an interesting feature shared by PRECEDE, PROPOSE and many of the compilation methods that we have considered is that they make use of information about inconsistent states in a modelled planning domain during compilation. Examples of compilation methods that use inconsistency constraints include [33], [21], [14] and [19]. Some critics have criticised the use of such inconsistency constraints, for example Tadepalli and Isukapalli [71] argue that to expect a domain modeller to supply, what they call "domain specific simplification rules" (similar to the pairs of predicates found in the negative state invariant used by PRECEDE), is requiring too much. However, we feel that their use is acceptable since constraints for limited sets of predicates can be automatically generated from the set of operators for a modelled planning domain, as was the case for the pairs of predicates in $IN$ for the planning world $SW$ (see appendix A.3). In addition, supplying a state invariant is analogous to the situation in formal specification of software systems and as such is as easy (or as hard) as modelling operators in terms of pre- and post-conditions.

## 8.2 Scope of the Compilation Methods

In this section we turn our attention to the scope of the novel compilation methods that we have developed. In the course of the discussion we shall consider both the methods of compilation and the way that we extended $MEA$, a planner in the class of DIP's, so that it could reason about compiled domain models. We start in section 8.2.1 by looking at PRECEDE domain model compilation and its use during planning and then in section 8.2.2 we consider PROPOSE.

### 8.2.1 Scope of PRECEDE

The specific goal that motivated work on PRECEDE was to develop an off-line method of compiling goal orders that are useful in improving planning performance. As we saw earlier (section 4.2.3) the output of PRECEDE compilation, for some modelled planning domain, is a partial order, $P$, over predicate goals in the domain model. A PRECEDE partial order $P$ is an accumulation of pairs of predicates $(x, y)$ for which $x$ must necessarily be achieved before $y$ or in other words where the following condition is true.

$$necessary\_before(x, y)$$

The partial order $P$ is transitive and this means that if both of the conditions below are true:

$$necessary\_before(a, b)$$
$$necessary\_before(b, c)$$

then $a$ is heuristically ordered to be achieved before $c$. So although the orders in $P$ are based on necessary interactions between pairs of predicates, the orders that are built up in $P$ are heuristic. When the PRECEDE compiled domain models were used in a series of empirical tests, the results showed that PRECEDE goal orders can improve planning performance and compare favourably to planning performance with ABGEN.

We argued earlier that a useful consequence of this method of compiling goal orders was that it avoided the specific collapsing problem that methods for automatically generated abstraction hierarchies, such as ALPINE [36] are susceptible to. It is our contention that PRECEDE overcomes this problem because it is based on necessary interactions between goals and as a consequence it can generate useful goal orders for domain models that ALPINE cannot generate useful abstractions for. However, it may be that there are modelled planning domains for which ALPINE can find useful hierarchies but where PRECEDE cannot. For example, we would not expect PRECEDE to generate useful goal orders for a domain model where all operators in $OS$ had single pre-conditions since PRECEDE orders are based on interactions *between* pre-conditions. Although a domain model of this type is of limited interest since the planning problem has been shown to be polynomial if operators are restricted to one pre-condition and where the number of goals is bounded by some constant [6]. Another situation in which ALPINE might be able to generate useful abstraction hierarchies when PRECEDE can't, is for a domain model where there are only possible interactions between pairs of predicates.

As mentioned earlier, pairs of predicate goals are added to a PRECEDE partial order $P$ on the basis of necessary interactions between them and their accumulation into the global partial order is heuristic. One consequence of this, is that in certain situations the precise relationship between a pair of predicates is unknown. As an illustration of this, lets suppose that for some modelled planning domain the following are true:

$$necessary\_before(a, b)$$
$$necessary\_before(b, c)$$

which means that whenever $a$ and $b$ must be achieved in the same state then $a$ must be achieved before $b$ and whenever $b$ and $c$ must be achieved in the same state then $b$ must be achieved before $c$. The accumulation of these *necessary_before* orders into the partial order $P$, because the relation is transitive, implies that $a$ can be achieved before $c$. Note the use of the word *can* to show that this order is heuristic, because in certain situations the actual order between $a$ and $c$ is unknown. In fact there are three cases to consider when $a$ and $c$ are ordered :

- when ordering a set of predicates that contain $a$, $b$ and $c$ then the presence of $b$ means that the order between $a$ and $c$ is $necessary\_before(a, c)$
- when ordering a set of predicates that contain $a$ and $c$ but not $b$ if $b$ implies $c$ then the order between $a$ and $c$ is $necessary\_before(a, c)$
- when ordering a set of predicates that contain $a$ and $c$ but not $b$ and where $b$ does not imply $c$ then the relationship between $a$ and $c$ is unknown

So, it is only in the third case that the heuristic accumulation of necessary before orders may incorrectly order predicate goals. However, there were two reasons why we decided to keep the PRECEDE partial orders in $P$ heuristic. The first of these is connected with the example planning system, $MEA$, that we selected to test the utility of PRECEDE compilation. As we saw in section 6.1, the strategy of this planner is to maintain partially developed plans as totally ordered sets of operators and during partial plan specialisation only the outstanding pre-conditions of the "earliest" operator in the partial plan are considered for goal achievement. So whenever $P$ is used to select a goal to achieve first during partial plan specialisation the only predicates that are considered are those that appear as the pre-conditions of the earliest operator. In this situation the heuristic orders in $P$ will not be considered. In fact, the only time when $MEA$ uses the heuristic orders in $P$ to order goals is with the set of goals that are input as part of a conjunctive goal planning problem.

The second reason why the orders in $P$ are heuristic is that this reduces the amount of compilation that is performed. This is because the only pairs of predicates that need be considered are those that appear together in operator pre-conditions.

However, a useful extension of PRECEDE would be to test all pairs of predicates $(x, y)$ in the domain model using the condition $necessary\_before(x, y)$. This extension may be useful when testing the impact of PRECEDE on other planning systems. It would ensure that all necessary orderings between pairs of predicates in the domain were known and were not heuristic. A consequence of this would be that the transitivity of $P$ would no longer be required since all pairs of ordered predicates would have been explicitly added to the relation. In fact, not all pairs of predicates need be tested for *necessary_before* since all elements in $IN$ can be ignored since we know, from the operational definition of $IN$ (page 47), that they can not be achieved in the same state. For all other pairs of predicates, we know that they can be achieved in the same state and either there is a necessary interaction between them in which case they are ordered with respect to each other or there is no necessary interaction between them and they are left unordered.

It appears that extending PRECEDE so that all pairs of predicates are tested for necessary interactions is feasible since the complexity of testing every set of predicates in a domain model is given below (where $N$ is the number of predicates and $S$ is the average number of argument "slots" per predicate)

$$O(N \times (N-1) \times 2^S)$$

This is exponential with respect to the average number of slots per predicate but this is acceptable for domain models where $S$ is small, as is the case for predicates in the example modelled planning domains that we have used.

Apart from this change to PRECEDE, so that all pairs of predicates in a domain are tested using *necessary_before*, there are other ways in which we could extend PRECEDE which involve integrating it with other approaches. For example, PRECEDE could be integrated with the method for partial plan selection based on state violation detection that features in [14] and [19].

It was noted earlier that the sets of "incompatible assertions" that are compiled by REFLECT [14] are very similar to the negative state invariant $IN$ that is used by PRECEDE during compilation. Indeed, the operational definition of $IN$ and the implementation that was based on this were inspired by the method that was outlined by Dawson and Siklóssy [14]. REFLECT selects partial plans whose outstanding pre-conditions describe consistent states of the modelled planning domain. This could be easily integrated with the use of PRECEDE goal orders during planning so that the state invariant $IN$ is first used for compiling the goal order $P$ and then $IN$ could also be used for selecting partial plans whose outstanding pre-conditions are consistent.

PRECEDE could also be combined with the approach to goal ordering of Cheng and Irani [33] since it would complement their approach which fails to detect certain necessary interactions between goals. We'll illustrate this with an example from the warehouse planning world using GOAL which is our implementation of the algorithms presented in [33] (see for example [50]). For a planning problem with the following goal condition:

$$G = \{on\_floor(crate1, s6), driver\_at(s9), at(crate2, s8)\}$$

GOAL order the goals in $G$ as follows:

$$[on\_floor(crate1, s6), \{driver\_at(s9), at(crate2, s8)\}]$$

where $on\_floor(crate1, s6)$ is first in the order and the other two goals are unordered with respect to each other. However, PRECEDE could also detect the following order:

$$necessary\_before(at(crate2, s8), driver\_at(s9))$$

because the pre-conditions of all achiever operators of $at(crate2, s8)$ require that the *truck* and *driver* are both at the same location adjacent to $s8$ (either $s4$, $s7$ or $s12$) which is inconsistent with the goal $driver\_at(s9)$.

The reason why GOAL fails to detect the necessary interaction between $at(crate2, s8)$ and $driver\_at(s9)$ is because it only considers bindings of operators which are *consistent* with all of the goals in $G$. This means that GOAL rejects all the achievers of $at(crate2, s8)$ because they are not consistent with the other goals in $G$. For example, the pre-conditions of $unload(c2, s7, s8)$ include $driver\_(s7)$ which is inconsistent with $driver\_at(s9)$ from $G$. So for the example goal $G$, GOAL does not consider any of the achievers of $at(crate2, s8)$ and hence fails to detect the necessary ordering between $driver\_at(s9)$ and $at(crate2, s8)$. We would expect that PRECEDE and GOAL could be combined and that PRECEDE would complement GOAL by identifying necessary interactions overlooked by GOAL.

### 8.2.2 Scope of PROPOSE

The output of PROPOSE compilation is a set of iterative macro operators for sorts in a modelled planning domain, called $\mathcal{M}$ and a set of enumerated sequences, called $\mathcal{EM}$, for unwinding the iterative parts of the macros. We feel that this approach to compilation has great potential and that our work so far has just begun to exploit the power of the approach. In this section we look at the scope of both PROPOSE compilation and the way that the PROPOSE compiled domain models are used during planning. We shall also consider extensions to the method where appropriate.

We start by looking at the form of the iterative macros and their correspondence to other example iterative macros that appear in the machine learning literature. One method of acquiring iterative macros is through experience based learning [65, 67]. The iterative parts of the macros are generated by generalising from sequences of operators that are repeated in example planning traces, this is the problem of "generalisation-to-n" [66].

Shavlik [65] observes that repetition of a sequence of operators is not sufficient for generalisation of the number of repetitions, there must also be a recursive structural pattern where each application of a sequence of operators achieves the pre-conditions of the next application of the sequence. From the invariant on "loops" in PROPOSE macros (see page 70) we know that PROPOSE iterative macros, of sort $s$, must satisfy their own pre-conditions, of sort $s$ and hence they appear to satisfy Shavliks' recursive structural pattern. For example, in the two agent planning world $SW^2$ for a task configuration of sort *agent* where the initial situation $i$ and the goal situation $g$ are as follows:

$$i = \{at(A, D, R), in(A, R)\}$$
$$g = \{at(A, D', R'), in(A', R')\}$$

PROPOSE identifies the following macros for this task configuration:

$$\overline{M}_{agent}(i, g) = \{[[gothrudoor(A, D, R_1), gotodoor(A, D_1, R_1)]^*], \ldots\}$$

where the set of macros includes the iterative macro that consists of a repeated sequence of the operators $gothrudoor(A, D, R_1)$ and $gotodoor(A, D_1, R_1)$. These operators repeatedly move the *robot* (denoted by variable $A$) from room to room. We can see that this repeated sequence of operators does satisfy its own pre-conditions, of sort $s$ because the predicate named *at_door* is a pre-condition of $gothrudoor(A, D, R_1)$ and is a post-condition of $gotodoor(A, D_1, R_1)$. In both cases, the second argument of the predicate named *at_door* is of sort *door* but it will be bound to some different object of sort *door* so it is "effective" (this condition avoids repeated sequences that achieve nothing).

The scope of PROPOSE iterative macros is limited to identifying sequences of operators that are repeatedly applied to the *same* object of some sort $s$, as is the situation in the $SW^2$ example given earlier. This means that there are sequences of operators that might be repeated in a plan but that PROPOSE will not identify. For example, consider a block stacking world where a robot arm can stack any number of blocks to form a tower[1]. Suppose that in this domain the sorts of objects include *block* and *arm*. So PROPOSE would compile macros for sort *block* that involve changing the situation of a *block* from on the table to stacked. However, no iterative macros would be identified because there are no sequences that are repeatedly applied to the same object of sort *block*. Rather the iterative sequence of actions are applied repeatedly to different objects of sort *block*, when a tower is created.

So the scope of PROPOSE iterative macros makes it unable to identify possible sequences of operators that can be applied to *different* objects of sort $s$ to achieve some conjunction of goals of sort $s$. Extending PROPOSE so that it can identify repeated sequences of actions that are applied to different objects of some sort is an area which we are keen to explore in future work.

Another issue is how far PROPOSEs' iterative macros compare to disjunctive macros which have been reported in the machine learning literature. We saw (page 37) that

[1]We use this example world because there is no comparable situation in the example planning worlds $SW$, $SW^2$ and $WW$.

systems such as FERMI [67] can generate iterative and disjunctive macro operators. For example, the following two macros, which are represented in the PROPOSE format where $a$ is the earliest operator in the macro and $f$ is the last:

$[a, b, c, d, e, f]$
$[a, b, x, d, y, f]$

can be combined to form a single macro as follows:

$[a, b, (c, x), d, (e, y), f]$

where pairs such as $(c, x)$ in the body of a macro represent a disjunction, in this case either operator $c$ or $x$ can be applied. A consequence of this is that where there were two macros originally, after they have been merged into a single disjunctive macro this describes the following four macros:

$[a, b, c, d, e, f]$
$[a, b, x, d, e, f]$
$[a, b, c, d, y, f]$
$[a, b, x, d, y, f]$

If PROPOSE had been used to compile this example planning domain then all four alternative macros (corresponding to the two two disjunctive macros) would have been enumerated. However, it may be that storing macros in this disjunctive form is more efficient than the linear form in which PROPOSE represents them. In addition, it may be the case that efficiency can be increased by storing the enumerated sequences that are compiled by PROPOSE in a disjunctive form. For example, in the warehouse world $WW$ for the sort *space* the following enumerated sequences are compiled:

$$\begin{aligned}
&\overline{EM}_{space}(s3, s13) = \{\\
&\quad [s3, s2, s6, s10, s14, s13],\\
&\quad [s3, s2, s6, s10, s9, s13],\\
&\quad [s3, s7, s6, s10, s14, s13],\\
&\quad [s3, s7, s6, s10, s9, s13]\}
\end{aligned}$$

which could be re-represented as :

$$\overline{EM}_{space}(s3, s13) = \{[s3, (s2, s7), s6, s10, (s9, s14), s13]\}$$

where the notation $(a, b)$ is used to represent a disjunction within the enumerated sequence. The effect of re-representing PROPOSE macros and enumerated sequences in disjunctive rather than linear form is another area that we are keen to explore.

Finally, lets consider the scope of the way that we have amended the planner *MEA* to use the PROPOSE compiled domain model. There are two main changes to the *MEA* algorithm. The first is that when bound with a PROPOSE compiled domain model, *MEA* uses iterative macros for task configurations of objects in the input planning problem goal when selecting operators during partial plan specialisation. The second change is that when an iterative component is selected from a PROPOSE macro, relevant enumerated sequences are used to "unwind" the iterative part of the macro. There are a number of ways that this implementation could be expanded, which include:

- PROPOSE iterative macros and enumerated sequences could be used to select and instantiate operators for goals other than those in the input top-level planning problem. For example, in the warehouse world $WW$, for a planning problem with goal and initial state as follows:

  $$G = \{on\_floor(crate1, s16), \ldots\}$$
  $$I = \{truck\_loaded(crate1, s13), \ldots, driver\_on\_floor(s4), \ldots\}$$

  the task configuration for *crate*1 has the *crate* initially loaded on a truck and in the goal it is on the floor. The macros for this task configuration include:

  $$[[drive\_load(C, S, S_1)]^*, unload(C, S_1, S_2)]$$

  This macro could then be used to achieve the desired situation for *crate*1 via a sequence (0, 1 or more) of $drive\_load(C, S, S_1)$ followed by $unload(C, S_1, S_2)$. The pre-conditions of $drive\_load(C, S, S_1)$ include *in_truck* and $driver\_at(S)$ which are not achieved by the initial state $I$ given above. To achieve these pre-conditions we have a task configuration of sort *driver* where the initial situation is that the driver is on the floor and the goal situation that it is in the *truck* at the desired location. Hence the iterative macro given below could be used to achieve this pre-condition goal.

  $$[[walk(S, S_1)]^*, get\_in(S_1, S_2)]$$

- PROPOSE iterative macros can be seen as abstract plans which contain all the operators required to change the situation of an object of some sort from an initial situation to some goal situation. However, there is no provision for any other sorts of objects that interact with this abstract plan. The assumption in the implementation is that given a task configuration for some sort $s$ then the PROPOSE macros for that task can be used directly to generate the plan. This implicitly assumes that all pre-conditions of operators in the macro that are not of sort $s$ are achieved from the initial state. For example, the iterative macro to change the situation of *truck* from at some *space* to another *space* is $[[drive(S, S')]^*]$ and if the initial space is $s1$ and the goal space is $s9$ then the sequence $[s1, s2, s6, s10, s9]$ could be used to unwind the sequence as follows:

  $$[drive(s1, s2), drive(s2, s6), drive(s6, s9), drive(s9, s10)]$$

  however, if the initial situation of *crate*2 was $on\_floor(crate2, s6)$ this would cause problems because the pre-conditions of $drive(s2, s6)$ that aren't of sort *truck* include $clear(s6)$ which is false from the initial state (since it is inconsistent with $on\_floor(crate2, s6)$). In this case the implementation of *MEA* would reject the PROPOSE macros and revert to its uncompiled method of operator selection.

In our future work we are keen to expand the way that *MEA* uses PROPOSE compiled domain models. A potential way of doing this comes from the work in *plan merging* of Yang and colleagues [77]. Their approach is to generate plans for individual goals in a conjunctive goal problem ignoring how each plan might interact with the other goals. Then the individual plans are merged together, handling any interactions while the plans are merged. They show that when certain restrictions hold, plans can be merged optimally. It may be that this method of plan merging will be useful when merging macros for different task configurations and different sorts. Clearly, PROPOSE could also be useful in improving the performance of the planner used in [77] that generates plans for the individual goals in the conjunctive planning problem.

## 8.3 Summary

The key difference between the compilation methods that we have developed and most of the other methods that we reviewed is that compilation with PRECEDE and PROPOSE is off-line whereas other methods can be seen as on-line or post-event. There are only a few examples of off-line compilation methods in the literature, as found in the work of Dawson and Siklóssy [14] and Etzioni [20, 21]. We contend that PRECEDE and PROPOSE are an important contribution to the move away from post-event and on-line compilation towards off-line compilation as a way to improve the performance of generative planning systems.

There are a number of ways in which the scope of the compilation methods (and their use during plan generation by a DIP) could be extended. For example, PRECEDE compilation could be extended so that all necessary interactions between pairs of predicates are considered during compilation. PROPOSE could be extended so that it compiles iterative macros for sets of objects of a sort rather than individual objects of a sort. We contend that widening the scope of PRECEDE and PROPOSE could further extend the considerable power of the compilation methods.

# Chapter 9

# Conclusions

This thesis has addressed the problem of improving the performance of generative domain independent planning systems. A common way of tackling this problem has been to embed the planning system with learning components that use prior or current planning experience to improve subsequent performance. However, we have argued for a move away from these post-event and on-line approaches towards what we call off-line compilation. We contend that for classical planners that reason about complete domain models (contain all information needed for plan generation) everything that can be compiled post-event or on-line can be compiled off-line direct from the domain model. In support of this thesis we have specified, designed, implemented and empirically evaluated two complementary methods of off-line compilation, which we call PRECEDE and PROPOSE.

PRECEDE was the subject of chapter 4 of this thesis. It is a method for generating goal orders direct from the model of a planning domain. The goal orders are based on necessary interactions between predicates in the modelled planning domain and can be used during partial plan specialisation to select which goal to achieve next. During compilation, the method uses information about inconsistent states of a modelled planning domain which is supplied, either by the domain modeller or automatically generated from the set of domain operators, in the form of a negative state invariant. It is negative because it specifies pairs of goals that cannot be achieved in the same state.

The other compilation method that we created in support of this thesis is called PROPOSE and it was the subject of chapter 5. It is a method for compiling iterative macro operators that can be used to select achiever operators during partial plan specialisation. The method uses information about consistent states of a modelled planning domain which is supplied in the form of a set of positive state invariants. It also uses the *sort* of objects in a domain model as a means of abstraction during compilation. Each sort is analysed by considering the possible situations that objects of that sort can occupy. The iterative macro-operators that are generated form abstract solutions which span all of the possible start and end situations for the sorts of objects in the domain model. PROPOSE also generates sequences of constants that can be used to unwind the iterative parts of the macros.

We evaluated these compilation methods in three ways, the first of which was by static evaluation of the output of the compilation. In section 4.3 we compared example goal orders compiled by PRECEDE with abstraction hierarchies that were generated by ALPINE [37], an established method of on-line compilation. We were able to prove that there is a correspondence between the goal orders generated by PRECEDE and abstraction hierarchies generated by ALPINE. This means that if, during compilation, PRECEDE identifies that $necessary\_before(x, y)$ is true for some $x$ and $y$, then $y$ can never be in a higher ALPINE abstraction level than $x$. We also argued that PRECEDE's goal orders are not susceptible to the same sort of collapsing problem as ALPINE abstraction hierarchies because they are based on necessary interactions between goals whereas ALPINE's are based on possible interactions between goals. This analysis led us to suppose that the amount of performance improvement with PRECEDE would compare favourably to that with ALPINE and that in domains for which ALPINE cannot generate useful hierarchies, PRECEDE might be able to generate useful goal orders.

In section 5.6 the iterative macros and enumerated sequences generated by PROPOSE were statically evaluated. We considered the potential way in which PROPOSE compilation might affect search and we also looked at the correspondence between PROPOSE macros and some example solution plans. Finally, we considered the relation between the output of PROPOSE compilation and the size of a domain model. The evaluation led us to suppose that the macros and enumerated sequences could reduce search during planning and that the size of the output of compilation would be acceptable, at least for the type of example domains that we are using. Hence we concluded that the overall effect of planning with a PROPOSE compiled domain model would be to improve planning.

The second part of the evaluation of our work was empirical testing of the effect of PRECEDE and PROPOSE compilation on the performance of an example planning system. This evaluation was the subject of chapters 6 and 7. We selected *MEA*, a fairly representative generative domain independent planner, to use in the empirical evaluation. We formalised the plan generation algorithm at the heart of *MEA* and then extended its input language so that it accepted PRECEDE and PROPOSE compiled domain models. For PRECEDE this meant changing *MEA* so that it used the PRECEDE goal orders to select outstanding goals to achieve next during partial plan specialisation. For PROPOSE, *MEA* was changed so that it used the iterative macros and enumerated sequences while selecting and instantiating achiever operators during partial plan specialisation.

A series of tests were carried out in a range of example planning domains and the test results were discussed in chapter 7. The tests showed that PRECEDE and PROPOSE, both individually and in combination, can improve performance of a generative domain independent planner. In addition, the amount of performance improvement when planning with a PRECEDE compiled domain model compared favourably with ALPINE [37], an established method of on-line learning. Also the amount of performance improvement with PROPOSE compared favourably with CHAR, an established method of post-event learning [46, 50].

The third way in which PRECEDE and PROPOSE were evaluated was in terms of their relation to other work in the literature. This evaluation was the subject of chapter 8. PRECEDE was in part motivated by on-line approaches that reject partial plans with inconsistent outstanding pre-conditions [14, 19], but it differs from them because its compilation is completely off-line and any inconsistent states that PRECEDE detects are used to order goals rather than reject partial plans. The nearest comparable method to PRECEDE is ALPINE [37] but PRECEDE is different because all compilation is completely off-line and its goal orders are based on necessary interactions between goals whereas ALPINES are based on possible interactions.

The important difference between PROPOSE and other methods that output iterative macros for planning (for example [67], [65]) is that PROPOSE compiles macros and enumerated sequences direct from the domain model without the need for training problems. In addition, the use of sort abstraction means that iterative macros can be compiled for all the sorts of objects within the domain model.

On the basis of the work that we have presented we conclude that PRECEDE and PROPOSE are novel, off-line, complementary compilation methods that can be useful in improving the performance of domain independent planners. Hence we feel that our work forms an important contribution to the shift of emphasis towards off-line compilation as a way of improving the performance of these planning systems.

## 9.1 Future Work

Of the work that we have presented in support of this thesis there are a number of aspects that we intend to develop further in our future work. Some of these were mentioned earlier in section 8.2 (page 126) when we discussed off-line compilation with PRECEDE and PROPOSE and the use of compiled domain models in planning. Here we focus on some of the more general aspects of our work in off-line compilation and how it can be extended.

The first of these is to demonstrate the further generality of the off-line compilation methods that we have created. It is our contention that PRECEDE and PROPOSE will be useful for improving the performance of other planners in the class of DIP's that we are interested in. In our work to date we have demonstrated their effect on *MEA*, a fairly representative DIP. So we are keen to explore the effect of PRECEDE and PROPOSE on other DIP's. An obvious extension to our work is to test the effect of the compilation methods with the planner *NLP*, which is a non-linear planner in the tradition of TWEAK [9]. This plan generation engine forms part of the *FM* planning and learning system [47, 46], as does *MEA*, so it should be possible to compare the effects of PRECEDE and PROPOSE compilation (and established on-line and post-event methods) between *MEA* and *NLP*.

Also, in future work we plan to generalise off-line compilation with PRECEDE so that all pairs of predicates within a domain model are tested using the *necessary_before* condition (as discussed in section 8.2.1). Hence, we plan to test the effect of compilation with PRECEDE and PROPOSE on the plan generation algorithm *NLP* and to compare the effect of the generalised PRECEDE with both of the plan generation algorithms *MEA* and *NLP*.

Another useful area of future work is to explore ways of automatically generating state invariants that can be used during compilation and planning. We have already begun work in this area, by developing a procedure for automatically generating the negative state invariant $IN$ that is used by PRECEDE. The $IN$ for the robot planning world $SW$ was automatically generated using this procedure (see appendix A.3). As we discussed in section 4.2.1 there are advantages from both automatically generated state invariants and from domain modeller supplied state invariants since in both cases they can throw up inconsistencies in the way that a planning domain has been modelled. Eventually we would like to develop a unified state invariant that could be used during compilation by both PRECEDE and PROPOSE. It could also be used during compilation to check the consistency of sets of domain operators and input planning problems.

We also plan to extend our work so that we can test our ideas on "real" application domains. One example planning domain is the management of traffic flow in the domain of air traffic control (ATC). This planning domain consists of a region of airspace which can be divided into smaller spaces known as sectors, within which may be radio beacons. Typical planning goals require routing aircraft through the airspace and assigning slots for take off, so that the flow of aircraft though different beacons and sectors is smoothed out over a given period of time. We argued elsewhere (see [49]) that generative domain independent planning systems satisfy some of the key requirements of this kind of planning task. For example, the planning is generative since when a route is planned for an aircraft it is assumed that all knowledge of the domain is complete and then the entire plan, for a certain "look ahead" time, is generated. The planner cannot be reactive since it is not acceptable for an aircraft to fly to a beacon, assess the situation and then wait until the remainder of the route is planned. Obviously, there is uncertainty in the domain since it can change in unpredicted ways but routes are generated on the assumption that the current "snap shot" of the domain is complete and correct. If the domain does change in an unpredicted way then the plan must be re-generated. We have conducted pilot experiments with a toy ATC planning domain which suggest that PRECEDE and PROPOSE compilation can improve planning performance in these domains. As an illustration, suppose that the ATC domain consists of *sectors* and *beacons* as follows:

$$sectors = \{s1, s2, s3\}$$
$$beacons = \{b1_a, b1_b, \ldots, b3_z\}$$

where the number of the beacon denotes that it is in a particular *sector* (for example $b1_a$ is in $s1$ and $b3_z$ is in $s3$). If the different situations of *aircraft* include $in\_sector(Ac, S)$ and $over\_beacon(Ac, B, S)$ then PRECEDE might identify orders as follows:

$$necessary\_before(in\_sector(Ac, S), over\_beacon(Ac, B, S))$$

where $Ac$ is of sort *aircraft*, $S$ is of sort *sector* and $B$ is of sort *beacon*. The order shows that during partial plan specialisation $in\_sector(Ac, S)$ must be achieved before $over\_beacon(Ac, B, S)$. As an illustration of PROPOSE compilation, suppose that *aircraft* is a sort in this domain model and that for a task configuration where the initial situation of some *aircraft* is $level\_flight(Ac, B, B', S)$ and the goal situation is $over\_beacon(Ac, B, S)$, the iterative macros include:

$$[[fly\_level(Ac, B, B', S)]^*]$$

Hence, the goal situation can be achieved by a series of suitably bound level flight operators. This iterative component could be unwound using enumerated sequences of values that have been generated for the appropriate start and end *beacons*, since *beacon* is the static sort which is changed by this iterative macro part.

## 9.2 Discussion

This thesis has addressed the problem of inefficiency of generative domain independent planning systems. We have argued for a move away from on-line and post-event learning towards off-line compilation as a way of improving the efficiency of these systems. In support of this argument we have created novel off-line compilation methods that compared favourably with established on-line and post-event compilation methods.

Although we have focussed on the problems of domain independent planners our thesis work can be seen in the more general context of problem reformulation. This has long been seen as a useful problem solving strategy and since the early work of Amarel [1] through the work of Korf [39] and more recently Knoblock [37], researchers have testified to the power of reformulation.

Problem reformulation is also the essence of many approaches to automatic program generation and we see a correspondence between approaches to automatic program generation and our approach to off-line compilation. For example, Lowry [44, 43] presents a framework for problem reformulation and algorithm synthesis. Within this framework specifications of problem domains are seen as theories. Algorithms for the problem domain are also seen as theories about state changes in the problem domain. Within this framework reformulation is seen as a representation map between the theories. So the process of automatic program generation can be seen as a series of mappings from specification to algorithm. There is a parallel here with domain model compilation for planning. Our starting point is a specification of some planning domain, the domain model, and the output of compilation is some re-representation of the domain model achieved by a series of mappings.

# Appendices

# Appendix A

# *SW*: The Robot World

The robot planning world *SW* is a model of a seven room, ten door planning domain, with a robot capable of moving boxes and opening doors which was originally defined by McCluskey [46]. Also, it is a variation of the planning domain used in [62]. The figure below shows a state of this world, where for example the robot is in *room*7, *box*1 is in *room*2, *door*47 (the door joining *room*4 and *room*7) is closed and *door*45 is open.

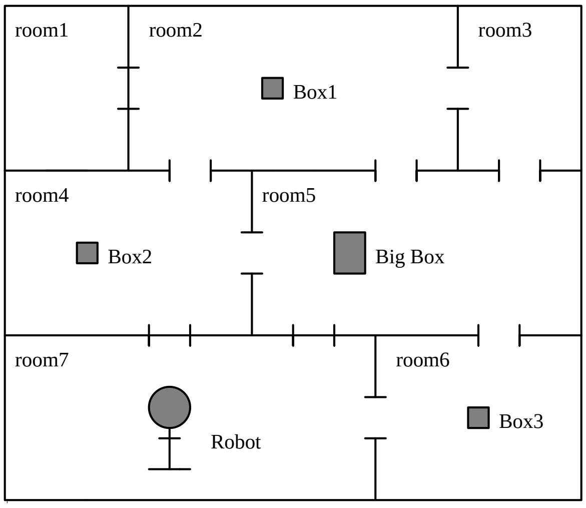


Figure A.1: *SW*: The Robot World

The state of the world that is shown in the figure is represented in the FM input language as follows:

$$\{robot\_in(room7), in\_room(box1, room2), in\_room(box2, room4), in\_room(box3, room6), in\_room(big\_box, room5), closed(door47), closed(door12), closed(door57), open(door67), open(door56), open(door45), open(door35), open(door25), open(door24), open(door23)\}$$

The table below gives information on the relative size and structure of this planning world. Column **ops** gives the number of operator instances, **preds** the number of predicate instances, and **states** the approximate number of world states. The final column **rules** gives the combined number of parameterised rule and axiom schemas that were posed by the modeller to specify relations between literals, and to specify inconsistent states.

| | **ops** | **preds** | **states** | **rules** |
|---|---|---|---|---|
| SW: robot | 207 | 208 | $7 \times 10^6$ | 27 |

## A.1 *OS*: Domain Operators

The set of domain operators $OS$ are defined as follows:

$(gothrudoor(D, R),$
$\quad \{\},$
$\quad \{connect(R1, R, D)\},$
$\quad \{robot\_at(D, R1), open(D)\},$
$\quad \{robot\_in(R)\},$
$\quad \{robot\_at(D, R)\},$
$\quad \{robot\_at(D, R1), robot\_near(\_), robot\_in(R1)\})$

$(gotodoor(D, R),$
$\quad \{\},$
$\quad \{connect(R, \_, D)\},$
$\quad \{robot\_in(R)\},$
$\quad \{robot\_at(D, R)\},$
$\quad \{\},$
$\quad \{robot\_at(\_, R), robot\_near(\_)\})$

$(pushthrudoor(Ob, D, To\_room),$
$\quad \{\},$
$\quad \{fits\_thru(Ob, D), connect(R, To\_room, D)\},$
$\quad \{at\_door(Ob, D, R), in\_room(Ob, R), robot\_near(Ob), open(D)\},$
$\quad \{in\_room(Ob, To\_room)\},$
$\quad \{robot\_in(To\_room), robot\_near(Ob)\},$
$\quad \{robot\_at(D, R), at\_door(Ob, D, R), robot\_near(\_), next\_to(Ob, \_),$
$\qquad next\_to(\_, Ob), robot\_in(R), in\_room(Ob, R)\})$

$(pushtodoor(Ob, D, R),$
$\quad \{\},$
$\quad \{connect(R, \_, D), type\_of(Ob, box)\},$
$\quad \{in\_room(Ob, R), robot\_near(Ob)\},$
$\quad \{at\_door(Ob, D, R)\},$
$\quad \{\},$
$\quad \{at\_door(Ob, \_, R), next\_to(Ob, \_), next\_to(\_, Ob)\})$

$(pushtobox(Ob1, Ob2),$
$\{\},$
$\{ne(Ob1, Ob2), type\_of(Ob1, box), type\_of(Ob2, box)\},$
$\{robot\_near(Ob1), in\_room(Ob2, R), in\_room(Ob1, R)\},$
$\{next\_to(Ob1, Ob2)\},$
$\{robot\_near(Ob1)\},$
$\{robot\_at(\_, R), at\_door(Ob1, \_, R), robot\_near(\_), next\_to(Ob1, \_),$
$next\_to(\_, Ob1)\})$

$(goto(X, R),$
$\{in\_room(X, R)\},$
$\{type\_of(X, box), type\_of(R, room)\},$
$\{in\_room(X, R), robot\_in(R)\},$
$\{robot\_near(X)\},$
$\{\},$
$\{robot\_at(\_, R), robot\_near(\_)\})$

$(open(D),$
$\{\},$
$\{connect(R, \_, D)\},$
$\{closed(D), robot\_at(D, R)\},$
$\{open(D)\},$
$\{\},$
$\{closed(D), robot\_near(\_)\})$

$(close(D),$
$\{\},$
$\{connect(R, \_, D)\},$
$\{open(D), robot\_at(D, R)\},$
$\{closed(D)\},$
$\{\},$
$\{open(D), robot\_near(\_)\})$

## A.2 *ES*: Environmental Information

The set of environmental information, *ES*, for this domain is given below. For ease of presentation in this section we have divided it into sub-sets, given below is the set of unchanging predicates and later are two sets of rules that specify relations between predicates in the domain model.

$\{type\_of(room1, room), type\_of(room2, room), type\_of(room3, room),$

$type\_of(room4, room), type\_of(room5, room), type\_of(room6, room),$
$type\_of(room7, room), type\_of(door23, door), type\_of(door24, door),$
$type\_of(door25, door), type\_of(door35, door), type\_of(door45, door),$
$type\_of(door56, door), type\_of(door12, door), type\_of(door47, door),$
$type\_of(door57, door), type\_of(door67, door),$
$type\_of(box2, box), type\_of(big\_box, box), type\_of(box1, box),$
$connect(room2, room3, door23), connect(room3, room2, door23),$
$connect(room2, room4, door24), connect(room4, room2, door24),$
$connect(room2, room5, door25), connect(room5, room2, door25),$
$connect(room6, room5, door56), connect(room5, room6, door56),$
$connect(room5, room3, door35), connect(room3, room5, door35),$
$connect(room5, room4, door45), connect(room4, room5, door45),$
$connect(room4, room7, door47), connect(room7, room4, door47),$
$connect(room5, room7, door57), connect(room7, room5, door57),$
$connect(room6, room7, door67), connect(room7, room6, door67),$
$connect(room1, room2, door12), connect(room2, room1, door12),$
$fits\_thru(box1, door24), fits\_thru(box1, door23), fits\_thru(box1, door25),$
$fits\_thru(box1, door56), fits\_thru(box1, door45), fits\_thru(box1, door35),$
$fits\_thru(box1, door47), fits\_thru(box1, door57), fits\_thru(box1, door12),$
$fits\_thru(box1, door67), fits\_thru(box2, door24), fits\_thru(box2, door23),$
$fits\_thru(box2, door25), fits\_thru(box2, door56), fits\_thru(box2, door45),$
$fits\_thru(box2, door35), fits\_thru(box2, door47), fits\_thru(box2, door57),$
$fits\_thru(box2, door12), fits\_thru(box2, door67), fits\_thru(big\_box, door24),$
$fits\_thru(big\_box, door25), fits\_thru(big\_box, door56),$
$fits\_thru(big\_box, door35), fits\_thru(big\_box, door47),$
$fits\_thru(big\_box, door57), fits\_thru(big\_box, door12),$
$fits\_thru(big\_box, door67)\}$

There are also rules which specify the relation between certain predicates in the domain. Each rule is expressed as a pair $(a, b)$ where $b$ can be inferred from $a$. For this modelled planning domain the rules for achievable predicates (as defined on page 15) in $ES$ is:

$\{(at\_door(O, \_, R), in\_room(O, R)),$
$(\{next\_to(O1, O2), in\_room(O1, R)\}, in\_room(O2, R)),$
$(\{next\_to(O2, O1), in\_room(O1, R)\}, in\_room(O2, R)),$
$(robot\_at(D, R), robot\_in(R)),$
$(\{robot\_near(B), in\_room(B, R)\}, robot\_in(R))\}$

so for example from the set of predicates $\{next\_to(O1, O2), in\_room(O1, R)\}$ the predicate $in\_room(O2, R)$ can be inferred. The rules for unchanging predicates are:

$\{(connect(X, Y, \_), ne(X, Y)),$
$(connect(\_, Y5, Z5), ne(Y5, Z5)),$

$(connect(X4, \_, Z4), ne(X4, Z4)),$
$(connect(X1, Y1, Z1), connect(Y1, X1, Z1)\}$

## A.3 Negative State Invariant: *IN*

The negative state invariant for this modelled planning domain was automatically generated using the procedure outlined below. The set *IN* for this domain is:

$\{(closed(D), open(D)),$
$(robot\_in(R1), robot\_in(R2)),$
$(robot\_in(R1), robot\_at(\_, R2)),$
$(robot\_near(B1), robot\_near(B2)),$
$(in\_room(B1, R1), in\_room(B1, R2)),$
$(in\_room(B1, R1), at\_door(B1, \_, R2)),$
$(robot\_at(D1, R1), robot\_in(R2)),$
$(robot\_at(D1, R1), robot\_at(\_, R2)),$
$(robot\_at(D1, R1), robot\_at(D2, R1)),$
$(at\_door(B1, D1, R1), in\_room(B1, R2)),$
$(at\_door(B1, D1, R1), at\_door(B1, \_, R2)),$
$(at\_door(B1, D1, R1), at\_door(B1, D2, R1))\}$

The procedure that was used to generate *IN* for this domain was based upon the operational definition given in section 4.2.1.

To generate *IN* meant that every pair of achievable predicates for the domain model must be tested to see if they can be achieved in the same state. To simplify the reasoning about possible ways that variables in the predicates could be instantiated the domain model was changed so that for each *sort* of variables in predicate arguments could be instantiated to one of two variables. Then the only pairs that need be considered were combinations of these. For example, consider the predicates $at\_door(B, D, R)$ and $in\_room(B', R')$ where the first argument of both is a variable of sort *box* and the variables $R$ and $R'$ are of sort *room*. Whatever way these variables are instantiated, there are two possibilities for each sort of variable, namely that $B$ and $B'$ are bound to the same object or that they are bound to different objects.

So the procedure uses a "dummy" domain model for compiling *IN* where for *box* the instances of the sort are $\{b1, b2\}$, for *room* they are $\{r1, r2\}$ and so on. This means that the possible combination of comparisons for these two predicates are as follows:

$at\_door(b1, \_, r1), in\_room(b1, r1)$
$at\_door(b2, \_, r1), in\_room(b1, r1)$
$at\_door(b1, \_, r2), in\_room(b1, r1)$
$at\_door(b2, \_, r2), in\_room(b1, r1)$

The "dummy" environmental information, *ES*, is as follows:

$$\{type\_of(r1, room), type\_of(r2, room), type\_of(d1, door), type\_of(d2, door),$$
$$type\_of(b1, box), type\_of(b2, box), connect(r1, r2, d1), connect(r2, r1, d1),$$
$$connect(r1, r2, d2), connect(r2, r1, d2), connect(r2, r2, d2),$$
$$fits\_thru(b1, d1), fits\_thru(b1, d2), fits\_thru(b2, d2),$$
$$fits\_thru(b2, d1)\}$$

After all pairs had been tested a post-processor outputs the set of pairs of predicates that form $IN$ where all predicate arguments are represented as variables, where different capital letters are used to represent the different bindings that were used during processing.

## A.4 Positive State Invariant: $IP_{sort}$

From the environmental information, $ES$, for this modelled planning we have the following sorts:

$$sorts = \{box, robot, door, room\}$$

and each sort is a set of objects as follows:

$$box = \{box1, box2, big\_box\}$$
$$door = \{door12, door23, door24, door25, door35, door45, door47, door57, door67\}$$
$$room = \{room1, room2, room3, room4, room5, room6, room7\}$$
$$robot = \{robot\}$$

The user supplied positive state invariant for this domain is as follows:

$$\begin{aligned} IP_{box} = \{ & \{in\_room(B, R), at\_door(B, D, R)\}, \\ & \{in\_room(B, R), next\_to(B, B')\}, \\ & \{in\_room(B, R)\}\} \\ IP_{robot} = \{ & \{robot\_in(R), robot\_at(D, R)\}, \\ & \{robot\_in(R), robot\_near(B)\}, \\ & \{robot\_in(R)\}\} \end{aligned}$$

Note that there is no user supplied positive state invariant for sorts *room* and *door*. This is perfectly acceptable since the user can supply invariants for as many (or as few) sorts as desired.

## A.5 PRECEDE Goal Orders

The goal order that is compiled by PRECEDE for this domain is as follows:

$$\begin{aligned} P = \{ & \\ & (in\_room(O, R), at\_door(O, D, R)), \\ & (in\_room(O, R), robot\_in(R)), \\ & (robot\_in(R), robot\_at(D, R))\} \end{aligned}$$

## A.6 $\mathcal{M}$: PROPOSE macros

For this modelled planning domain, the set $\mathcal{M}$ is made up of a set $M_s$ where $s$ can be one of the sorts listed in section A.4. The elements of each $M_s$ are pairs of the form $((a, b), c)$ where $a$ and $b$ are a task configuration, $a$ is a start situation for objects of sort $s$, $b$ is the end situation and $c$ is the set of macros for this task configuration.

For simplicity, the anonymous variable "_" is used to represent all variables in predicate arguments in the macro. When the macros are used during planning, these variable arguments must be instantiated to objects of the correct sort. For example, the first element in $M_{robot}$ below has as both the start and end situation of the macro, *robot_in(_)* and *robot_at(_, _)*. Where the first argument of *robot_in* is of sort *room* as is the second argument of *robot_at*. The planning application that accepts a domain model including these compiled macros must be able to bind the variables correctly.

$M_{robot} = \{$
$((\{robot\_in(\_), robot\_at(\_,\_)\}, \{robot\_in(\_), robot\_at(\_,\_)\}), \{$
$[[gotodoor(\_,\_), gothrudoor(\_,\_)]^*],$
$[gothrudoor(\_,\_)],$
$[[gothrudoor(\_,\_), gotodoor(\_,\_)]^*],$
$[gotodoor(\_,\_)]\}),$
$((\{robot\_in(\_), robot\_at(\_,\_)\}, \{robot\_in(\_), robot\_near(\_)\}), \{$
$[[gotodoor(\_,\_), gothrudoor(\_,\_)]^*, goto(\_,\_)]$
$\}),$
$((\{robot\_in(\_), robot\_at(\_,\_)\}, \{robot\_in(\_)\}), \{$
$\}),$
$((\{robot\_in(\_), robot\_near(\_)\}, \{robot\_in(\_), robot\_at(\_,\_)\}), \{$
$[gotodoor(\_,\_)], [gothrudoor(\_,\_), gotodoor(\_,\_)]^*]\}),$
$((\{robot\_in(\_), robot\_near(\_)\}, \{robot\_in(\_), robot\_near(\_)\}), \{$
$[[gothrudoor(\_,\_), gotodoor(\_,\_)]^*], goto(\_,\_)]\}),$
$((\{robot\_in(\_), robot\_near(\_)\}, \{robot\_in(\_)\}), \{$
$\}),$
$((\{robot\_in(\_)\}, \{robot\_in(\_), robot\_at(\_,\_)\}), \{$
$[gotodoor(\_,\_)]\}),$
$((\{robot\_in(\_)\}, \{robot\_in(\_), robot\_near(\_)\}), \{$
$[goto(\_,\_)]\}),$
$((\{robot\_in(\_)\}, \{robot\_in(\_)\}), \{$
$\})\}$

$M_{box} = \{$
$((\{in\_room(\_,\_), at\_door(\_,\_,\_)\}, \{in\_room(\_,\_), at\_door(\_,\_,\_)\}), \{$
$[[pushtodoor(\_,\_,\_), pushthrudoor(\_,\_,\_)]^*],$
$[[pushthrudoor(\_,\_,\_), pushtodoor(\_,\_,\_)]^*],$
$[pushtodoor(\_,\_,\_)]\}),$
$((\{in\_room(\_,\_), at\_door(\_,\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{$
$[pushtobox(\_,\_)]\}),$

$((\{in\_room(\_,\_), at\_door(\_,\_,\_)\}, \{in\_room(\_,\_)\}), \{$
$[[pushthrudoor(\_,\_,\_), pushtodoor(\_,\_,\_)]^*, pushthrudoor(\_,\_,\_)]\}),$
$((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_), at\_door(\_,\_,\_)\}), \{$
$[pushtodoor(\_,\_,\_), [pushthrudoor(\_,\_,\_), pushtodoor(\_,\_,\_)]^*]\}),$
$((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{$
$[[pushtodoor(\_,\_,\_), pushthrudoor(\_,\_,\_)]^*], pushtobox(\_,\_)]\}),$
$((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_)\}), \{$
$[[pushtodoor(\_,\_,\_), pushthrudoor(\_,\_,\_)]^*]\}),$
$((\{in\_room(\_,\_)\}, \{in\_room(\_,\_), at\_door(\_,\_,\_)\}), \{$
$[pushtodoor(\_,\_,\_), [pushthrudoor(\_,\_,\_), pushtodoor(\_,\_,\_)]^*]\}),$
$((\{in\_room(\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{$
$[pushtodoor(\_,\_,\_), pushthrudoor(\_,\_,\_)]^*, pushtobox(\_,\_)]\}),$
$((\{in\_room(\_,\_)\}, \{in\_room(\_,\_)\}), \{$
$[[pushtodoor(\_,\_,\_), pushthrudoor(\_,\_,\_)]^*]\})$
$\}$

## A.7 $\mathcal{EM}$: PROPOSE Enumerations

There is only one static sort that is changed by iterative macros in this modelled planning domain, which is:

$\{door, room\}$

Below we list part of the set of enumerated sequences for this static sort:

$EM_{room,door} = \{$
$((room1, door12),$
$\{[room1, door12]\}),$
$((room1, door23),$
$\{[room1, door12, room2, door23]\}),$
$((room1, door24),$
$\{[room1, door12, room2, door24]\}),$
$((room1, door25),$
$\{[room1, door12, room2, door25]\}),$
$((room1, door35),$
$\{[room1, door12, room2, door23, room3, door35],$
$[room1, door12, room2, door25, room5, door35]\}),$
$((room1, door45),$
$\{[room1, door12, room2, door24, room4, door45]\})$
$((room1, door47),$
$\{[room1, door12, room2, door24, room4, door47]\}),$
$((room1, door57),$
$\{[room1, door12, room2, door25, room5, door57]\}),$
$\ldots\}$

## A.8 Empirical Results

In this section we list the results of the empirical tests that were the subject of chapter 7. The section is divided into three subsections which correspond to the following performance metrics that were used in the tests:

- cumulative CPU time
- cumulative number of partial plans
- number of operators

Each section is subdivided into two, the first part presents the results for a randomly generated problem sample that had been controlled for correctness by removing all tasks that the basic planner *MEA* could not solve. The second section is the results for an uncontrolled problem sample which had been randomly generated.

In the tables the *columns* give the results for each of the 9 different planning applications which were tested (for further explanation of these different applications see section 7.1). Down the left hand side of the tables the *rows* are numbered with the task number. The tasks that were used are listed on page 157.

### A.8.1 Cumulative CPU

#### A.8.1.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 45 | | 11 | | 138 | | 9 | | 36 | | 17 | | 13 | | 12 | | 10 | |
| 2. | 4 | | 3 | | 14 | | 3 | | 3 | | 4 | | 4 | | 3 | | 3 | |
| 3. | 3 | | 2 | | 6 | | 2 | | 3 | | 2 | | 3 | | 2 | | 3 | |
| 4. | 2 | | 3 | | 5 | | 3 | | 2 | | 4 | | 3 | | 3 | | 3 | |
| 5. | 19 | | 10 | | 28 | | 6 | | 21 | | 7 | | 12 | | 7 | | 7 | |
| 6. | 7 | | 11 | f | 12 | | 10 | | 6 | | 6 | f | 9 | f | 8 | | 9 | |
| 7. | 23 | | 11 | | 27 | | 12 | | 17 | | 10 | | 14 | | 11 | | 14 | |
| 8. | 10 | | 3 | | 21 | | 4 | | 12 | | 3 | | 4 | | 4 | | 4 | |
| 9. | 7 | | 6 | | 14 | | 5 | | 8 | | 4 | | 7 | | 5 | | 6 | |
| 10. | 7 | | 3 | | 5 | | 2 | | 8 | | 3 | | 3 | | 3 | | 3 | |
| 11. | 76 | | 6 | | 20 | | 13 | | 19 | | 5 | f | 7 | | 11 | | 13 | |
| 12. | 3 | | 4 | | 8 | | 3 | | 3 | | 4 | | 5 | | 3 | | 4 | |
| 13. | 2 | | 2 | | 5 | | 2 | | 2 | | 1 | | 2 | | 2 | | 2 | |
| 14. | 14 | | 6 | | 11 | | 6 | | 15 | | 5 | | 7 | | 5 | | 7 | |
| 15. | 5 | | 9 | | 10 | | 8 | | 6 | | 7 | | 10 | | 6 | | 10 | |
| 16. | 0 | | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 0 | | 1 | |
| 17. | 4 | | 3 | | 7 | | 4 | | 5 | | 4 | | 4 | | 3 | | 5 | |
| 18. | 31 | | 7 | | 20 | | 7 | | 27 | | 8 | | 9 | | 7 | | 8 | |
| 19. | 5 | | 4 | | 9 | | 3 | | 6 | | 5 | | 5 | | 3 | | 4 | |
| 20. | 36 | | 6 | | 28 | | 12 | | 11 | | 5 | | 7 | | 14 | | 13 | |
| 21. | 2 | | 2 | | 3 | | 1 | | 3 | | 1 | | 2 | | 1 | | 1 | |
| 22. | 0 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 23. | 11 | | 3 | | 12 | | 3 | | 12 | | 2 | | 3 | | 2 | | 3 | |
| 24. | 6 | | 7 | | 5 | | 4 | | 6 | | 5 | | 9 | | 3 | f | 5 | |
| 25. | 11 | | 8 | | 10 | | 8 | | 9 | | 1 | f | 10 | | 7 | | 10 | |
| 26. | 28 | | 3 | | 9 | | 3 | | 5 | | 3 | | 3 | | 4 | | 3 | |
| 27. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 1 | |
| 28. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 1 | | 0 | | 0 | |
| 29. | 3 | | 4 | | 2 | | 2 | | 4 | | 2 | | 4 | | 2 | | 3 | |
| 30. | 19 | | 19 | | 20 | | 10 | | 20 | | 19 | | 15 | | 12 | | 12 | |
| 31. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 32. | 84 | | 11 | | 30 | | 19 | | 24 | | 11 | | 12 | | 10 | f | 22 | |
| 33. | 16 | | 7 | | 10 | | 8 | | 14 | | 7 | | 8 | | 15 | f | 9 | |
| 34. | 8 | | 6 | | 4 | | 8 | | 5 | | 1 | f | 7 | | 8 | | 10 | |
| 35. | 5 | | 3 | | 3 | | 5 | | 5 | | 3 | | 4 | | 4 | | 6 | |
| 36. | 6 | | 8 | | 7 | | 3 | | 4 | | 9 | | 5 | | 4 | | 4 | |
| 37. | 2 | | 2 | | 3 | | 2 | | 3 | | 2 | | 3 | | 1 | | 2 | |
| 38. | 10 | | 7 | | 5 | | 5 | | 7 | | 4 | | 8 | | 5 | | 6 | |
| 39. | 9 | | 6 | | 5 | | 6 | | 8 | | 2 | f | 6 | | 5 | | 7 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40. | 12 | | 5 | | 2 | | 5 | | 13 | | 4 | | 6 | | 2 | | 6 | |
| 41. | 5 | | 5 | | 4 | | 5 | | 6 | | 5 | | 6 | | 4 | | 6 | |
| 42. | 13 | | 6 | | 6 | | 6 | | 5 | | 6 | | 5 | | 6 | | 6 | |
| 43. | 99 | | 5 | | 43 | | 5 | | 105 | | 5 | | 6 | | 5 | | 6 | |
| 44. | 5 | | 4 | | 4 | | 14 | | 6 | | 4 | | 5 | | 11 | | 17 | |
| 45. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 46. | 11 | | 8 | | 5 | | 8 | | 9 | | 6 | | 10 | | 7 | | 9 | |
| 47. | 10 | | 4 | | 3 | | 4 | | 7 | | 4 | | 5 | | 3 | | 5 | |
| 48. | 59 | | 5 | | 13 | | 7 | | 11 | | 5 | | 6 | | 11 | f | 8 | |
| 49. | 0 | | 1 | | 1 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 50. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 51. | 23 | | 19 | | 18 | | 12 | | 21 | | 17 | f | 22 | | 9 | | 14 | |
| 52. | 9 | | 5 | | 6 | | 4 | | 10 | | 4 | | 6 | | 5 | | 5 | |
| 53. | 295 | | 22 | | 57 | | 19 | | 57 | | 12 | | 19 | | 11 | | 18 | |
| 54. | 7 | | 4 | | 6 | | 4 | | 7 | | 4 | | 4 | | 4 | | 5 | |
| 55. | 1 | | 2 | | 1 | | 1 | | 2 | | 1 | | 2 | | 1 | | 2 | |
| 56. | 3 | | 6 | | 3 | | 4 | | 4 | | 2 | | 7 | | 3 | | 4 | |
| 57. | 56 | | 10 | | 36 | | 14 | | 18 | | 10 | | 11 | | 13 | | 16 | |
| 58. | 1 | | 1 | | 2 | | 1 | | 2 | | 1 | | 2 | | 1 | | 1 | |
| 59. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 60. | 6 | | 4 | | 5 | | 5 | | 5 | | 4 | | 5 | | 4 | | 5 | |
| 61. | 4 | | 7 | | 4 | | 8 | | 5 | | 7 | | 9 | | 7 | | 9 | |
| 62. | 4 | | 3 | | 4 | | 3 | | 4 | | 2 | | 3 | | 3 | | 3 | |
| 63. | 6 | | 4 | | 2 | | 4 | | 6 | | 2 | | 4 | | 2 | | 4 | |
| 64. | 241 | | 12 | | 18 | | 23 | | 24 | | 7 | | 12 | | 10 | | 22 | |
| 65. | 84 | | 9 | | 52 | | 10 | | 26 | | 9 | | 11 | | 10 | | 11 | |
| 66. | 3 | | 2 | | 3 | | 2 | | 2 | | 2 | | 2 | | 2 | | 3 | |
| 67. | 8 | | 5 | | 3 | | 5 | | 8 | | 3 | f | 5 | | 4 | | 6 | |
| 68. | 21 | | 12 | | 14 | f | 12 | | 14 | | 8 | | 20 | | 6 | f | 19 | |
| 69. | 4 | | 3 | | 3 | f | 3 | | 4 | | 2 | | 3 | | 6 | | 4 | |
| 70. | 18 | | 9 | | 12 | f | 11 | | 12 | | 7 | | 11 | | 12 | | 12 | |
| 71. | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 72. | 70 | | 6 | | 54 | | 6 | | 58 | | 7 | | 6 | | 12 | | 6 | |
| 73. | 32 | | 5 | | 9 | | 5 | | 6 | | 5 | | 5 | | 5 | | 5 | |
| 74. | 33 | | 11 | | 5 | | 7 | | 6 | | 12 | | 7 | | 6 | | 8 | |
| 75. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 76. | 13 | | 16 | | 6 | | 9 | | 16 | | 6 | | 19 | | 6 | | 11 | |
| 77. | 3 | | 4 | | 4 | | 3 | | 3 | | 2 | | 4 | | 4 | | 3 | |
| 78. | 6 | | 10 | | 2 | | 3 | | 7 | | 3 | f | 11 | | 3 | | 4 | |
| 79. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 80. | 4 | | 2 | | 2 | | 3 | | 3 | | 2 | | 3 | | 2 | | 3 | |
| 81. | 2 | | 2 | | 14 | | 2 | | 3 | | 2 | | 2 | | 2 | | 3 | |
| 82. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |
| 83. | 6 | | 3 | | 4 | | 3 | | 4 | | 3 | | 4 | | 3 | | 4 | |
| 84. | 18 | | 6 | | 15 | | 7 | | 10 | | 5 | | 7 | | 7 | | 9 | |
| 85. | 32 | | 9 | | 52 | f | 9 | | 14 | | 8 | | 10 | | 5 | | 10 | |
| 86. | 2 | | 3 | | 2 | | 3 | | 2 | | 2 | | 4 | | 2 | | 3 | |
| 87. | 8 | | 5 | | 8 | | 5 | | 10 | | 4 | | 6 | | 4 | | 5 | |
| 88. | 34 | | 5 | | 15 | | 9 | | 14 | | 5 | | 5 | | 8 | | 10 | |
| 89. | 16 | | 5 | | 12 | | 5 | | 17 | | 5 | | 5 | | 5 | | 5 | |
| 90. | 10 | | 13 | | 7 | | 14 | | 7 | | 9 | | 16 | | 9 | | 16 | |
| 91. | 8 | | 7 | | 7 | | 7 | | 9 | | 7 | | 9 | | 7 | | 9 | |
| 92. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 93. | 30 | | 24 | | 26 | | 13 | | 19 | | 5 | f | 26 | | 9 | | 15 | |
| 94. | 6 | | 4 | | 6 | | 5 | | 5 | | 4 | | 5 | | 5 | | 5 | |
| 95. | 2 | | 2 | | 2 | f | 2 | | 2 | | 1 | | 2 | | 2 | | 2 | |
| 96. | 6 | | 5 | | 5 | | 4 | | 6 | | 4 | | 6 | | 3 | | 4 | |
| 97. | 13 | | 4 | | 3 | | 4 | | 15 | | 3 | | 5 | | 3 | | 4 | |
| 98. | 5 | | 5 | | 2 | f | 6 | | 6 | | 4 | | 6 | | 5 | | 7 | |
| 99. | 5 | | 3 | | 8 | | 3 | | 5 | | 2 | | 3 | | 2 | | 4 | |
| 100. | 5 | | 6 | | 2 | | 3 | | 5 | | 2 | | 7 | | 2 | | 3 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 1905 | | 562 | | 1126 | | 546 | | 991 | | 442 | | 630 | | 483 | | 624 | |

### A.8.1.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 45 | | 11 | | 88 | | 9 | | 36 | | 17 | | 13 | | 13 | | 10 | |
| 2. | 4 | | 3 | | 7 | | 3 | | 3 | | 4 | | 4 | | 3 | | 3 | |
| 3. | 3 | | 2 | | 2 | | 2 | | 3 | | 2 | | 2 | | 2 | | 3 | |
| 4. | 2 | | 3 | | 2 | | 3 | | 2 | | 4 | | 4 | | 3 | | 3 | |
| 5. | 19 | | 10 | | 16 | | 6 | | 21 | | 7 | | 12 | | 7 | | 7 | |
| 6. | 7 | | 10 | f | 6 | | 10 | | 6 | | 6 | f | 9 | f | 8 | | 9 | |
| 7. | 23 | | 11 | | 14 | | 12 | | 17 | | 10 | | 14 | | 11 | | 14 | |
| 8. | 10 | | 3 | | 11 | | 4 | | 12 | | 3 | | 4 | | 4 | | 4 | |
| 9. | 7 | | 6 | | 7 | | 5 | | 8 | | 5 | | 7 | | 5 | | 6 | |
| 10. | 7 | | 3 | | 2 | | 2 | | 8 | | 3 | | 3 | | 3 | | 3 | |
| 11. | 76 | | 6 | | 10 | | 13 | | 19 | | 5 | f | 7 | | 11 | | 14 | |
| 12. | 3 | | 4 | | 4 | | 3 | | 3 | | 4 | | 5 | | 3 | | 4 | |
| 13. | 2 | | 2 | | 2 | | 2 | | 2 | | 1 | | 2 | | 2 | | 2 | |
| 14. | 13 | | 6 | | 5 | | 6 | | 15 | | 5 | | 7 | | 5 | | 7 | |
| 15. | 5 | | 9 | | 5 | | 8 | | 6 | | 7 | | 10 | | 6 | | 9 | |
| 16. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 17. | 2 | f | 3 | | 2 | f | 6 | | 3 | f | 3 | f | 4 | | 5 | | 7 | |
| 18. | 4 | | 3 | | 3 | | 4 | | 5 | | 4 | | 4 | | 3 | | 5 | |
| 19. | 31 | | 7 | | 11 | | 7 | | 27 | | 8 | | 9 | | 7 | | 8 | |
| 20. | 5 | | 4 | | 5 | | 3 | | 6 | | 5 | | 5 | | 3 | | 4 | |
| 21. | 35 | | 6 | | 16 | | 12 | | 11 | | 5 | | 7 | | 14 | | 13 | |
| 22. | 2 | | 2 | | 2 | | 1 | | 3 | | 1 | | 2 | | 1 | | 1 | |
| 23. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 24. | 10 | | 3 | | 7 | | 3 | | 12 | | 2 | | 3 | | 2 | | 3 | |
| 25. | 6 | | 8 | | 5 | | 4 | | 6 | | 5 | | 9 | | 3 | f | 5 | |
| 26. | 11 | | 8 | | 10 | | 8 | | 9 | | 1 | f | 10 | | 7 | | 10 | |
| 27. | 27 | | 3 | | 9 | | 3 | | 5 | | 3 | | 3 | | 4 | | 3 | |
| 28. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 29. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 1 | | 0 | | 0 | |
| 30. | 3 | | 4 | | 2 | | 2 | | 4 | | 2 | | 4 | | 2 | | 3 | |
| 31. | 3 | f | 3 | f | 2 | f | 3 | f | 4 | f | 2 | f | 3 | f | 2 | f | 4 | f |
| 32. | 19 | | 20 | | 20 | | 10 | | 20 | | 19 | | 15 | | 11 | | 12 | |
| 33. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 34. | 85 | | 11 | | 30 | | 19 | | 24 | | 11 | | 12 | | 10 | f | 21 | |
| 35. | 16 | | 7 | | 10 | | 8 | | 14 | | 7 | | 8 | | 15 | f | 9 | |
| 36. | 129 | f | 12 | f | 56 | f | 16 | f | 104 | f | 9 | f | 10 | f | 4 | f | 15 | f |
| 37. | 8 | | 6 | | 4 | | 8 | | 5 | | 1 | f | 7 | | 8 | | 10 | |
| 38. | 5 | | 3 | | 3 | | 5 | | 5 | | 3 | | 4 | | 4 | | 6 | |
| 39. | 6 | | 8 | | 7 | | 3 | | 4 | | 9 | | 5 | | 5 | | 4 | |
| 40. | 2 | | 2 | | 3 | | 2 | | 3 | | 2 | | 3 | | 2 | | 2 | |
| 41. | 10 | | 7 | | 5 | | 5 | | 7 | | 4 | | 8 | | 5 | | 6 | |
| 42. | 9 | | 5 | | 5 | | 6 | | 8 | | 2 | f | 6 | | 6 | | 7 | |
| 43. | 12 | | 6 | | 2 | | 5 | | 14 | | 4 | | 6 | | 7 | | 6 | |
| 44. | 5 | | 5 | | 4 | | 5 | | 6 | | 5 | | 6 | | 5 | | 6 | |
| 45. | 14 | | 6 | | 6 | | 6 | | 5 | | 6 | | 5 | | 6 | | 6 | |
| 46. | 101 | | 5 | | 43 | | 5 | | 105 | | 5 | | 6 | | 5 | | 6 | |
| 47. | 5 | | 4 | | 4 | | 14 | | 6 | | 4 | | 5 | | 11 | | 17 | |
| 48. | 2 | f | 5 | | 1 | f | 3 | | 2 | f | 3 | | 6 | | 3 | | 4 | |
| 49. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 50. | 11 | | 8 | | 5 | | 8 | | 9 | | 6 | | 10 | | 7 | | 9 | |
| 51. | 10 | | 4 | | 3 | | 4 | | 7 | | 4 | | 5 | | 3 | | 5 | |
| 52. | 60 | | 5 | | 13 | | 7 | | 11 | | 5 | | 6 | | 8 | | 8 | |
| 53. | 0 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 54. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | |
| 55. | 23 | | 19 | | 18 | | 12 | | 21 | | 17 | f | 22 | | 15 | | 14 | |
| 56. | 9 | | 5 | | 6 | | 4 | | 10 | | 4 | | 6 | | 5 | | 5 | |
| 57. | 295 | | 22 | | 57 | | 20 | | 57 | | 12 | | 19 | | 30 | | 18 | |
| 58. | 7 | | 4 | | 6 | | 4 | | 7 | | 4 | | 4 | | 5 | | 5 | |
| 59. | 1 | | 1 | | 1 | | 2 | | 2 | | 1 | | 2 | | 1 | | 2 | |
| 60. | 3 | | 6 | | 3 | | 4 | | 4 | | 2 | | 7 | | 4 | | 4 | |
| 61. | 56 | | 10 | | 35 | | 14 | | 18 | | 10 | | 11 | | 22 | | 16 | |
| 62. | 1 | | 1 | | 2 | | 1 | | 2 | | 1 | | 2 | | 2 | | 1 | |
| 63. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 64. | 6 | | 4 | | 5 | | 5 | | 5 | | 4 | | 5 | | 5 | | 5 | |
| 65. | 4 | | 7 | | 4 | | 8 | | 5 | | 7 | | 9 | | 8 | | 10 | |
| 66. | 4 | | 3 | | 4 | | 3 | | 4 | | 2 | | 3 | | 3 | | 3 | |
| 67. | 6 | | 4 | | 2 | | 4 | | 6 | | 2 | | 4 | | 3 | | 4 | |
| 68. | 244 | | 12 | | 18 | | 22 | | 24 | | 7 | | 13 | | 12 | | 22 | |
| 69. | 84 | | 9 | | 52 | | 10 | | 26 | | 10 | | 11 | | 3 | f | 11 | |
| 70. | 3 | | 2 | | 3 | | 2 | | 2 | | 2 | | 2 | | 3 | | 3 | |
| 71. | 8 | | 5 | | 3 | | 5 | | 9 | | 3 | f | 6 | | 3 | | 6 | |
| 72. | 21 | | 12 | | 14 | f | 12 | | 15 | | 8 | | 20 | | 11 | f | 20 | |
| 73. | 4 | | 3 | | 3 | f | 3 | | 4 | | 2 | | 3 | | 3 | f | 4 | |
| 74. | 18 | | 9 | | 12 | f | 10 | | 12 | | 7 | | 11 | | 9 | f | 12 | |
| 75. | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | | 2 | | 2 | | 1 | |
| 76. | 70 | | 6 | | 54 | | 6 | | 58 | | 7 | | 6 | | 15 | f | 6 | |
| 77. | 32 | | 5 | | 9 | | 5 | | 6 | | 5 | | 5 | | 2 | f | 5 | |
| 78. | 33 | | 11 | | 5 | | 7 | | 5 | | 12 | | 7 | | 7 | | 8 | |
| 79. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80. | 13 | | 16 | | 6 | | 9 | | 16 | | 6 | | 19 | | 8 | | 11 | |
| 81. | 3 | | 4 | | 4 | | 3 | | 3 | | 2 | | 4 | | 5 | | 3 | |
| 82. | 6 | | 10 | | 2 | | 3 | | 7 | | 3 | f | 11 | | 3 | | 4 | |
| 83. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 1 | |
| 84. | 4 | | 2 | | 2 | | 3 | | 3 | | 2 | | 3 | | 2 | | 3 | |
| 85. | 2 | | 2 | | 14 | | 2 | | 3 | | 2 | | 2 | | 2 | | 3 | |
| 86. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 2 | | 2 | |
| 87. | 6 | | 4 | | 4 | | 3 | | 4 | | 3 | | 4 | | 5 | f | 4 | |
| 88. | 18 | | 6 | | 15 | | 7 | | 10 | | 5 | | 7 | | 25 | | 9 | |
| 89. | 31 | | 9 | | 52 | f | 9 | | 14 | | 8 | | 10 | | 19 | | 10 | |
| 90. | 2 | | 3 | | 2 | | 3 | | 2 | | 2 | | 4 | | 2 | | 3 | |
| 91. | 8 | | 5 | | 8 | | 5 | | 10 | | 4 | | 6 | | 4 | | 5 | |
| 92. | 34 | | 5 | | 15 | | 9 | | 14 | | 5 | | 5 | | 8 | | 10 | |
| 93. | 16 | | 5 | | 12 | | 5 | | 17 | | 5 | | 5 | | 5 | | 5 | |
| 94. | 10 | | 14 | | 7 | | 13 | | 7 | | 9 | | 16 | | 12 | | 16 | |
| 95. | 8 | | 7 | | 7 | | 8 | | 9 | | 7 | | 9 | | 9 | | 9 | |
| 96. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 97. | 29 | | 25 | | 26 | | 13 | | 18 | | 5 | f | 26 | | 14 | | 15 | |
| 98. | 6 | | 5 | | 6 | | 5 | | 6 | | 4 | | 5 | | 1 | f | 6 | |
| 99. | 2 | | 2 | | 2 | f | 2 | | 2 | | 1 | | 2 | | 3 | | 2 | |
| 100. | 6 | | 5 | | 5 | | 4 | | 6 | | 4 | | 5 | | 5 | | 4 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 2016 | | 572 | | 992 | | 559 | | 1076 | | 450 | | 634 | | 576 | | 639 | |

## A.8.2 Partial Plans

### A.8.2.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 410 | | 146 | | 631 | | 115 | | 294 | | 199 | | 145 | | 138 | | 114 | |
| 2. | 64 | | 56 | | 95 | | 44 | | 46 | | 46 | | 54 | | 44 | | 42 | |
| 3. | 54 | | 42 | | 48 | | 42 | | 54 | | 34 | | 42 | | 40 | | 42 | |
| 4. | 44 | | 51 | | 34 | | 43 | | 33 | | 46 | | 51 | | 40 | | 43 | |
| 5. | 195 | | 131 | | 169 | | 83 | | 192 | | 83 | | 128 | | 76 | | 80 | |
| 6. | 112 | | 175 | f | 82 | | 158 | | 87 | | 99 | f | 126 | f | 97 | | 123 | |
| 7. | 271 | | 153 | | 161 | | 151 | | 205 | | 118 | | 163 | | 120 | | 161 | |
| 8. | 144 | | 51 | | 131 | | 58 | | 144 | | 45 | | 51 | | 52 | | 58 | |
| 9. | 112 | | 99 | | 93 | | 80 | | 106 | | 64 | | 98 | | 70 | | 79 | |
| 10. | 98 | | 36 | | 36 | | 38 | | 98 | | 36 | | 36 | | 33 | | 38 | |
| 11. | 452 | | 92 | | 105 | | 153 | | 187 | | 73 | f | 88 | | 98 | | 141 | |
| 12. | 59 | | 66 | | 44 | | 49 | | 51 | | 43 | | 66 | | 31 | | 48 | |
| 13. | 33 | | 20 | | 31 | | 27 | | 30 | | 17 | | 20 | | 24 | | 27 | |
| 14. | 185 | | 89 | | 70 | | 84 | | 174 | | 58 | | 87 | | 62 | | 82 | |
| 15. | 91 | | 133 | | 66 | | 123 | | 83 | | 95 | | 127 | | 62 | | 117 | |
| 16. | 7 | | 7 | | 2 | | 7 | | 7 | | 7 | | 7 | | 2 | | 7 | |
| 17. | 66 | | 49 | | 37 | | 59 | | 63 | | 40 | | 49 | | 30 | | 59 | |
| 18. | 327 | | 104 | | 106 | | 90 | | 264 | | 89 | | 106 | | 64 | | 94 | |
| 19. | 80 | | 73 | | 58 | | 52 | | 80 | | 70 | | 73 | | 42 | | 52 | |
| 20. | 313 | | 85 | | 135 | | 152 | | 133 | | 58 | | 84 | | 132 | | 143 | |
| 21. | 36 | | 27 | | 22 | | 20 | | 36 | | 12 | | 27 | | 12 | | 20 | |
| 22. | 8 | | 8 | | 10 | | 8 | | 8 | | 6 | | 8 | | 3 | | 8 | |
| 23. | 123 | | 50 | | 64 | | 49 | | 123 | | 26 | | 45 | | 24 | | 44 | |
| 24. | 104 | | 118 | | 54 | | 68 | | 89 | | 59 | | 117 | | 36 | f | 67 | |
| 25. | 149 | | 120 | | 91 | | 116 | | 122 | | 5 | f | 116 | | 62 | | 112 | |
| 26. | 218 | | 49 | | 85 | | 50 | | 68 | | 36 | | 42 | | 45 | | 43 | |
| 27. | 4 | | 4 | | 3 | | 4 | | 4 | | 3 | | 4 | | 3 | | 4 | |
| 28. | 3 | | 5 | | 3 | | 3 | | 3 | | 3 | | 5 | | 3 | | 3 | |
| 29. | 57 | | 62 | | 22 | | 38 | | 57 | | 25 | | 50 | | 22 | | 38 | |
| 30. | 225 | | 225 | | 140 | | 115 | | 199 | | 160 | | 152 | | 108 | | 115 | |
| 31. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 32. | 594 | | 140 | | 207 | | 212 | | 237 | | 119 | | 124 | | 102 | f | 196 | |
| 33. | 192 | | 97 | | 87 | | 101 | | 148 | | 81 | | 92 | | 136 | f | 96 | |
| 34. | 106 | | 84 | | 35 | | 116 | | 56 | | 11 | f | 81 | | 86 | | 113 | |
| 35. | 83 | | 55 | | 39 | | 86 | | 74 | | 43 | | 55 | | 47 | | 84 | |
| 36. | 83 | | 113 | | 57 | | 47 | | 43 | | 77 | | 49 | | 53 | | 47 | |
| 37. | 44 | | 36 | | 36 | | 25 | | 44 | | 17 | | 36 | | 17 | | 25 | |
| 38. | 143 | | 101 | | 58 | | 69 | | 95 | | 48 | | 96 | | 50 | | 66 | |
| 39. | 143 | | 84 | | 47 | | 88 | | 113 | | 24 | f | 75 | | 48 | | 79 | |
| 40. | 127 | | 71 | | 27 | | 62 | | 127 | | 37 | | 71 | | 23 | | 62 | |
| 41. | 80 | | 73 | | 41 | | 68 | | 74 | | 54 | | 73 | | 39 | | 67 | |
| 42. | 152 | | 78 | | 62 | | 80 | | 56 | | 54 | | 55 | | 59 | | 57 | |
| 43. | 459 | | 59 | | 276 | | 61 | | 459 | | 49 | | 59 | | 56 | | 61 | |
| 44. | 71 | | 51 | | 38 | | 163 | | 71 | | 44 | | 51 | | 86 | | 163 | |
| 45. | 8 | | 8 | | 5 | | 8 | | 8 | | 7 | | 8 | | 5 | | 8 | |
| 46. | 151 | | 112 | | 57 | | 107 | | 113 | | 62 | | 111 | | 74 | | 102 | |
| 47. | 114 | | 63 | | 28 | | 61 | | 85 | | 40 | | 63 | | 33 | | 61 | |
| 48. | 451 | | 79 | | 125 | | 100 | | 120 | | 58 | | 69 | | 113 | f | 86 | |
| 49. | 6 | | 6 | | 2 | | 6 | | 6 | | 2 | | 6 | | 2 | | 6 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50. | 3 | | 3 | | 6 | | 3 | | 3 | | 6 | | 3 | | 1 | | 3 | |
| 51. | 254 | | 238 | | 155 | | 156 | | 206 | | 207 | f | 236 | | 98 | | 154 | |
| 52. | 124 | | 71 | | 57 | | 58 | | 117 | | 40 | | 64 | | 58 | | 51 | |
| 53. | 1243 | | 250 | | 394 | | 214 | | 448 | | 134 | | 199 | | 124 | | 173 | |
| 54. | 109 | | 52 | | 55 | | 61 | | 91 | | 41 | | 49 | | 41 | | 58 | |
| 55. | 24 | | 25 | | 10 | | 25 | | 24 | | 8 | | 25 | | 12 | | 25 | |
| 56. | 57 | | 85 | | 23 | | 55 | | 57 | | 21 | | 85 | | 23 | | 55 | |
| 57. | 412 | | 116 | | 205 | | 154 | | 166 | | 89 | | 111 | | 105 | | 149 | |
| 58. | 24 | | 20 | | 14 | | 20 | | 24 | | 14 | | 20 | | 14 | | 20 | |
| 59. | 15 | | 11 | | 5 | | 11 | | 15 | | 5 | | 11 | | 5 | | 11 | |
| 60. | 87 | | 68 | | 56 | | 63 | | 53 | | 42 | | 62 | | 46 | | 57 | |
| 61. | 57 | | 77 | | 35 | | 77 | | 53 | | 66 | | 73 | | 64 | | 73 | |
| 62. | 57 | | 34 | | 46 | | 34 | | 54 | | 19 | | 31 | | 30 | | 31 | |
| 63. | 86 | | 59 | | 15 | | 59 | | 86 | | 20 | | 59 | | 15 | | 59 | |
| 64. | 818 | | 128 | | 150 | | 185 | | 220 | | 73 | | 116 | | 97 | | 168 | |
| 65. | 587 | | 128 | | 386 | | 126 | | 252 | | 97 | | 120 | | 102 | | 118 | |
| 66. | 48 | | 19 | | 36 | | 36 | | 34 | | 12 | | 19 | | 16 | | 32 | |
| 67. | 104 | | 74 | | 30 | | 74 | | 104 | | 45 | f | 74 | | 40 | | 74 | |
| 68. | 253 | | 169 | | 124 | f | 163 | | 175 | | 94 | | 226 | | 73 | f | 217 | |
| 69. | 68 | | 36 | | 32 | f | 47 | | 59 | | 23 | | 32 | | 71 | | 43 | |
| 70. | 230 | | 135 | | 126 | f | 132 | | 148 | | 72 | | 129 | | 121 | | 125 | |
| 71. | 14 | | 13 | | 19 | | 17 | | 14 | | 13 | | 13 | | 16 | | 17 | |
| 72. | 487 | | 77 | | 389 | | 78 | | 364 | | 80 | | 67 | | 115 | | 68 | |
| 73. | 262 | | 74 | | 72 | | 71 | | 73 | | 52 | | 63 | | 59 | | 63 | |
| 74. | 260 | | 136 | | 46 | | 90 | | 71 | | 121 | | 74 | | 61 | | 90 | |
| 75. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 76. | 180 | | 199 | | 55 | | 125 | | 180 | | 68 | | 195 | | 66 | | 121 | |
| 77. | 50 | | 61 | | 40 | | 54 | | 47 | | 19 | | 58 | | 46 | | 51 | |
| 78. | 91 | | 119 | | 17 | | 52 | | 91 | | 33 | f | 119 | | 36 | | 52 | |
| 79. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 80. | 63 | | 36 | | 25 | | 42 | | 40 | | 27 | | 36 | | 25 | | 39 | |
| 81. | 40 | | 26 | | 156 | | 33 | | 36 | | 22 | | 22 | | 26 | | 29 | |
| 82. | 20 | | 20 | | 6 | | 20 | | 20 | | 14 | | 20 | | 5 | | 20 | |
| 83. | 95 | | 57 | | 31 | | 56 | | 66 | | 36 | | 53 | | 34 | | 52 | |
| 84. | 196 | | 78 | | 129 | | 101 | | 118 | | 61 | | 77 | | 69 | | 100 | |
| 85. | 297 | | 122 | | 336 | f | 120 | | 139 | | 91 | | 105 | | 51 | | 101 | |
| 86. | 34 | | 45 | | 18 | | 42 | | 34 | | 28 | | 45 | | 18 | | 42 | |
| 87. | 110 | | 73 | | 79 | | 59 | | 107 | | 46 | | 70 | | 44 | | 56 | |
| 88. | 330 | | 62 | | 104 | | 122 | | 154 | | 54 | | 57 | | 77 | | 114 | |
| 89. | 181 | | 65 | | 128 | | 65 | | 164 | | 51 | | 58 | | 58 | | 58 | |
| 90. | 142 | | 173 | | 77 | | 170 | | 94 | | 92 | | 170 | | 82 | | 167 | |
| 91. | 107 | | 102 | | 61 | | 101 | | 107 | | 92 | | 102 | | 68 | | 101 | |
| 92. | 9 | | 9 | | 6 | | 9 | | 9 | | 6 | | 9 | | 6 | | 9 | |
| 93. | 332 | | 301 | | 200 | | 165 | | 218 | | 75 | f | 290 | | 85 | | 163 | |
| 94. | 90 | | 60 | | 64 | | 66 | | 70 | | 41 | | 57 | | 50 | | 63 | |
| 95. | 28 | | 25 | | 10 | f | 25 | | 28 | | 12 | | 25 | | 14 | | 25 | |
| 96. | 96 | | 69 | | 47 | | 57 | | 76 | | 42 | | 63 | | 38 | | 51 | |
| 97. | 153 | | 58 | | 31 | | 55 | | 153 | | 35 | | 58 | | 33 | | 55 | |
| 98. | 81 | | 74 | | 13 | f | 83 | | 80 | | 47 | | 73 | | 57 | | 82 | |
| 99. | 80 | | 39 | | 51 | | 45 | | 71 | | 23 | | 36 | | 21 | | 42 | |
| 100. | 68 | | 80 | | 12 | | 38 | | 68 | | 18 | | 80 | | 20 | | 38 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 15608 | | 7688 | | 8038 | | 7254 | | 10051 | | 4910 | | 7181 | | 5041 | | 6949 | |

### A.8.2.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 410 | | 146 | | 631 | | 115 | | 294 | | 199 | | 145 | | 138 | | 114 | |
| 2. | 64 | | 56 | | 95 | | 44 | | 46 | | 46 | | 54 | | 44 | | 42 | |
| 3. | 54 | | 42 | | 48 | | 42 | | 54 | | 34 | | 42 | | 40 | | 42 | |
| 4. | 44 | | 51 | | 34 | | 43 | | 33 | | 46 | | 51 | | 40 | | 43 | |
| 5. | 195 | | 131 | | 169 | | 83 | | 192 | | 83 | | 128 | | 76 | | 80 | |
| 6. | 112 | | 175 | f | 82 | | 158 | | 87 | | 99 | f | 126 | f | 97 | | 123 | |
| 7. | 271 | | 153 | | 161 | | 151 | | 205 | | 118 | | 163 | | 120 | | 161 | |
| 8. | 144 | | 51 | | 131 | | 58 | | 144 | | 45 | | 51 | | 52 | | 58 | |
| 9. | 112 | | 99 | | 93 | | 80 | | 106 | | 64 | | 98 | | 70 | | 79 | |
| 10. | 98 | | 36 | | 36 | | 38 | | 98 | | 36 | | 36 | | 33 | | 38 | |
| 11. | 452 | | 92 | | 105 | | 153 | | 187 | | 73 | f | 88 | | 98 | | 141 | |
| 12. | 59 | | 66 | | 44 | | 49 | | 51 | | 43 | | 66 | | 31 | | 48 | |
| 13. | 33 | | 20 | | 31 | | 27 | | 30 | | 17 | | 20 | | 24 | | 27 | |
| 14. | 185 | | 89 | | 70 | | 84 | | 174 | | 58 | | 87 | | 62 | | 82 | |
| 15. | 91 | | 133 | | 66 | | 123 | | 83 | | 95 | | 127 | | 62 | | 117 | |
| 16. | 7 | | 7 | | 2 | | 7 | | 7 | | 7 | | 7 | | 2 | | 7 | |
| 17. | 45 | f | 43 | | 35 | f | 82 | | 45 | f | 38 | f | 43 | | 53 | | 82 | |
| 18. | 66 | | 49 | | 37 | | 59 | | 63 | | 40 | | 49 | | 30 | | 59 | |
| 19. | 327 | | 104 | | 106 | | 90 | | 264 | | 89 | | 106 | | 64 | | 94 | |
| 20. | 80 | | 73 | | 58 | | 52 | | 80 | | 70 | | 73 | | 42 | | 52 | |
| 21. | 313 | | 85 | | 135 | | 152 | | 133 | | 58 | | 84 | | 132 | | 143 | |
| 22. | 36 | | 27 | | 22 | | 20 | | 36 | | 12 | | 27 | | 12 | | 20 | |
| 23. | 8 | | 8 | | 10 | | 8 | | 8 | | 6 | | 8 | | 3 | | 8 | |
| 24. | 123 | | 50 | | 64 | | 49 | | 123 | | 26 | | 45 | | 24 | | 44 | |
| 25. | 104 | | 118 | | 54 | | 68 | | 89 | | 59 | | 117 | | 36 | f | 67 | |
| 26. | 149 | | 120 | | 91 | | 116 | | 122 | | 5 | f | 116 | | 62 | | 112 | |
| 27. | 218 | | 49 | | 85 | | 50 | | 68 | | 36 | | 42 | | 45 | | 43 | |
| 28. | 4 | | 4 | | 3 | | 4 | | 4 | | 3 | | 4 | | 3 | | 4 | |
| 29. | 3 | | 5 | | 3 | | 3 | | 3 | | 3 | | 5 | | 3 | | 3 | |
| 30. | 57 | | 62 | | 22 | | 38 | | 57 | | 25 | | 50 | | 22 | | 38 | |
| 31. | 65 | f | 46 | f | 29 | f | 46 | f | 65 | f | 28 | f | 46 | f | 28 | f | 46 | f |
| 32. | 225 | | 225 | | 140 | | 115 | | 199 | | 160 | | 152 | | 108 | | 115 | |
| 33. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 34. | 594 | | 140 | | 207 | | 212 | | 237 | | 119 | | 124 | | 102 | f | 196 | |
| 35. | 192 | | 97 | | 87 | | 101 | | 148 | | 81 | | 92 | | 136 | f | 96 | |
| 36. | 828 | f | 155 | f | 367 | f | 203 | f | 684 | f | 109 | f | 112 | f | 43 | f | 160 | f |
| 37. | 106 | | 84 | | 35 | | 116 | | 56 | | 11 | f | 81 | | 86 | | 113 | |
| 38. | 83 | | 55 | | 39 | | 86 | | 74 | | 43 | | 55 | | 47 | | 84 | |
| 39. | 83 | | 113 | | 57 | | 47 | | 43 | | 77 | | 49 | | 56 | | 47 | |
| 40. | 44 | | 36 | | 36 | | 25 | | 44 | | 17 | | 36 | | 17 | | 25 | |
| 41. | 143 | | 101 | | 58 | | 69 | | 95 | | 48 | | 96 | | 50 | | 66 | |
| 42. | 143 | | 84 | | 47 | | 88 | | 113 | | 24 | f | 75 | | 53 | | 79 | |
| 43. | 127 | | 71 | | 27 | | 62 | | 127 | | 37 | | 71 | | 64 | | 62 | |
| 44. | 80 | | 73 | | 41 | | 68 | | 74 | | 54 | | 73 | | 43 | | 67 | |
| 45. | 152 | | 78 | | 62 | | 80 | | 56 | | 54 | | 55 | | 61 | | 57 | |
| 46. | 459 | | 59 | | 276 | | 61 | | 459 | | 49 | | 59 | | 65 | | 61 | |
| 47. | 71 | | 51 | | 38 | | 163 | | 71 | | 44 | | 51 | | 86 | | 163 | |
| 48. | 38 | f | 87 | | 21 | f | 51 | | 38 | f | 45 | | 87 | | 31 | | 50 | |
| 49. | 8 | | 8 | | 5 | | 8 | | 8 | | 7 | | 8 | | 5 | | 8 | |
| 50. | 151 | | 112 | | 57 | | 107 | | 113 | | 62 | | 111 | | 69 | | 102 | |
| 51. | 114 | | 63 | | 28 | | 61 | | 85 | | 40 | | 63 | | 38 | | 61 | |
| 52. | 451 | | 79 | | 125 | | 100 | | 120 | | 58 | | 69 | | 80 | | 86 | |
| 53. | 6 | | 6 | | 2 | | 6 | | 6 | | 2 | | 6 | | 2 | | 6 | |
| 54. | 3 | | 3 | | 6 | | 3 | | 3 | | 6 | | 3 | | 6 | | 3 | |
| 55. | 254 | | 238 | | 155 | | 156 | | 206 | | 207 | f | 236 | | 137 | | 154 | |
| 56. | 124 | | 71 | | 57 | | 58 | | 117 | | 40 | | 64 | | 51 | | 51 | |
| 57. | 1243 | | 250 | | 394 | | 214 | | 448 | | 134 | | 199 | | 249 | | 173 | |
| 58. | 109 | | 52 | | 55 | | 61 | | 91 | | 41 | | 49 | | 41 | | 58 | |
| 59. | 24 | | 25 | | 10 | | 25 | | 24 | | 8 | | 25 | | 11 | | 25 | |
| 60. | 57 | | 85 | | 23 | | 55 | | 57 | | 21 | | 85 | | 23 | | 55 | |
| 61. | 412 | | 116 | | 205 | | 154 | | 166 | | 89 | | 111 | | 155 | | 149 | |
| 62. | 24 | | 20 | | 14 | | 20 | | 24 | | 14 | | 20 | | 14 | | 20 | |
| 63. | 15 | | 11 | | 5 | | 11 | | 15 | | 5 | | 11 | | 5 | | 11 | |
| 64. | 87 | | 68 | | 56 | | 63 | | 53 | | 42 | | 62 | | 46 | | 57 | |
| 65. | 57 | | 77 | | 35 | | 77 | | 53 | | 66 | | 73 | | 64 | | 73 | |
| 66. | 57 | | 34 | | 46 | | 34 | | 54 | | 19 | | 31 | | 30 | | 31 | |
| 67. | 86 | | 59 | | 15 | | 59 | | 86 | | 20 | | 59 | | 18 | | 59 | |
| 68. | 818 | | 128 | | 150 | | 185 | | 220 | | 73 | | 116 | | 105 | | 168 | |
| 69. | 587 | | 128 | | 386 | | 126 | | 252 | | 97 | | 120 | | 38 | f | 118 | |
| 70. | 48 | | 19 | | 36 | | 36 | | 34 | | 12 | | 19 | | 35 | | 32 | |
| 71. | 104 | | 74 | | 30 | | 74 | | 104 | | 45 | f | 74 | | 40 | | 74 | |
| 72. | 253 | | 169 | | 124 | f | 163 | | 175 | | 94 | | 226 | | 96 | f | 217 | |
| 73. | 68 | | 36 | | 32 | f | 47 | | 59 | | 23 | | 32 | | 19 | f | 43 | |
| 74. | 230 | | 135 | | 126 | f | 132 | | 148 | | 72 | | 129 | | 82 | f | 125 | |
| 75. | 14 | | 13 | | 19 | | 17 | | 14 | | 13 | | 13 | | 16 | | 17 | |
| 76. | 487 | | 77 | | 389 | | 78 | | 364 | | 80 | | 67 | | 100 | f | 68 | |
| 77. | 262 | | 74 | | 72 | | 71 | | 73 | | 52 | | 63 | | 14 | f | 63 | |
| 78. | 260 | | 136 | | 46 | | 90 | | 71 | | 121 | | 74 | | 63 | | 90 | |
| 79. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80. | 180 | | 199 | | 55 | | 125 | | 180 | | 68 | | 195 | | 96 | | 121 | |
| 81. | 50 | | 61 | | 40 | | 54 | | 47 | | 19 | | 58 | | 75 | | 51 | |
| 82. | 91 | | 119 | | 17 | | 52 | | 91 | | 33 | f | 119 | | 36 | | 52 | |
| 83. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 84. | 63 | | 36 | | 25 | | 42 | | 40 | | 27 | | 36 | | 25 | | 39 | |
| 85. | 40 | | 26 | | 156 | | 33 | | 36 | | 22 | | 22 | | 26 | | 29 | |
| 86. | 20 | | 20 | | 6 | | 20 | | 20 | | 14 | | 20 | | 25 | | 20 | |
| 87. | 95 | | 57 | | 31 | | 56 | | 66 | | 36 | | 53 | | 66 | f | 52 | |
| 88. | 196 | | 78 | | 129 | | 101 | | 118 | | 61 | | 77 | | 210 | | 100 | |
| 89. | 297 | | 122 | | 336 | f | 120 | | 139 | | 91 | | 105 | | 193 | | 101 | |
| 90. | 34 | | 45 | | 18 | | 42 | | 34 | | 28 | | 45 | | 33 | | 42 | |
| 91. | 110 | | 73 | | 79 | | 59 | | 107 | | 46 | | 70 | | 48 | | 56 | |
| 92. | 330 | | 62 | | 104 | | 122 | | 154 | | 54 | | 57 | | 91 | | 114 | |
| 93. | 181 | | 65 | | 128 | | 65 | | 164 | | 51 | | 58 | | 59 | | 58 | |
| 94. | 142 | | 173 | | 77 | | 170 | | 94 | | 92 | | 170 | | 126 | | 167 | |
| 95. | 107 | | 102 | | 61 | | 101 | | 107 | | 92 | | 102 | | 61 | | 101 | |
| 96. | 9 | | 9 | | 6 | | 9 | | 9 | | 6 | | 9 | | 6 | | 9 | |
| 97. | 332 | | 301 | | 200 | | 165 | | 218 | | 75 | f | 290 | | 119 | | 163 | |
| 98. | 90 | | 60 | | 64 | | 66 | | 70 | | 41 | | 57 | | 12 | f | 63 | |
| 99. | 28 | | 25 | | 10 | f | 25 | | 28 | | 12 | | 25 | | 29 | | 25 | |
| 100. | 96 | | 69 | | 47 | | 57 | | 76 | | 42 | | 63 | | 54 | | 51 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 16202 | | 7768 | | 8383 | | 7415 | | 10511 | | 5007 | | 7222 | | 5639 | | 7070 | |

### A.8.3 Operators

#### A.8.3.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 40 | | 38 | | 39 | | 38 | | 40 | | 40 | | 38 | | 39 | | 38 | |
| 2. | 20 | | 17 | | 20 | | 15 | | 20 | | 17 | | 17 | | 15 | | 15 | |
| 3. | 15 | | 13 | | 13 | | 13 | | 15 | | 13 | | 13 | | 13 | | 13 | |
| 4. | 13 | | 15 | | 13 | | 15 | | 13 | | 15 | | 15 | | 15 | | 15 | |
| 5. | 23 | | 23 | | 24 | | 24 | | 23 | | 23 | | 23 | | 24 | | 24 | |
| 6. | 25 | | 0 | f | 25 | | 31 | | 25 | | 0 | f | 0 | f | 35 | | 29 | |
| 7. | 43 | | 40 | | 43 | | 40 | | 46 | | 43 | | 46 | | 43 | | 46 | |
| 8. | 21 | | 18 | | 22 | | 18 | | 21 | | 18 | | 18 | | 18 | | 18 | |
| 9. | 25 | | 22 | | 25 | | 25 | | 25 | | 22 | | 22 | | 27 | | 25 | |
| 10. | 10 | | 15 | | 14 | | 10 | | 10 | | 14 | | 15 | | 14 | | 10 | |
| 11. | 24 | | 20 | | 22 | | 24 | | 24 | | 0 | f | 20 | | 24 | | 24 | |
| 12. | 18 | | 17 | | 18 | | 18 | | 18 | | 19 | | 17 | | 18 | | 18 | |
| 13. | 11 | | 9 | | 11 | | 9 | | 11 | | 9 | | 9 | | 9 | | 9 | |
| 14. | 31 | | 27 | | 31 | | 31 | | 31 | | 27 | | 27 | | 31 | | 31 | |
| 15. | 22 | | 29 | | 22 | | 29 | | 22 | | 29 | | 29 | | 29 | | 29 | |
| 16. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 17. | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | |
| 18. | 30 | | 31 | | 31 | | 32 | | 31 | | 35 | | 32 | | 32 | | 31 | |
| 19. | 16 | | 15 | | 16 | | 16 | | 16 | | 23 | | 15 | | 16 | | 16 | |
| 20. | 42 | | 30 | | 49 | | 42 | | 39 | | 30 | | 30 | | 49 | | 39 | |
| 21. | 9 | | 11 | | 7 | | 7 | | 9 | | 7 | | 11 | | 7 | | 7 | |
| 22. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 23. | 16 | | 14 | | 14 | | 14 | | 16 | | 14 | | 14 | | 14 | | 14 | |
| 24. | 25 | | 23 | | 25 | | 22 | | 25 | | 23 | | 23 | | 0 | f | 22 | |
| 25. | 29 | | 34 | | 30 | | 34 | | 30 | | 0 | f | 34 | | 38 | | 34 | |
| 26. | 15 | | 15 | | 18 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | |
| 27. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 28. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 29. | 12 | | 13 | | 12 | | 12 | | 12 | | 13 | | 13 | | 12 | | 12 | |
| 30. | 39 | | 46 | | 40 | | 39 | | 39 | | 39 | | 46 | | 40 | | 39 | |
| 31. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 32. | 33 | | 27 | | 33 | | 33 | | 33 | | 27 | | 27 | | 0 | f | 33 | |
| 33. | 34 | | 34 | | 34 | | 34 | | 34 | | 34 | | 34 | | 0 | f | 34 | |
| 34. | 20 | | 28 | | 20 | | 27 | | 20 | | 0 | f | 28 | | 27 | | 27 | |
| 35. | 27 | | 19 | | 22 | | 27 | | 27 | | 17 | | 19 | | 27 | | 27 | |
| 36. | 17 | | 32 | | 18 | | 16 | | 18 | | 32 | | 19 | | 16 | | 16 | |
| 37. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 38. | 24 | | 28 | | 24 | | 24 | | 24 | | 22 | | 28 | | 24 | | 24 | |
| 39. | 30 | | 24 | | 29 | | 30 | | 30 | | 0 | f | 24 | | 30 | | 30 | |
| 40. | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | |
| 41. | 25 | | 24 | | 25 | | 25 | | 25 | | 24 | | 24 | | 25 | | 25 | |
| 42. | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | | 32 | | 26 | |
| 43. | 13 | | 13 | | 14 | | 13 | | 13 | | 14 | | 13 | | 13 | | 13 | |
| 44. | 27 | | 20 | | 27 | | 27 | | 27 | | 20 | | 20 | | 27 | | 27 | |
| 45. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 46. | 34 | | 30 | | 33 | | 33 | | 34 | | 29 | | 30 | | 33 | | 33 | |
| 47. | 18 | | 16 | | 16 | | 16 | | 16 | | 19 | | 16 | | 16 | | 16 | |
| 48. | 23 | | 21 | | 23 | | 23 | | 23 | | 20 | | 21 | | 0 | f | 23 | |
| 49. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 51. | 28 | | 30 | | 33 | | 28 | | 28 | | 0 | f | 30 | | 28 | | 28 | |
| 52. | 26 | | 23 | | 26 | | 21 | | 26 | | 23 | | 23 | | 24 | | 21 | |
| 53. | 27 | | 28 | | 35 | | 28 | | 27 | | 25 | | 25 | | 28 | | 25 | |
| 54. | 28 | | 23 | | 27 | | 24 | | 28 | | 21 | | 23 | | 24 | | 24 | |
| 55. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 56. | 19 | | 16 | | 19 | | 19 | | 19 | | 16 | | 16 | | 19 | | 19 | |
| 57. | 29 | | 29 | | 29 | | 33 | | 29 | | 29 | | 29 | | 33 | | 33 | |
| 58. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 59. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 60. | 22 | | 22 | | 21 | | 26 | | 23 | | 21 | | 22 | | 26 | | 26 | |
| 61. | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | |
| 62. | 12 | | 12 | | 15 | | 12 | | 12 | | 12 | | 12 | | 15 | | 12 | |
| 63. | 7 | | 7 | | 9 | | 7 | | 7 | | 10 | | 7 | | 9 | | 7 | |
| 64. | 26 | | 27 | | 27 | | 27 | | 26 | | 26 | | 27 | | 28 | | 27 | |
| 65. | 66 | | 37 | | 63 | | 37 | | 50 | | 37 | | 37 | | 37 | | 37 | |
| 66. | 14 | | 9 | | 18 | | 14 | | 14 | | 9 | | 9 | | 14 | | 14 | |
| 67. | 12 | | 10 | | 10 | | 10 | | 12 | | 0 | f | 10 | | 10 | | 10 | |
| 68. | 36 | | 36 | | 0 | f | 36 | | 36 | | 36 | | 48 | | 0 | f | 48 | |
| 69. | 20 | | 15 | | 0 | f | 18 | | 20 | | 14 | | 15 | | 24 | | 18 | |
| 70. | 45 | | 35 | | 0 | f | 46 | | 45 | | 35 | | 35 | | 47 | | 46 | |
| 71. | 7 | | 7 | | 9 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 72. | 26 | | 27 | | 23 | | 26 | | 26 | | 23 | | 27 | | 23 | | 26 | |
| 73. | 25 | | 23 | | 34 | | 23 | | 26 | | 25 | | 24 | | 24 | | 24 | |
| 74. | 24 | | 39 | | 24 | | 30 | | 24 | | 23 | | 26 | | 25 | | 30 | |
| 75. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 76. | 28 | | 31 | | 31 | | 31 | | 28 | | 30 | | 31 | | 34 | | 31 | |
| 77. | 14 | | 14 | | 22 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | |
| 78. | 10 | | 10 | | 10 | | 10 | | 10 | | 0 | f | 10 | | 10 | | 10 | |
| 79. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 80. | 17 | | 11 | | 16 | | 16 | | 17 | | 11 | | 11 | | 16 | | 16 | |
| 81. | 11 | | 11 | | 19 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 82. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 83. | 16 | | 16 | | 16 | | 16 | | 19 | | 16 | | 16 | | 16 | | 16 | |
| 84. | 30 | | 28 | | 41 | | 28 | | 30 | | 28 | | 28 | | 34 | | 28 | |
| 85. | 30 | | 24 | | 0 | f | 24 | | 30 | | 24 | | 24 | | 26 | | 24 | |
| 86. | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | |
| 87. | 25 | | 21 | | 25 | | 25 | | 25 | | 21 | | 21 | | 25 | | 25 | |
| 88. | 24 | | 18 | | 28 | | 24 | | 24 | | 17 | | 18 | | 24 | | 24 | |
| 89. | 28 | | 23 | | 34 | | 23 | | 28 | | 23 | | 23 | | 23 | | 23 | |
| 90. | 29 | | 37 | | 27 | | 37 | | 29 | | 37 | | 37 | | 38 | | 37 | |
| 91. | 23 | | 22 | | 23 | | 23 | | 23 | | 22 | | 22 | | 23 | | 23 | |
| 92. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 93. | 47 | | 33 | | 44 | | 43 | | 47 | | 0 | f | 33 | | 40 | | 43 | |
| 94. | 19 | | 26 | | 23 | | 22 | | 23 | | 23 | | 26 | | 26 | | 22 | |
| 95. | 9 | | 9 | | 0 | f | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 96. | 33 | | 23 | | 24 | | 19 | | 33 | | 18 | | 23 | | 19 | | 19 | |
| 97. | 19 | | 20 | | 16 | | 19 | | 19 | | 16 | | 20 | | 16 | | 19 | |
| 98. | 21 | | 21 | | 0 | f | 26 | | 21 | | 20 | | 21 | | 26 | | 26 | |
| 99. | 26 | | 14 | | 26 | | 14 | | 26 | | 14 | | 14 | | 14 | | 14 | |
| 100. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 2022 | | 1883 | | 1914 | | 1972 | | 2016 | | 1657 | | 1874 | | 1875 | | 1982 | |

### A.8.3.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 40 | | 38 | | 39 | | 38 | | 40 | | 40 | | 38 | | 39 | | 38 | |
| 2. | 20 | | 17 | | 20 | | 15 | | 20 | | 17 | | 17 | | 15 | | 15 | |
| 3. | 15 | | 13 | | 13 | | 13 | | 15 | | 13 | | 13 | | 13 | | 13 | |
| 4. | 13 | | 15 | | 13 | | 15 | | 13 | | 15 | | 15 | | 15 | | 15 | |
| 5. | 23 | | 23 | | 24 | | 24 | | 23 | | 23 | | 23 | | 24 | | 24 | |
| 6. | 25 | | 0 | f | 25 | | 31 | | 25 | | 0 | f | 0 | f | 35 | | 29 | |
| 7. | 43 | | 40 | | 43 | | 40 | | 46 | | 43 | | 46 | | 43 | | 46 | |
| 8. | 21 | | 18 | | 22 | | 18 | | 21 | | 18 | | 18 | | 18 | | 18 | |
| 9. | 25 | | 22 | | 25 | | 25 | | 25 | | 22 | | 22 | | 27 | | 25 | |
| 10. | 10 | | 15 | | 14 | | 10 | | 10 | | 14 | | 15 | | 14 | | 10 | |
| 11. | 24 | | 20 | | 22 | | 24 | | 24 | | 0 | f | 20 | | 24 | | 24 | |
| 12. | 18 | | 17 | | 18 | | 18 | | 18 | | 19 | | 17 | | 18 | | 18 | |
| 13. | 11 | | 9 | | 11 | | 9 | | 11 | | 9 | | 9 | | 9 | | 9 | |
| 14. | 31 | | 27 | | 31 | | 31 | | 31 | | 27 | | 27 | | 31 | | 31 | |
| 15. | 22 | | 29 | | 22 | | 29 | | 22 | | 29 | | 29 | | 29 | | 29 | |
| 16. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 17. | 0 | f | 19 | | 0 | f | 26 | | 0 | f | 0 | f | 19 | | 26 | | 26 | |
| 18. | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | | 19 | |
| 19. | 30 | | 31 | | 31 | | 32 | | 31 | | 35 | | 32 | | 32 | | 31 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20. | 16 | | 15 | | 16 | | 16 | | 16 | | 23 | | 15 | | 16 | | 16 | |
| 21. | 42 | | 30 | | 49 | | 42 | | 39 | | 30 | | 30 | | 49 | | 39 | |
| 22. | 9 | | 11 | | 7 | | 7 | | 9 | | 7 | | 11 | | 7 | | 7 | |
| 23. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 24. | 16 | | 14 | | 14 | | 14 | | 16 | | 14 | | 14 | | 14 | | 14 | |
| 25. | 25 | | 23 | | 25 | | 22 | | 25 | | 23 | | 23 | | 0 | f | 22 | |
| 26. | 29 | | 34 | | 30 | | 34 | | 30 | | 0 | f | 34 | | 38 | | 34 | |
| 27. | 15 | | 15 | | 18 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | |
| 28. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 29. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 30. | 12 | | 13 | | 12 | | 12 | | 12 | | 13 | | 13 | | 12 | | 12 | |
| 31. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 32. | 39 | | 46 | | 40 | | 39 | | 39 | | 39 | | 46 | | 40 | | 39 | |
| 33. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 34. | 33 | | 27 | | 33 | | 33 | | 33 | | 27 | | 27 | | 0 | f | 33 | |
| 35. | 34 | | 34 | | 34 | | 34 | | 34 | | 34 | | 34 | | 0 | f | 34 | |
| 36. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 37. | 20 | | 28 | | 20 | | 27 | | 20 | | 0 | f | 28 | | 27 | | 27 | |
| 38. | 27 | | 19 | | 22 | | 27 | | 27 | | 17 | | 19 | | 27 | | 27 | |
| 39. | 17 | | 32 | | 18 | | 16 | | 18 | | 32 | | 19 | | 16 | | 16 | |
| 40. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 41. | 24 | | 28 | | 24 | | 24 | | 24 | | 22 | | 28 | | 24 | | 24 | |
| 42. | 30 | | 24 | | 29 | | 30 | | 30 | | 0 | f | 24 | | 30 | | 30 | |
| 43. | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | |
| 44. | 25 | | 24 | | 25 | | 25 | | 25 | | 24 | | 24 | | 25 | | 25 | |
| 45. | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | | 26 | |
| 46. | 13 | | 13 | | 14 | | 13 | | 13 | | 14 | | 13 | | 14 | | 13 | |
| 47. | 27 | | 20 | | 27 | | 27 | | 27 | | 20 | | 20 | | 27 | | 27 | |
| 48. | 0 | f | 17 | | 0 | f | 17 | | 0 | f | 17 | | 17 | | 17 | | 17 | |
| 49. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 50. | 34 | | 30 | | 33 | | 33 | | 34 | | 29 | | 30 | | 33 | | 33 | |
| 51. | 18 | | 16 | | 16 | | 16 | | 16 | | 19 | | 16 | | 19 | | 16 | |
| 52. | 23 | | 21 | | 23 | | 23 | | 23 | | 20 | | 21 | | 23 | | 23 | |
| 53. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 54. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 55. | 28 | | 30 | | 33 | | 28 | | 28 | | 0 | f | 30 | | 28 | | 28 | |
| 56. | 26 | | 23 | | 26 | | 21 | | 26 | | 23 | | 23 | | 24 | | 21 | |
| 57. | 27 | | 28 | | 35 | | 28 | | 27 | | 25 | | 25 | | 33 | | 25 | |
| 58. | 28 | | 23 | | 27 | | 24 | | 28 | | 21 | | 23 | | 24 | | 24 | |
| 59. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 60. | 19 | | 16 | | 19 | | 19 | | 19 | | 16 | | 16 | | 19 | | 19 | |
| 61. | 29 | | 29 | | 29 | | 33 | | 29 | | 29 | | 29 | | 40 | | 33 | |
| 62. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 63. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 64. | 22 | | 22 | | 21 | | 26 | | 23 | | 21 | | 22 | | 26 | | 26 | |
| 65. | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | | 24 | |
| 66. | 12 | | 12 | | 15 | | 12 | | 12 | | 12 | | 12 | | 15 | | 12 | |
| 67. | 7 | | 7 | | 9 | | 7 | | 7 | | 10 | | 7 | | 9 | | 7 | |
| 68. | 26 | | 27 | | 27 | | 27 | | 26 | | 26 | | 27 | | 28 | | 27 | |
| 69. | 66 | | 37 | | 63 | | 37 | | 50 | | 37 | | 37 | | 0 | f | 37 | |
| 70. | 14 | | 9 | | 18 | | 14 | | 14 | | 9 | | 9 | | 18 | | 14 | |
| 71. | 12 | | 10 | | 10 | | 10 | | 12 | | 0 | f | 10 | | 10 | | 10 | |
| 72. | 36 | | 36 | | 0 | f | 36 | | 36 | | 36 | | 48 | | 0 | f | 48 | |
| 73. | 20 | | 15 | | 0 | f | 18 | | 20 | | 14 | | 15 | | 0 | f | 18 | |
| 74. | 45 | | 35 | | 0 | f | 46 | | 45 | | 35 | | 35 | | 0 | f | 46 | |
| 75. | 7 | | 7 | | 9 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 76. | 26 | | 27 | | 23 | | 26 | | 26 | | 23 | | 27 | | 0 | f | 26 | |
| 77. | 25 | | 23 | | 34 | | 23 | | 26 | | 25 | | 24 | | 0 | f | 24 | |
| 78. | 24 | | 39 | | 24 | | 30 | | 24 | | 23 | | 26 | | 25 | | 30 | |
| 79. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 80. | 28 | | 31 | | 31 | | 31 | | 28 | | 30 | | 31 | | 34 | | 31 | |
| 81. | 14 | | 14 | | 22 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | |
| 82. | 10 | | 10 | | 10 | | 10 | | 10 | | 0 | f | 10 | | 10 | | 10 | |
| 83. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 84. | 17 | | 11 | | 16 | | 16 | | 17 | | 11 | | 11 | | 16 | | 16 | |
| 85. | 11 | | 11 | | 19 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 86. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 6 | | 5 | |
| 87. | 16 | | 16 | | 16 | | 16 | | 19 | | 16 | | 16 | | 0 | f | 16 | |
| 88. | 30 | | 28 | | 41 | | 28 | | 30 | | 28 | | 28 | | 39 | | 28 | |
| 89. | 30 | | 24 | | 0 | f | 24 | | 30 | | 24 | | 24 | | 30 | | 24 | |
| 90. | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | |
| 91. | 25 | | 21 | | 25 | | 25 | | 25 | | 21 | | 21 | | 25 | | 25 | |
| 92. | 24 | | 18 | | 28 | | 24 | | 24 | | 17 | | 18 | | 24 | | 24 | |
| 93. | 28 | | 23 | | 34 | | 23 | | 28 | | 23 | | 23 | | 23 | | 23 | |
| 94. | 29 | | 37 | | 27 | | 37 | | 29 | | 37 | | 37 | | 37 | | 37 | |
| 95. | 23 | | 22 | | 23 | | 23 | | 23 | | 22 | | 22 | | 23 | | 23 | |
| 96. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 97. | 47 | | 33 | | 44 | | 43 | | 47 | | 0 | f | 33 | | 43 | | 43 | |
| 98. | 19 | | 26 | | 23 | | 22 | | 23 | | 23 | | 26 | | 0 | f | 22 | |
| 99. | 9 | | 9 | | 0 | f | 9 | | 9 | | 9 | | 9 | | 13 | | 9 | |
| 100. | 33 | | 23 | | 24 | | 19 | | 33 | | 18 | | 23 | | 19 | | 19 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 1947 | | 1855 | | 1863 | | 1947 | | 1941 | | 1615 | | 1846 | | 1709 | | 1957 | |

## A.9 Problem Sample

### A.9.1 Controlled Problem Sample

The table below lists the problems in this sample. The tasks are numbered down the left hand side and the first line of each problem is the goal condition $G$ and the second is the initial state. So for example task number 1 in this sample is:

$$\begin{aligned}&G = \{at\_door(big\_box, door45, room4), at\_door(box1, door35, room3),\\&\qquad\quad at\_door(box2, door45, room4)\}\\&I = \{open(door67), closed(door57), open(door47), closed(door12),\\&closed(door56), closed(door45), closed(door35), open(door25), closed(door24),\\&open(door23), robot\_in(room1), in\_room(big\_box, room3),\\&in\_room(box2, room4), in\_room(box1, room6)\}\end{aligned}$$

The set of problems that could not be solved by all the planning applications is:

$$\{6, 11, 24, 25, 32, 33, 34, 39, 48, 51, 67, 68, 69, 70, 78, 85, 93, 95, 98\}$$

A sample of 81 problems that could be solved by all of the planning applications was formed by removing this set of tasks from the sample.

|  | Task |
|---|---|
| 1. | at_door(big_box , door45 , room4) , at_door(box1 , door35 , room3) , at_door(box2 , door45 , room4) |
|  | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room6) |
| 2. | robot_at(door25 , room5) , at_door(box1 , door24 , room2) , in_room(big_box , room7) |
|  | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room6) , in_room(big_box , room6) , in_room(box2 , room3) , in_room(box1 , room2) |
| 3. | robot_at(door24 , room4) , closed(door35) |
|  | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room7) |
| 4. | closed(door12) , at_door(box2 , door67 , room7) |
|  | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room5) , in_room(box1 , room1) |
| 5. | at_door(big_box , door23 , room3) , at_door(box2 , door25 , room2) |
|  | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room3) , in_room(box1 , room3) |
| 6. | robot_near(box1) , in_room(box2 , room2) , closed(door67) , in_room(box1 , room2) , closed(door57) |
|  | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room2) , in_room(box1 , room7) |
| 7. | in_room(big_box , room4) , robot_in(room4) , at_door(box1 , door35 , room3) , at_door(box2 , door47 , room4) |
|  | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room7) |
| 8. | in_room(box2 , room2) , robot_in(room1) , in_room(big_box , room3) , at_door(box2 , door23 , room2) |
|  | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room5) |
| 9. | at_door(big_box , door23 , room3) , open(door12) , closed(door23) , in_room(box1 , room6) , robot_at(door57 , room5) |
|  | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room2) |
| 10. | closed(door12) , at_door(box1 , door56 , room5) |
|  | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room3) , in_room(box1 , room3) |
| 11. | closed(door57) , at_door(big_box , door47 , room7) , at_door(box1 , door35 , room5) , robot_at(door24 , room4) |
|  | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room2) |
| 12. | robot_at(door35 , room5) , at_door(big_box , door47 , room4) |
|  | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room6) , in_room(box1 , room3) |
| 13. | closed(door12) , at_door(box2 , door23 , room2) |
|  | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room2) |
| 14. | at_door(box2 , door35 , room5) , open(door47) , robot_at(door67 , room6) , at_door(big_box , door12 , room2) |
|  | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room1) |
| 15. | in_room(box1 , room2) , closed(door12) , robot_in(room6) , at_door(box2 , door23 , room2) , robot_at(door56 , room6) |
|  | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room4) |
| 16. | robot_at(door56 , room5) |
|  | open(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room2) |
| 17. | at_door(box1 , door35 , room5) , at_door(box2 , door56 , room6) , at_door(big_box , door23 , room3) |
|  | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 18. | at_door(box2 , door12 , room2) , at_door(big_box , door57 , room5) , in_room(box1 , room6) |
|  | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room7) , in_room(box1 , room2) |
| 19. | closed(door24) , robot_at(door25 , room5) , closed(door45) , at_door(box2 , door56 , room6) , open(door57) |
|  | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room6) , in_room(box1 , room7) |
| 20. | robot_at(door12 , room1) , at_door(big_box , door57 , room5) , open(door47) , at_door(box1 , door24 , room4) |
|  | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room6) , in_room(box2 , room7) , in_room(box1 , room6) |

| | Task |
|---|---|
| 21. | at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room3) |
| 22. | at_door(box1 , door35 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room2) , in_room(box2 , room7) , in_room(box1 , room5) |
| 23. | at_door(box2 , door56 , room6) , robot_in(room5) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room2) |
| 24. | closed(door24) , in_room(box2 , room6) , at_door(big_box , door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room5) |
| 25. | closed(door56) , at_door(big_box , door24 , room2) , robot_in(room1) , at_door(box2 , door35 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 26. | robot_in(room6) , in_room(box1 , room6) , robot_at(door67 , room6) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room7) , in_room(box1 , room2) |
| 27. | at_door(big_box , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room2) , in_room(big_box , room3) , in_room(box2 , room2) , in_room(box1 , room1) |
| 28. | open(door24) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room4) |
| 29. | at_door(big_box , door57 , room7) , open(door47) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room6) , in_room(box1 , room1) |
| 30. | open(door12) , robot_in(room4) , at_door(box1 , door35 , room5) , at_door(box2 , door56 , room6) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room6) |
| 31. | closed(door25) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room4) , in_room(box1 , room3) |
| 32. | in_room(box2 , room7) , robot_at(door67 , room6) , at_door(big_box , door23 , room3) , at_door(box1 , door25 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room3) , in_room(box1 , room6) |
| 33. | in_room(box1 , room2) , at_door(big_box , door23 , room3) , at_door(box2 , door25 , room2) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room1) , in_room(box2 , room7) , in_room(box1 , room6) |
| 34. | at_door(box2 , door57 , room7) , closed(door67) , in_room(big_box , room5) , in_room(box1 , room1) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room7) , in_room(box1 , room3) |
| 35. | closed(door45) , robot_at(door67 , room6) , at_door(big_box , door25 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room5) |
| 36. | closed(door57) , in_room(box1 , room5) , open(door35) , at_door(box2 , door56 , room6) , closed(door67) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room2) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room5) |
| 37. | at_door(box1 , door57 , room7) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room6) , in_room(box1 , room4) |
| 38. | at_door(box1 , door12 , room2) , open(door56) , at_door(box2 , door45 , room4) , robot_at(door67 , room6) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room5) |
| 39. | at_door(big_box , door57 , room7) , in_room(box2 , room2) , robot_at(door24 , room4) , at_door(box1 , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room5) , in_room(box1 , room7) |
| 40. | at_door(big_box , door57 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room1) , in_room(box2 , room3) , in_room(box1 , room5) |

| | Task |
|---|---|
| 41. | open(door57) , open(door47) , robot_at(door57 , room5) , at_door(box1 , door35 , room3) , at_door(big_box , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room6) |
| 42. | in_room(box2 , room3) , at_door(big_box , door45 , room5) , at_door(box1 , door25 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room1) , in_room(box2 , room2) , in_room(box1 , room7) |
| 43. | at_door(box1 , door56 , room6) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room1) |
| 44. | at_door(box2 , door67 , room7) , in_room(box2 , room7) , at_door(box1 , door67 , room6) , robot_at(door23 , room3) , at_door(big_box , door67 , room7) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room6) |
| 45. | at_door(box1 , door12 , room1) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room1) |
| 46. | at_door(box2 , door56 , room6) , robot_at(door24 , room4) , at_door(box1 , door67 , room7) , at_door(big_box , door47 , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room3) , in_room(big_box , room1) , in_room(box2 , room6) , in_room(box1 , room7) |
| 47. | at_door(box1 , door12 , room1) , in_room(box2 , room3) , robot_in(room1) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room1) |
| 48. | in_room(box1 , room2) , robot_at(door23 , room3) , at_door(box2 , door56 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room3) , in_room(box2 , room7) , in_room(box1 , room1) |
| 49. | robot_at(door24 , room4) , closed(door23) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room6) |
| 50. | robot_in(room4) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room1) |
| 51. | in_room(big_box , room1) , at_door(box1 , door57 , room7) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room7) , in_room(box2 , room1) , in_room(box1 , room6) |
| 52. | at_door(box2 , door47 , room7) , open(door57) , open(door45) , in_room(box1 , room2) , robot_at(door45 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room6) , in_room(box2 , room3) , in_room(box1 , room7) |
| 53. | in_room(big_box , room4) , at_door(box2 , door12 , room2) , in_room(box1 , room7) , robot_at(door23 , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room7) , in_room(box1 , room1) |
| 54. | closed(door23) , at_door(box2 , door47 , room7) , robot_at(door24 , room4) , at_door(box1 , door12 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room6) , in_room(box1 , room5) |
| 55. | robot_at(door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room7) |
| 56. | at_door(box2 , door45 , room5) , robot_at(door24 , room4) , open(door35) , at_door(big_box , door12 , room1) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room5) |
| 57. | closed(door23) , at_door(big_box , door57 , room5) , at_door(box2 , door56 , room6) , robot_at(door35 , room5) , at_door(box1 , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room5) , in_room(box2 , room2) , in_room(box1 , room7) |
| 58. | robot_at(door56 , room6) |
| | open(door67) , open(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room7) |
| 59. | robot_at(door35 , room5) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room7) |
| 60. | open(door57) , in_room(box1 , room2) , at_door(box2 , door35 , room5) , robot_at(door67 , room6) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room4) |

| | Task |
|---|---|
| 61. | at_door(box1 , door23 , room2) , robot_in(room1) , at_door(big_box , door12 , room2) , at_door(box2 , door56 , room5) , in_room(box1 , room2) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room2) , in_room(box2 , room2) , in_room(box1 , room1) |
| 62. | at_door(big_box , door57 , room5) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room1) |
| 63. | robot_in(room6) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room6) |
| 64. | in_room(box1 , room5) , closed(door12) , at_door(box2 , door67 , room7) , at_door(box1 , door35 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room6) |
| 65. | at_door(big_box , door56 , room5) , robot_in(room1) , closed(door25) , in_room(box2 , room3) , in_room(box1 , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room1) , in_room(box2 , room7) , in_room(box1 , room7) |
| 66. | robot_at(door67 , room6) , at_door(box2 , door25 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room2) |
| 67. | robot_at(door67 , room6) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room4) |
| 68. | closed(door47) , robot_in(room1) , at_door(box1 , door35 , room3) , at_door(box2 , door23 , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room7) , in_room(box1 , room7) |
| 69. | at_door(box2 , door56 , room5) , robot_at(door45 , room4) , at_door(box1 , door35 , room5) , open(door12) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room5) , in_room(box1 , room7) |
| 70. | robot_at(door24 , room4) , at_door(box2 , door56 , room6) , at_door(big_box , door25 , room5) , at_door(box1 , door12 , room2) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room1) , in_room(big_box , room2) , in_room(box2 , room3) , in_room(box1 , room6) |
| 71. | closed(door12) , at_door(big_box , door35 , room3) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room5) |
| 72. | at_door(box1 , door57 , room7) , at_door(box2 , door56 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room4) , in_room(box2 , room1) , in_room(box1 , room1) |
| 73. | robot_in(room3) , in_room(box1 , room2) , closed(door12) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room1) , in_room(box2 , room6) , in_room(box1 , room2) |
| 74. | closed(door12) , at_door(box2 , door56 , room6) , at_door(box1 , door35 , room5) , open(door24) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room5) |
| 75. | open(door24) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room7) |
| 76. | at_door(box2 , door35 , room5) , open(door47) , robot_in(room1) , at_door(big_box , door57 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room7) |
| 77. | robot_in(room7) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room3) |
| 78. | in_room(box1 , room2) , at_door(box1 , door25 , room2) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room2) , in_room(box1 , room4) |
| 79. | robot_in(room1) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room5) , in_room(box1 , room4) |
| 80. | in_room(box1 , room6) , robot_at(door24 , room4) , at_door(big_box , door56 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room4) , in_room(box1 , room6) |

| | Task |
|---|---|
| 81. | at_door(box2 , door23 , room2) , at_door(box1 , door35 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room2) |
| 82. | robot_at(door24 , room4) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room6) , in_room(box2 , room2) , in_room(box1 , room1) |
| 83. | robot_in(room5) , at_door(box1 , door35 , room5) , closed(door23) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room2) |
| 84. | in_room(big_box , room1) , closed(door12) , at_door(box1 , door24 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room6) |
| 85. | closed(door23) , in_room(big_box , room2) , robot_in(room3) , at_door(box1 , door35 , room3) , at_door(box2 , door23 , room2) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room4) |
| 86. | robot_in(room1) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room1) , in_room(box1 , room1) |
| 87. | at_door(big_box , door23 , room2) , robot_at(door24 , room4) , open(door57) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room1) , in_room(box1 , room2) |
| 88. | at_door(box1 , door45 , room4) , robot_at(door57 , room7) , open(door23) , closed(door12) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room5) |
| 89. | at_door(box2 , door45 , room5) , closed(door23) , in_room(box1 , room2) , robot_in(room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room4) |
| 90. | at_door(box1 , door12 , room2) , at_door(big_box , door57 , room7) , at_door(box2 , door12 , room2) , robot_in(room1) , closed(door23) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room2) |
| 91. | at_door(box1 , door56 , room6) , in_room(box1 , room6) , at_door(box2 , door23 , room2) , at_door(big_box , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room2) , in_room(box2 , room2) , in_room(box1 , room2) |
| 92. | robot_in(room1) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 93. | at_door(box2 , door23 , room3) , in_room(box1 , room2) , at_door(big_box , door23 , room2) , robot_at(door67 , room6) , closed(door23) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room7) |
| 94. | closed(door12) , at_door(box2 , door56 , room6) , closed(door45) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room2) |
| 95. | robot_at(door12 , room1) , closed(door12) , closed(door23) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 96. | robot_at(door67 , room6) , in_room(box1 , room2) , at_door(big_box , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room5) |
| 97. | closed(door12) , at_door(box2 , door23 , room3) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room3) |
| 98. | at_door(big_box , door57 , room5) , at_door(box2 , door23 , room3) , closed(door47) , closed(door25) , robot_at(door24 , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room5) , in_room(big_box , room2) , in_room(box2 , room5) , in_room(box1 , room1) |
| 99. | at_door(big_box , door24 , room2) , open(door67) , open(door35) , robot_at(door67 , room6) , in_room(box1 , room2) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room2) , in_room(box2 , room6) , in_room(box1 , room4) |
| 100. | at_door(box2 , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room1) , in_room(big_box , room2) , in_room(box2 , room3) , in_room(box1 , room2) |

### A.9.2 Uncontrolled Problem Sample

| | Task |
|---|---|
| 1. | at_door(big_box , door45 , room4) , at_door(box1 , door35 , room3) , at_door(box2 , door45 , room4) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room6) |
| 2. | robot_at(door25 , room5) , at_door(box1 , door24 , room2) , in_room(big_box , room7) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room6) , in_room(big_box , room6) , in_room(box2 , room3) , in_room(box1 , room2) |
| 3. | robot_at(door24 , room4) , closed(door35) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room7) |
| 4. | closed(door12) , at_door(box2 , door67 , room7) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room5) , in_room(box1 , room1) |
| 5. | at_door(big_box , door23 , room3) , at_door(box2 , door25 , room2) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room3) , in_room(box1 , room3) |
| 6. | robot_near(box1) , in_room(box2 , room2) , closed(door67) , in_room(box1 , room2) , closed(door57) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room2) , in_room(box1 , room7) |
| 7. | in_room(big_box , room4) , robot_in(room4) , at_door(box1 , door35 , room3) , at_door(box2 , door47 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room7) |
| 8. | in_room(box2 , room2) , robot_in(room1) , in_room(big_box , room3) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room5) |
| 9. | at_door(big_box , door23 , room3) , open(door12) , closed(door23) , in_room(box1 , room6) , robot_at(door57 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room2) |
| 10. | closed(door12) , at_door(box1 , door56 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room3) , in_room(box1 , room3) |
| 11. | closed(door57) , at_door(big_box , door47 , room7) , at_door(box1 , door35 , room5) , robot_at(door24 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room2) |
| 12. | robot_at(door35 , room5) , at_door(big_box , door47 , room4) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room6) , in_room(box1 , room3) |
| 13. | closed(door12) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room2) |
| 14. | at_door(box2 , door35 , room5) , open(door47) , robot_at(door67 , room6) , at_door(big_box , door12 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room1) |
| 15. | in_room(box1 , room2) , closed(door12) , robot_in(room6) , at_door(box2 , door23 , room2) , robot_at(door56 , room6) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room4) |
| 16. | robot_at(door56 , room5) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room2) |
| 17. | open(door35) , closed(door24) , robot_in(room1) , robot_at(door12 , room1) , at_door(big_box , door25 , room2) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room2) , in_room(box2 , room2) , in_room(box1 , room2) |
| 18. | at_door(box1 , door35 , room5) , at_door(box2 , door56 , room6) , at_door(big_box , door23 , room3) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 19. | at_door(box2 , door12 , room2) , at_door(big_box , door57 , room5) , in_room(box1 , room6) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room7) , in_room(box1 , room2) |
| 20. | closed(door24) , robot_at(door25 , room5) , closed(door45) , at_door(box2 , door56 , room6) , open(door57) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room6) , in_room(box1 , room7) |

| | Task |
|---|---|
| 21. | robot_at(door12 , room1) , at_door(big_box , door57 , room5) , open(door47) , at_door(box1 , door24 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 22. | at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room3) |
| 23. | at_door(box1 , door35 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room2) , in_room(box2 , room7) , in_room(box1 , room5) |
| 24. | at_door(box2 , door56 , room6) , robot_in(room5) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room2) |
| 25. | closed(door24) , in_room(box2 , room6) , at_door(big_box , door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room5) |
| 26. | closed(door56) , at_door(big_box , door24 , room2) , robot_in(room1) , at_door(box2 , door35 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 27. | robot_in(room6) , in_room(box1 , room6) , robot_at(door67 , room6) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room5) , in_room(box2 , room7) , in_room(box1 , room2) |
| 28. | at_door(big_box , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room2) , in_room(big_box , room3) , in_room(box2 , room2) , in_room(box1 , room1) |
| 29. | open(door24) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room4) |
| 30. | at_door(big_box , door57 , room7) , open(door47) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room6) , in_room(box1 , room1) |
| 31. | at_door(big_box , door56 , room5) , robot_in(room5) , closed(door12) , robot_near(box1) , at_door(box1 , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room5) , in_room(big_box , room5) , in_room(box2 , room3) , in_room(box1 , room3) |
| 32. | open(door12) , robot_in(room4) , at_door(box1 , door35 , room5) , at_door(box2 , door56 , room6) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room6) |
| 33. | closed(door25) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room4) , in_room(box1 , room3) |
| 34. | in_room(box2 , room7) , robot_at(door67 , room6) , at_door(big_box , door23 , room3) , at_door(box1 , door25 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room3) , in_room(box1 , room6) |
| 35. | in_room(box1 , room2) , at_door(big_box , door23 , room3) , at_door(box2 , door25 , room2) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room1) , in_room(box2 , room7) , in_room(box1 , room6) |
| 36. | robot_near(box2) , at_door(big_box , door24 , room2) , at_door(box2 , door23 , room2) , open(door67) , robot_in(room1) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room6) , in_room(box2 , room2) , in_room(box1 , room1) |
| 37. | at_door(box2 , door57 , room7) , closed(door67) , in_room(big_box , room5) , in_room(box1 , room1) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room3) , in_room(box2 , room7) , in_room(box1 , room3) |
| 38. | closed(door45) , robot_at(door67 , room6) , at_door(big_box , door25 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room5) |
| 39. | closed(door57) , in_room(box1 , room5) , open(door35) , at_door(box2 , door56 , room6) , closed(door67) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room2) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room5) |
| 40. | at_door(box1 , door57 , room7) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room6) , in_room(box1 , room4) |

| | Task |
|---|---|
| 41. | at_door(box1 , door12 , room2) , open(door56) , at_door(box2 , door45 , room4) , robot_at(door67 , room6) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room5) |
| 42. | at_door(big_box , door57 , room7) , in_room(box2 , room2) , robot_at(door24 , room4) , at_door(box1 , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room5) , in_room(box1 , room7) |
| 43. | at_door(big_box , door57 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room1) , in_room(box2 , room3) , in_room(box1 , room5) |
| 44. | open(door57) , open(door47) , robot_at(door57 , room5) , at_door(box1 , door35 , room3) , at_door(big_box , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room2) , in_room(box1 , room6) |
| 45. | in_room(box2 , room3) , at_door(big_box , door45 , room5) , at_door(box1 , door25 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room1) , in_room(box2 , room2) , in_room(box1 , room7) |
| 46. | at_door(box1 , door56 , room6) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room6) , in_room(box1 , room1) |
| 47. | at_door(box2 , door67 , room7) , in_room(box2 , room7) , at_door(box1 , door67 , room6) , robot_at(door23 , room3) , at_door(big_box , door67 , room7) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room6) |
| 48. | closed(door24) , closed(door57) , open(door35) , robot_at(door35 , room5) , at_door(box2 , door23 , room3) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room4) , in_room(big_box , room6) , in_room(box2 , room2) , in_room(box1 , room6) |
| 49. | at_door(box1 , door12 , room1) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room7) , in_room(box2 , room5) , in_room(box1 , room1) |
| 50. | at_door(box2 , door56 , room6) , robot_at(door24 , room4) , at_door(box1 , door67 , room7) , at_door(big_box , door47 , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room3) , in_room(big_box , room1) , in_room(box2 , room6) , in_room(box1 , room7) |
| 51. | at_door(box1 , door12 , room1) , in_room(box2 , room3) , robot_in(room1) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room1) |
| 52. | in_room(box1 , room2) , robot_at(door23 , room3) , at_door(box2 , door56 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room3) , in_room(box2 , room7) , in_room(box1 , room1) |
| 53. | robot_at(door24 , room4) , closed(door23) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room6) |
| 54. | robot_in(room4) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room1) |
| 55. | in_room(big_box , room1) , at_door(box1 , door57 , room7) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room7) , in_room(box2 , room1) , in_room(box1 , room6) |
| 56. | at_door(box2 , door47 , room7) , open(door57) , open(door45) , in_room(box1 , room2) , robot_at(door45 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room6) , in_room(box2 , room3) , in_room(box1 , room7) |
| 57. | in_room(big_box , room4) , at_door(box2 , door12 , room2) , in_room(box1 , room7) , robot_at(door23 , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room7) , in_room(box1 , room1) |
| 58. | closed(door23) , at_door(box2 , door47 , room7) , robot_at(door24 , room4) , at_door(box1 , door12 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room6) , in_room(box1 , room5) |
| 59. | robot_at(door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room7) |
| 60. | at_door(box2 , door45 , room5) , robot_at(door24 , room4) , open(door35) , at_door(big_box , door12 , room1) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room5) |

| | Task |
|---|---|
| 61. | closed(door23) , at_door(big_box , door57 , room5) , at_door(box2 , door56 , room6) , robot_at(door35 , room5) , at_door(box1 , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room5) , in_room(box2 , room2) , in_room(box1 , room7) |
| 62. | robot_at(door56 , room6) |
| | open(door67) , open(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room7) |
| 63. | robot_at(door35 , room5) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room7) |
| 64. | open(door57) , in_room(box1 , room2) , at_door(box2 , door35 , room5) , robot_at(door67 , room6) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room5) , in_room(box2 , room4) , in_room(box1 , room4) |
| 65. | at_door(box1 , door23 , room2) , robot_in(room1) , at_door(big_box , door12 , room2) , at_door(box2 , door56 , room5) , in_room(box1 , room2) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room2) , in_room(box2 , room2) , in_room(box1 , room1) |
| 66. | at_door(big_box , door57 , room5) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room4) , in_room(box2 , room5) , in_room(box1 , room1) |
| 67. | robot_in(room6) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room6) |
| 68. | in_room(box1 , room5) , closed(door12) , at_door(box2 , door67 , room7) , at_door(box1 , door35 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room6) |
| 69. | at_door(big_box , door56 , room5) , robot_in(room1) , closed(door25) , in_room(box2 , room3) , in_room(box1 , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room1) , in_room(box2 , room7) , in_room(box1 , room7) |
| 70. | robot_at(door67 , room6) , at_door(box2 , door25 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room2) |
| 71. | robot_at(door67 , room6) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room4) , in_room(box2 , room4) , in_room(box1 , room4) |
| 72. | closed(door47) , robot_in(room1) , at_door(box1 , door35 , room3) , at_door(box2 , door23 , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room7) , in_room(big_box , room2) , in_room(box2 , room7) , in_room(box1 , room7) |
| 73. | at_door(box2 , door56 , room5) , robot_at(door45 , room4) , at_door(box1 , door35 , room5) , open(door12) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room6) , in_room(box2 , room5) , in_room(box1 , room7) |
| 74. | robot_at(door24 , room4) , at_door(box2 , door56 , room6) , at_door(big_box , door25 , room5) , at_door(box1 , door12 , room2) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room1) , in_room(big_box , room2) , in_room(box2 , room3) , in_room(box1 , room6) |
| 75. | closed(door12) , at_door(big_box , door35 , room3) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room5) |
| 76. | at_door(box1 , door57 , room7) , at_door(box2 , door56 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room4) , in_room(box2 , room1) , in_room(box1 , room1) |
| 77. | robot_in(room3) , in_room(box1 , room2) , closed(door12) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room1) , in_room(box2 , room6) , in_room(box1 , room2) |
| 78. | closed(door12) , at_door(box2 , door56 , room6) , at_door(box1 , door35 , room5) , open(door24) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room5) |
| 79. | open(door24) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room7) |
| 80. | at_door(box2 , door35 , room5) , open(door47) , robot_in(room1) , at_door(big_box , door57 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room4) , in_room(box2 , room2) , in_room(box1 , room7) |

| | Task |
|---|---|
| 81. | robot_in(room7) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room1) , in_room(box2 , room5) , in_room(box1 , room3) |
| 82. | in_room(box1 , room2) , at_door(box1 , door25 , room2) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room2) , in_room(box1 , room4) |
| 83. | robot_in(room1) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room5) , in_room(box1 , room4) |
| 84. | in_room(box1 , room6) , robot_at(door24 , room4) , at_door(big_box , door56 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room2) , in_room(big_box , room2) , in_room(box2 , room4) , in_room(box1 , room6) |
| 85. | at_door(box2 , door23 , room2) , at_door(box1 , door35 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room4) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room2) |
| 86. | robot_at(door24 , room4) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room3) , in_room(big_box , room6) , in_room(box2 , room2) , in_room(box1 , room1) |
| 87. | robot_in(room5) , at_door(box1 , door35 , room5) , closed(door23) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room3) , in_room(big_box , room4) , in_room(box2 , room6) , in_room(box1 , room2) |
| 88. | in_room(big_box , room1) , closed(door12) , at_door(box1 , door24 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room6) |
| 89. | closed(door23) , in_room(big_box , room2) , robot_in(room3) , at_door(box1 , door35 , room3) , at_door(box2 , door23 , room2) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , robot_in(room4) , in_room(big_box , room2) , in_room(box2 , room1) , in_room(box1 , room4) |
| 90. | robot_in(room1) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room2) , in_room(big_box , room5) , in_room(box2 , room1) , in_room(box1 , room1) |
| 91. | at_door(big_box , door23 , room2) , robot_at(door24 , room4) , open(door57) , at_door(box2 , door23 , room2) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room1) , in_room(box2 , room1) , in_room(box1 , room2) |
| 92. | at_door(box1 , door45 , room4) , robot_at(door57 , room7) , open(door23) , closed(door12) , at_door(box2 , door23 , room2) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room7) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room5) |
| 93. | at_door(box2 , door45 , room5) , closed(door23) , in_room(box1 , room2) , robot_in(room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , robot_in(room7) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room4) |
| 94. | at_door(box1 , door12 , room2) , at_door(big_box , door57 , room7) , at_door(box2 , door12 , room2) , robot_in(room1) , closed(door23) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room7) , in_room(box2 , room4) , in_room(box1 , room2) |
| 95. | at_door(box1 , door56 , room6) , in_room(box1 , room6) , at_door(box2 , door23 , room2) , at_door(big_box , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room1) , in_room(big_box , room2) , in_room(box2 , room2) , in_room(box1 , room2) |
| 96. | robot_in(room1) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , robot_in(room3) , in_room(big_box , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 97. | at_door(box2 , door23 , room3) , in_room(box1 , room2) , at_door(big_box , door23 , room2) , robot_at(door67 , room6) , closed(door23) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room1) , in_room(big_box , room3) , in_room(box2 , room3) , in_room(box1 , room7) |
| 98. | closed(door12) , at_door(box2 , door56 , room6) , closed(door45) , at_door(big_box , door57 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , robot_in(room5) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room2) |
| 99. | robot_at(door12 , room1) , closed(door12) , closed(door23) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , robot_in(room6) , in_room(big_box , room3) , in_room(box2 , room4) , in_room(box1 , room4) |
| 100. | robot_at(door67 , room6) , in_room(box1 , room2) , at_door(big_box , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , robot_in(room5) , in_room(big_box , room6) , in_room(box2 , room1) , in_room(box1 , room5) |

## A.10 Comparison of ALPINE and ABGEN hierarchies

ABGEN is our adaptation of the ALPINE algorithm of [36]. Direct comparison of the abstraction hierarchies generated by ABGEN with those generated by ALPINE is not straightforward since the operator representation used by the PRODIGY system, for which ALPINE is specialised, is different to *MEA*, the example DIP that we have used. For example, PRODIGY accepts negated literals in the operator pre-conditions whereas *MEA* does not.

So we have been unable to compare hierarchies generated by ABGEN with those generated by ALPINE on the same modelled planning domains that ALPINE was tested on. However, for two different representations of the STRIPS robot planning *SW* we can show that the two methods generate similar hierarchies.

This is illustrated for the following planning goal:

$$\{in\_room(box1, room1), closed(door56), closed(door12), robot\_in(room3), \\ in\_room(box2, room3)\}$$

Below we show example hierarchies that are generated for this goal. The ALPINE hierarchy is taken from an example in [36, pp 107] but the representation has been changed slightly. The first change is that predicates given in the example in [36] have been re-represented. The table below shows the correspondence between the way predicates used by the different methods are expressed.

| ABGEN | ALPINE |
|---|---|
| in_room(B,R) | (inroom box room) |
| robot_in(R) | (inroom robot room) |
| closed(D) | (status door status) |
| open(D) | (status door status) |
| at_door(B,D,R) | (next-to box door) |
| robot_at(D,R) | (next-to robot door) |
| next_to(B,C) | (next-to box box) |
| robot_near(B) | (next-to robot box) |

The second change is that all unchanging predicates have been removed from the compilation. For example, the truth value of predicates such as the one given below cannot be changed by *MEA* operators and are not included in ABGEN hierarchies.

$connects(Room, Room, Door)$

The two example hierarchies are shown in the table:

| | ABGEN | ALPINE |
|---|---|---|
| 1. | in_room(B,R) | in_room(B,R) |
| 2. | robot_in(R) | robot_in(R), closed(D), open(D) |
| 3. | at_door(B,D,R) | at_door(B, D, R), robot_at(D, R), next_to(B, C), robot _ near(B) |
| 4. | closed(D),open(D) | |
| 5. | next_ to(B, C), robot_ near( B), robot_at(D, R) | |

Although the hierarchies are not identical they are very similar. For instance, the predicate $in\_room(B, R)$ is in the highest level of the hierarchy (level 1) in both hierarchies. Also, the predicates $next\_to(B, C)$, $robot\_near(B)$ and $robot\_at(D, R)$ are in the lowest level in the ABGEN and ALPINE hierarchies, level 5 and level 3 respectively. The only observable differences in the hierarchies are that the ABGEN hierarchy has more levels and that ABGEN places predicate $at\_door(B, D, R)$ higher, than the predicates describing the status of the door, than does ALPINE. This difference can be accounted for by a combination of the differences in the representation of the domain, especially operator representation and in the way that the domain has been modelled.

# Appendix B

# $_{SW}2$: The Person World

This planning domain is similar to the STRIPS-world as reviewed in appendix A but with two robots (ann and albert) who both have the capability of the single robot. Typical solutions to tasks in this modelled planning domain combine the resources of ann and albert, and a considerable increase in complexity arises from the reasoning

A representation of this domain is shown in the figure below. As for $SW$ the modelled planning domain consists of boxes, a series of six rooms with doors between them that can be opened or closed. Doors are numbered from the rooms that they join, where the numbers are arranged in ascending order.

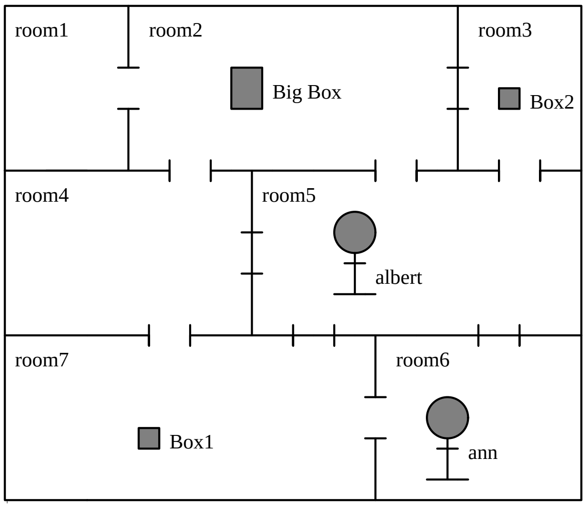


Figure B.1: $SW^2$: The Person World

The state of the world that is shown in the figure is represented in the FM input language as follows:

$$\{open(door67), closed(door57), open(door47), open(door12), closed(door56),$$
$$closed(door45), open(door35), open(door25), open(door24), closed(door23),$$
$$in\_room(big\_box, room2), in(albert, room5), in(ann, room6),$$
$$in\_room(box2, room3), in\_room(box1, room7)\}$$

The table below gives information on the relative size and structure of this world.

Column **ops** gives the number of operator instances, **preds** the number of predicate instances, and **states** the approximate number of world states. The final column **rules** gives the combined number of parameterised rule and axiom schemas that were posed by the modeller to specify relations between literals, and to specify inconsistent states.

| | **ops** | **preds** | **states** | **rules** |
|---|---|---|---|---|
| $SW^2$: person | 414 | 240 | $10^9$ | 27 |

## B.1 *OS*: Domain Operators

The set of domain operators are defined as follows:

$(gothrudoor(Agent, D, R),$
$\{\},$
$\{connect(R1, R, D), type\_of(Agent, agent)\},$
$\{at(Agent, D, R1), open(D)\},$
$\{in(Agent, R)\},$
$\{at(Agent, D, R)\},$
$\{at(Agent, D, R1), near(Agent, \_), in(Agent, R1)\})$

$(gotodoor(Agent, D, R),$
$\{\},$
$\{connect(R, \_, D), type\_of(Agent, agent)\},$
$\{in(Agent, R)\},$
$\{at(Agent, D, R)\},$
$\{\},$
$\{at(Agent, \_, R), near(Agent, \_)\})$

$(pushthrudoor(Agent, Ob, D, To\_room),$
$\{\},$
$\{fits\_thru(Ob, D), type\_of(Agent, agent), connect(R, To\_room, D)\},$
$\{at\_door(Ob, D, R), in\_room(Ob, R), near(Agent, Ob), open(D)\},$
$\{in\_room(Ob, To\_room)\},$
$\{in(Agent, To\_room), near(Agent, Ob)\},$
$\{at(Agent, D, R), at\_door(Ob, D, R), near(\_, Ob), near(Agent, \_),$
$next\_to(Ob, \_), next\_to(\_, Ob), in(Agent, R), in\_room(Ob, R)\})$

$(pushtodoor(Agent, Ob, D, R),$
$\{\},$
$\{connect(R, \_, D), type\_of(Agent, agent), type\_of(Ob, box)\},$
$\{in\_room(Ob, R), in(Agent, R), near(Agent, Ob)\},$
$\{at\_door(Ob, D, R)\},$

$\{near(Agent, Ob)\}$,
$\{at\_door(Ob, \_, R), near(\_, Ob), near(Agent, \_), next\_to(Ob, \_),$
$next\_to(\_, Ob)\})$

$(pushtobox(Agent, Ob1, Ob2),$
$\{\}$,
$\{ne(Ob1, Ob2), type\_of(Agent, agent), type\_of(Ob1, box),$
$type\_of(Ob2, box)\}$,
$\{near(Agent, Ob1), in\_room(Ob2, R), in\_room(Ob1, R)\}$,
$\{next\_to(Ob1, Ob2)\}$,
$\{near(Agent, Ob1)\}$,
$\{at(Agent, \_, R), at\_door(Ob1, \_, R), near(Agent, \_), near(\_, Ob),$
$next\_to(Ob1, \_), next\_to(\_, Ob1)\})$

$(goto(Agent, X, R),$
$\{in\_room(X, R)\}$,
$\{type\_of(X, box), type\_of(R, room), type\_of(Agent, agent)\}$,
$\{in\_room(X, R), in(Agent, R)\}$,
$\{near(Agent, X)\}$,
$\{\}$,
$\{at(Agent, \_, R), near(Agent, \_)\})$

$(open(Agent, D),$
$\{\}$,
$\{connect(R, \_, D), type\_of(Agent, agent)\}$,
$\{closed(D), at(Agent, D, R)\}$,
$\{open(D)\}$,
$\{\}$,
$\{closed(D), near(Agent, \_)\})$

$(close(Agent, D),$
$\{\}$,
$\{connect(R, \_, D), type\_of(Agent, agent)\}$,
$\{open(D), at(Agent, D, R)\}$,
$\{closed(D)\}$,
$\{\}$,
$\{open(D), near(Agent, \_)\})$

## B.2 *ES*: Environmental Information

The set of environmental information, *ES*, for this domain is given below. For ease of presentation in this section we have divided it into sub-sets. Below is the set of unchanging predicates and later are rules that specify relations between predicates in the domain model.

$$\{type\_of(ann, agent), type\_of(albert, agent), type\_of(room1, room),$$
$$type\_of(room2, room), type\_of(room3, room), type\_of(room4, room),$$
$$type\_of(room5, room), type\_of(room6, room), type\_of(room7, room),$$
$$type\_of(door23, door), type\_of(door24, door), type\_of(door25, door),$$
$$type\_of(door35, door), type\_of(door45, door), type\_of(door56, door),$$
$$type\_of(door12, door), type\_of(door47, door), type\_of(door57, door),$$
$$type\_of(door67, door), type\_of(box2, box), type\_of(big\_box, box),$$
$$type\_of(box1, box), connect(room2, room3, door23),$$
$$connect(room3, room2, door23), connect(room2, room4, door24),$$
$$connect(room4, room2, door24), connect(room2, room5, door25),$$
$$connect(room5, room2, door25), connect(room6, room5, door56),$$
$$connect(room5, room6, door56), connect(room5, room3, door35),$$
$$connect(room3, room5, door35), connect(room5, room4, door45),$$
$$connect(room4, room5, door45), connect(room4, room7, door47),$$
$$connect(room7, room4, door47), connect(room5, room7, door57),$$
$$connect(room7, room5, door57), connect(room6, room7, door67),$$
$$connect(room7, room6, door67), connect(room1, room2, door12),$$
$$connect(room2, room1, door12), fits\_thru(box1, door24),$$
$$fits\_thru(box1, door23), fits\_thru(box1, door25), fits\_thru(box1, door56),$$
$$fits\_thru(box1, door45), fits\_thru(box1, door35), fits\_thru(box1, door47),$$
$$fits\_thru(box1, door57), fits\_thru(box1, door12), fits\_thru(box1, door67),$$
$$fits\_thru(box2, door24), fits\_thru(box2, door23), fits\_thru(box2, door25),$$
$$fits\_thru(box2, door56), fits\_thru(box2, door45), fits\_thru(box2, door35),$$
$$fits\_thru(box2, door47), fits\_thru(box2, door57), fits\_thru(box2, door12),$$
$$fits\_thru(box2, door67), fits\_thru(big\_box, door24),$$
$$fits\_thru(big\_box, door25), fits\_thru(big\_box, door56),$$
$$fits\_thru(big\_box, door35), fits\_thru(big\_box, door47),$$
$$fits\_thru(big\_box, door57), fits\_thru(big\_box, door12),$$
$$fits\_thru(big\_box, door67)\}$$

*ES* also includes rules which specify the relation between certain predicates in the domain. Each rule is expressed as a pair $(a, b)$ where $b$ can be inferred from $a$. For this modelled planning domain the rules for achievable predicates (as defined on page 15) are:

$$\{(at\_door(O, \_, R), in\_room(O, R)),$$
$$(\{next\_to(O1, O2), in\_room(O1, R)\}, in\_room(O2, R)),$$
$$(\{next\_to(O2, O1), in\_room(O1, R)\}, in\_room(O2, R)),$$
$$(at(Ag, D, R), in(Ag, R)),$$
$$(\{near(Ag, B), in\_room(B, R)\}, in(Ag, R))\}$$

and the rules for unchanging predicates are:

$$\{(connect(X, Y, \_), ne(X, Y)),$$
$$(connect(\_, Y5, Z5), ne(Y5, Z5)),$$
$$(connect(X4, \_, Z4), ne(X4, Z4)),$$
$$(connect(X1, Y1, Z1), connect(Y1, X1, Z1))\}$$

## B.3 $IN$: Negative State Invariant

The negative state invariant for this modelled planning domain was supplied by the domain modeller. The set $IN$ for this domain is:

$$\{(open(D), closed(D)),$$
$$(near(Ag, \_), at(Ag, \_, \_)),$$
$$(at\_door(\_, D, \_), at(Ag, D, \_)),$$
$$(in(Ag, D), in(Ag, E)),$$
$$(near(Ag, D), near(Ag, E)),$$
$$(at\_door(B, Y, \_), at\_door(B, X, \_)),$$
$$(at\_door(B, X, Proom), at\_door(B, X, Qroom)),$$
$$(at\_door(B, \_, X), in\_room(B, Y)),$$
$$(in\_room(X, Y), in\_room(X, Z)),$$
$$(at(Ag, \_, Y), at(Ag, \_, Z)),$$
$$(at(Ag, \_, Y), in(Ag, Z)),$$
$$(at(Ag, Z, X), at(Ag, Y, X)),$$
$$(open(D), closed(D)),$$
$$(next\_to(B, \_), at\_door(B, \_, \_)),$$
$$(next\_to(\_, B), at\_door(B, \_, \_))\}$$

## B.4 Positive State Invariant: $IP_{sort}$

From the environmental information, $ES$, for this modelled planning we have the following sorts:

$$sorts = \{box, robot, door, room\}$$

and each sort is a set of objects of that sort, as follows:

$$box = \{box1, box2, big\_box\}$$
$$door = \{door12, door23, door24, door25, door35, door45, door47, door57, door67\}$$
$$room = \{room1, room2, room3, room4, room5, room6, room7\}$$
$$robot = \{ann, albert\}$$

The user supplied positive state invariant for this domain is as follows:

$$
\begin{aligned}
&IP_{box} = \{\\
&\quad \{in\_room(Box, Room), at\_door(Box, Door, Room)\},\\
&\quad \{in\_room(Box, Room), next\_to(Box, Box1)\},\\
&\quad \{in\_room(Box, Room)\}\}
\end{aligned}
$$

$$
\begin{aligned}
&IP_{robot} = \{\\
&\quad \{in(Agent, Room), at(Agent, Door, Room)\},\\
&\quad \{in(Agent, Room), near(Agent, Box)\},\\
&\quad \{in(Agent, Room)\}\}
\end{aligned}
$$

$$
\begin{aligned}
&IP_{door} = \{\\
&\quad \{open(Door)\},\\
&\quad \{closed(Door)\}\}
\end{aligned}
$$

## B.5 PRECEDE Goal Order: $P$

The set of domain predicates for this modelled planning domain are:

$$
\begin{aligned}
&\{open(D), closed(D), in(A, R), near(A, O), next\_to(O, O'), at\_door(O, D, R),\\
&\quad in\_room(O, R)\}
\end{aligned}
$$

where $D$, $A$, $R$ are variables which can be bound to objects of sort *door*, *robot* and *room* respectively and $O$, $O'$ to objects of sort *box*. The output of PRECEDE compilation for this domain model is:

$$
\begin{aligned}
&P = \{\\
&\quad (in(A, R), at(A, D, R)),\\
&\quad (in(A, R), near(A, O)),\\
&\quad (near(A, O), next\_to(O, O')),\\
&\quad (in\_room(O, R), at\_door(O, D, R))\}
\end{aligned}
$$

## B.6 $\mathcal{M}$: PROPOSE macros

For this modelled planning domain, the set $\mathcal{M}$ is made up of a set $M_s$ where $s$ can be one of the sorts listed in section B.4. The elements of each $M_s$ are pairs of the form $((a, b), c)$ where $a$ and $b$ are a task configuration, $a$ is a start situation for objects of sort $s$, $b$ is the end situation and $c$ is the set of macros for this task configuration.

For simplicity, the anonymous variable "_" is used to represent all variables in predicate arguments in the macro. When the macros are used during planning, these variable arguments must be instantiated to objects of the correct sort. For example, the first element in $M_{robot}$ below has as both the start and end situation of the macro, $robot\_in(\_)$ and $robot\_at(\_, \_)$. Where the first argument of *robot_in* is of sort *room* as is the second argument of $robot\_at$. The planning application that accepts a domain model including these compiled macros must be able to bind the variables correctly.

$$
\begin{array}{l}
M_{robot} = \{ \\
\quad ((\{at(\_,\_,\_), in(\_,\_)\}, \{at(\_,\_,\_), in(\_,\_)\}), \{ \\
\qquad [[gotodoor(\_,\_,\_), gothrudoor(\_,\_,\_)]^*], \\
\qquad [gothrudoor(\_,\_,\_)], \\
\qquad [[gothrudoor(\_,\_,\_), gotodoor(\_,\_,\_)]^*], \\
\qquad [gotodoor(\_,\_,\_)]\}), \\
\quad ((\{at(\_,\_,\_), in(\_,\_)\}, \{near(\_,\_), in(\_,\_)\}), \{ \\
\qquad [gothrudoor(\_,\_,\_), [gotodoor(\_,\_,\_), gothrudoor(\_,\_,\_)]^*, goto(\_,\_,\_)], \\
\qquad [goto(\_,\_,\_)]\}), \\
\quad ((\{at(\_,\_,\_), in(\_,\_)\}, \{in(\_,\_)\}), \{ \\
\qquad \}), \\
\quad ((\{near(\_,\_), in(\_,\_)\}, \{at(\_,\_,\_), in(\_,\_)\}), \{ \\
\qquad [[gotodoor(\_,\_,\_), gothrudoor(\_,\_,\_)]^*, gotodoor(\_,\_,\_)]\}), \\
\quad ((\{near(\_,\_), in(\_,\_)\}, \{near(\_,\_), in(\_,\_)\}), \{ \\
\qquad [[gotodoor(\_,\_,\_), gothrudoor(\_,\_,\_)]^*, goto(\_,\_,\_)]\}), \\
\quad ((\{near(\_,\_), in(\_,\_)\}, \{in(\_,\_)\}), \{ \\
\qquad \}), \\
\quad ((\{in(\_,\_)\}, \{at(\_,\_,\_), in(\_,\_)\}), \{ \\
\qquad [gotodoor(\_,\_,\_), [gothrudoor(\_,\_,\_), gotodoor(\_,\_,\_)]^*]\}), \\
\quad ((\{in(\_,\_)\}, \{near(\_,\_), in(\_,\_)\}), \{ \\
\qquad [goto(\_,\_,\_)], \\
\qquad [gotodoor(\_,\_,\_), [gothrudoor(\_,\_,\_), gotodoor(\_,\_,\_)]^*, goto(\_,\_,\_)]\}), \\
\quad ((\{in(\_,\_)\}, \{in(\_,\_)\}), \{ \\
\qquad \})\}
\end{array}
$$

$$
\begin{array}{l}
M_{box} = \{ \\
\quad ((\{in\_room(\_,\_), at\_door(\_,\_,\_)\}, \{in\_room(\_,\_), at\_door(\_,\_,\_)\}), \{ \\
\qquad [[pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*], \\
\qquad [pushthrudoor(\_,\_,\_,\_)], \\
\qquad [[pushthrudoor(\_,\_,\_,\_), pushtodoor(\_,\_,\_,\_)]^*], \\
\qquad [pushtodoor(\_,\_,\_,\_)]\}), \\
\quad ((\{at\_door(\_,\_,\_), in\_room(\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{ \\
\qquad [pushthrudoor(\_,\_,\_,\_), [pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*, \\
\qquad\quad pushtobox(\_,\_,\_)], [pushtobox(\_,\_,\_)]\}), \\
\quad ((\{at\_door(\_,\_,\_), in\_room(\_,\_)\}, \{in\_room(\_,\_)\}\}, \{ \\
\qquad \}), \\
\quad ((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_), at\_door(\_,\_,\_)\}), \{ \\
\qquad [[pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*, pushtodoor(\_,\_,\_,\_)]\}), \\
\quad ((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{ \\
\qquad [[pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*, pushtobox(\_,\_,\_)]\}), \\
\quad ((\{in\_room(\_,\_), next\_to(\_,\_)\}, \{in\_room(\_,\_)\}\}, \{ \\
\qquad \}),
\end{array}
$$

$((\{in\_room(\_,\_)\}, \{at\_door(\_,\_,\_), in\_room(\_,\_)\}), \{$
$\quad [[pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*, pushtodoor(\_,\_,\_,\_)]\}),$
$((\{in\_room(\_,\_)\}, \{in\_room(\_,\_), next\_to(\_,\_)\}), \{$
$\quad [[pushtodoor(\_,\_,\_,\_), pushthrudoor(\_,\_,\_,\_)]^*, pushtobox(\_,\_,\_)]\}),$
$((\{in\_room(\_,\_)\}, \{in\_room(\_,\_)\}), \{$
$\quad \})\}$

## B.7 $\mathcal{EM}$: PROPOSE Enumerations

There is only one static sort that is changed by iterative macros in this modelled planning domain, which is:

$\{door, room\}$

Below we list part of the set of enumerated sequences for this static sort:

$EM_{room,door} = \{$
$\quad ((room1, door12), \{[room1, door12]\}),$
$\quad ((room1, door23), \{[room1, door12, room2, door23]\}),$
$\quad ((room1, door24), \{[room1, door12, room2, door24]\}),$
$\quad ((room1, door25), \{[room1, door12, room2, door25]\}),$
$\quad ((room1, door35), \{[room1, door12, room2, door23, room3, door35],$
$\quad\quad [room1, door12, room2, door25, room5, door35]\}),$
$\quad ((room1, door45), \{[room1, door12, room2, door24, room4, door45]\})$
$\quad ((room1, door47), \{[room1, door12, room2, door24, room4, door47]\}),$
$\quad ((room1, door57), \{[room1, door12, room2, door25, room5, door57]\}),$
$\ldots\}$

## B.8 Empirical Results

In this section we list the results of the empirical tests that were the subject of chapter 7. The section is divided into three subsections which correspond to the following performance metrics that were used in the tests:

- cumulative CPU time
- cumulative number of partial plans
- number of operators

Each section is subdivided into two, the first part presents the results for a randomly generated problem sample that had been controlled for correctness by removing all tasks that the basic planner *MEA* could not solve. The second section is the results for an uncontrolled problem sample which had been randomly generated.

In the tables the *columns* give the results for each of the 9 different planning applications which were tested (for further explanation of these different applications see section 7.1). Down the left hand side of the tables the *rows* are numbered with the task number. The tasks that were used are listed on page 187.

### B.8.1 Cumulative CPU

#### B.8.1.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 6 | | 22 | | 6 | | 6 | | 7 | f | 23 | | 14 | | 7 | | 7 | f |
| 2. | 104 | | 8 | | 461 | | 12 | | 26 | | 14 | | 10 | | 40 | | 14 | |
| 3. | 5 | | 4 | | 1 | | 5 | | 5 | | 2 | | 5 | | 2 | | 6 | |
| 4. | 1 | | 2 | | 18 | | 2 | | 2 | | 3 | | 2 | | 4 | | 2 | |
| 5. | 4 | | 3 | | 2 | | 4 | | 5 | | 3 | | 4 | | 2 | | 4 | |
| 6. | 2 | | 2 | | 4 | | 2 | | 2 | | 3 | | 2 | | 4 | | 2 | |
| 7. | 2 | | 2 | | 2 | | 2 | | 1 | | 2 | | 2 | | 3 | | 2 | |
| 8. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 9. | 3 | | 3 | | 4 | | 2 | | 3 | | 3 | | 3 | | 4 | | 3 | |
| 10. | 14 | | 3 | | 262 | | 6 | | 16 | | 5 | | 4 | | 51 | | 7 | |
| 11. | 90 | | 13 | | 63 | | 15 | | 13 | | 13 | | 15 | | 36 | | 14 | |
| 12. | 79 | | 696 | a | 337 | | 10 | | 83 | f | 738 | a | 713 | a | 11 | | 11 | f |
| 13. | 4 | | 4 | | 59 | | 4 | | 2 | | 4 | | 4 | | 12 | | 3 | |
| 14. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 15. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 16. | 1 | | 4 | | 1 | | 5 | | 1 | | 6 | | 4 | | 5 | | 6 | |
| 17. | 1 | | 1 | | 2 | | 1 | | 1 | | 2 | | 2 | | 1 | | 2 | |
| 18. | 341 | | 13 | | 1123 | a | 896 | a | 342 | | 15 | | 13 | | 58 | | 372 | |
| 19. | 2 | | 2 | | 2 | | 2 | | 3 | | 2 | | 2 | | 2 | | 3 | |
| 20. | 4 | | 5 | | 6 | | 4 | | 5 | | 3 | | 6 | | 5 | | 5 | |
| 21. | 42 | | 13 | | 35 | | 14 | | 9 | f | 15 | | 10 | | 24 | | 13 | f |
| 22. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |
| 23. | 61 | | 3 | | 5 | | 3 | | 64 | | 7 | | 3 | | 4 | | 3 | |
| 24. | 94 | | 15 | | 163 | | 10 | | 100 | | 8 | | 17 | | 15 | | 12 | |
| 25. | 198 | | 13 | | 13 | | 11 | | 14 | | 27 | | 14 | | 15 | | 12 | |
| 26. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 27. | 339 | | 5 | | 1315 | | 6 | | 348 | | 37 | | 5 | | 44 | | 6 | |
| 28. | 624 | | 8 | | 238 | | 548 | | 640 | | 6 | | 10 | | 131 | | 22 | |
| 29. | 245 | | 6 | | 21 | | 5 | | 258 | | 8 | | 10 | | 10 | | 11 | |
| 30. | 9 | | 4 | | 14 | | 4 | | 10 | | 6 | | 5 | | 6 | | 4 | |
| 31. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 32. | 9 | | 12 | | 22 | | 17 | | 11 | | 34 | | 14 | | 126 | | 19 | |
| 33. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 34. | 19 | | 8 | | 1406 | a | 13 | | 15 | | 9 | | 9 | | 39 | | 15 | |
| 35. | 1 | | 2 | | 7 | | 2 | | 1 | | 3 | | 2 | | 1 | | 2 | |
| 36. | 4 | | 7 | | 4 | | 4 | | 7 | | 8 | | 12 | | 3 | | 8 | |
| 37. | 653 | | 39 | | 157 | | 27 | | 278 | | 99 | f | 36 | | 109 | | 26 | |
| 38. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |
| 39. | 106 | | 7 | | 22 | | 7 | | 112 | | 21 | | 9 | | 7 | | 7 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40. | 22 | | 3 | | 54 | | 8 | | 23 | | 25 | | 4 | | 6 | | 8 | |
| 41. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 42. | 52 | | 12 | | 28 | | 7 | | 56 | | 11 | | 13 | | 17 | | 7 | |
| 43. | 9 | | 10 | | 12 | | 10 | | 10 | | 192 | | 11 | | 15 | | 14 | |
| 44. | 31 | | 5 | | 93 | | 11 | | 12 | | 4 | | 7 | | 13 | | 12 | |
| 45. | 6 | | 9 | | 3 | | 6 | | 5 | | 13 | | 4 | | 3 | | 5 | |
| 46. | 85 | | 735 | | 76 | | 664 | a | 118 | | 64 | | 687 | | 126 | | 276 | |
| 47. | 5 | | 4 | | 1 | | 5 | | 5 | | 3 | | 5 | | 1 | | 6 | |
| 48. | 2 | | 2 | | 1 | | 1 | | 2 | | 2 | | 2 | | 1 | | 1 | |
| 49. | 1 | | 1 | | 2 | | 1 | | 1 | | 6 | | 1 | | 1 | | 1 | |
| 50. | 15 | | 4 | | 7 | | 5 | | 6 | | 6 | | 5 | | 6 | | 6 | |
| 51. | 79 | | 8 | | 128 | | 8 | | 48 | | 17 | | 9 | | 11 | | 9 | |
| 52. | 30 | | 6 | | 45 | | 7 | | 33 | | 5 | | 7 | | 9 | | 8 | |
| 53. | 3 | | 4 | | 7 | | 3 | | 3 | | 6 | | 4 | | 2 | f | 3 | |
| 54. | 7 | | 98 | | 9 | | 18 | | 9 | | 10 | | 35 | | 10 | | 20 | |
| 55. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 56. | 39 | | 11 | | 56 | | 26 | | 43 | | 15 | | 12 | | 17 | | 23 | |
| 57. | 1 | | 1 | | 20 | | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | |
| 58. | 0 | | 0 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 59. | 159 | | 1116 | a | 22 | | 6 | | 13 | | 1329 | a | 1148 | a | 12 | | 7 | |
| 60. | 6 | | 2 | | 13 | | 2 | | 7 | | 5 | | 3 | | 7 | | 3 | |
| 61. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 62. | 4 | | 4 | | 16 | | 4 | | 4 | | 8 | | 4 | | 7 | | 4 | |
| 63. | 3 | | 9 | | 2 | | 2 | | 4 | | 6 | | 10 | | 2 | | 3 | |
| 64. | 4 | | 5 | | 3 | | 4 | | 4 | | 5 | | 5 | | 5 | | 4 | |
| 65. | 31 | | 13 | | 134 | | 16 | | 19 | f | 16 | | 11 | | 76 | | 14 | |
| 66. | 3 | | 3 | | 2 | | 3 | | 3 | | 2 | | 3 | | 1 | | 4 | |
| 67. | 13 | | 8 | | 17 | | 11 | | 8 | | 5 | | 5 | | 10 | | 7 | |
| 68. | 212 | | 4 | | 20 | | 5 | | 223 | | 5 | | 4 | | 11 | | 5 | |
| 69. | 3 | | 3 | | 6 | | 3 | | 4 | | 4 | | 4 | | 112 | | 4 | |
| 70. | 222 | | 8 | | 36 | | 8 | | 234 | | 139 | | 11 | | 20 | | 11 | |
| 71. | 115 | | 3 | | 319 | | 5 | | 6 | | 4 | | 3 | | 34 | | 5 | |
| 72. | 16 | | 3 | | 11 | | 4 | | 17 | | 5 | | 3 | | 8 | | 4 | |
| 73. | 9 | | 4 | | 24 | | 9 | | 11 | | 8 | | 6 | | 15 | | 12 | |
| 74. | 6 | | 2 | | 6 | | 7 | | 7 | | 2 | | 3 | | 4 | | 7 | |
| 75. | 4 | | 7 | | 11 | | 5 | | 5 | | 19 | | 7 | | 14 | | 5 | |
| 76. | 4 | | 3 | | 5 | | 4 | | 4 | | 3 | | 2 | | 6 | | 3 | |
| 77. | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | |
| 78. | 124 | | 3 | | 11 | | 5 | | 131 | | 7 | | 4 | | 9 | | 5 | |
| 79. | 4 | | 4 | | 3 | | 5 | | 5 | | 3 | | 5 | | 3 | | 5 | |
| 80. | 466 | | 3 | | 108 | | 4 | | 478 | | 3 | | 4 | | 21 | | 4 | |
| 81. | 1 | | 1 | | 2 | | 1 | | 2 | | 1 | | 2 | | 2 | | 2 | |
| 82. | 62 | | 4 | | 22 | | 17 | | 17 | | 13 | | 4 | | 42 | | 16 | |
| 83. | 1152 | | 5 | | 25 | | 14 | | 37 | | 10 | | 6 | | 14 | | 14 | |
| 84. | 4 | | 5 | | 3 | | 4 | | 5 | | 5 | | 5 | | 2 | | 5 | |
| 85. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 86. | 3 | | 2 | | 30 | | 4 | | 3 | | 1 | f | 3 | | 10 | | 4 | |
| 87. | 5 | | 4 | | 15 | | 4 | | 5 | | 2 | f | 5 | | 8 | | 5 | |
| 88. | 4 | | 3 | | 12 | | 4 | | 8 | | 3 | | 13 | | 4 | | 8 | |
| 89. | 3 | | 18 | | 79 | | 5 | | 4 | | 2 | | 20 | | 2 | | 5 | |
| 90. | 10 | | 3 | | 10 | | 4 | | 13 | | 3 | | 3 | | 4 | | 4 | |
| 91. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 92. | 2 | | 2 | | 3 | | 3 | | 3 | | 1 | | 2 | | 2 | | 4 | |
| 93. | 6 | | 6 | | 279 | | 4 | | 6 | f | 4 | | 6 | f | 16 | | 4 | f |
| 94. | 6 | | 6 | | 8 | | 6 | | 5 | | 5 | | 5 | | 8 | | 5 | |
| 95. | 24 | | 5 | | 8 | | 9 | | 11 | f | 9 | | 5 | f | 7 | | 8 | f |
| 96. | 2 | | 2 | | 2 | | 3 | | 3 | | 2 | | 3 | | 2 | | 3 | |
| 97. | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 98. | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | | 1 | |
| 99. | 6 | | 6 | | 227 | | 5 | | 5 | | 5 | f | 5 | | 5 | | 4 | |
| 100. | 118 | | 15 | | 87 | | 12 | | 123 | | 18 | | 11 | | 25 | | 9 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 6282 | | 3150 | | 7912 | | 2653 | | 4209 | | 3188 | | 3128 | | 1563 | | 1252 | |

### B.8.1.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 2. | 1309 | a | 104 | | 922 | a | 1008 | a | 1269 | a | 125 | | 112 | | 989 | a | 1042 | a |
| 3. | 1037 | a | 19 | | 704 | a | 46 | | 1193 | a | 19 | | 20 | | 215 | | 38 | |
| 4. | 6 | | 22 | | 5 | | 6 | | 6 | f | 20 | | 14 | | 6 | | 7 | f |
| 5. | 104 | | 8 | | 452 | | 11 | | 26 | | 13 | | 10 | | 44 | | 13 | |
| 6. | 5 | | 4 | | 1 | | 5 | | 5 | | 3 | | 5 | | 2 | | 6 | |
| 7. | 1 | | 2 | | 18 | | 2 | | 2 | | 3 | | 2 | | 4 | | 2 | |
| 8. | 1223 | a | 29 | | 7 | f | 24 | | 1233 | a | 18 | | 32 | | 34 | | 26 | |
| 9. | 4 | | 3 | | 2 | | 4 | | 5 | | 3 | | 4 | | 2 | | 4 | |
| 10. | 2 | | 2 | | 4 | | 2 | | 2 | | 3 | | 2 | | 3 | | 2 | |
| 11. | 1154 | a | 1148 | a | 1116 | a | 1178 | a | 1234 | a | 1324 | a | 1226 | a | 1153 | a | 1303 | a |
| 12. | 2 | | 2 | | 2 | | 2 | | 1 | | 2 | | 2 | | 2 | | 2 | |
| 13. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 14. | 1157 | a | 6 | | 1249 | a | 6 | | 298 | | 8 | | 6 | | 43 | | 6 | |
| 15. | 1125 | a | 21 | | 985 | a | 27 | | 1174 | a | 337 | | 23 | f | 87 | | 31 | f |
| 16. | 1093 | a | 685 | | 1105 | a | 1186 | a | 1033 | a | 20 | | 66 | | 1237 | a | 1263 | a |
| 17. | 3 | | 2 | | 4 | | 2 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 18. | 1224 | a | 154 | | 1109 | a | 514 | | 93 | | 113 | | 75 | | 90 | | 83 | |
| 19. | 15 | | 3 | | 194 | | 6 | | 16 | | 4 | | 4 | | 7 | | 7 | |
| 20. | 1041 | a | 62 | | 998 | a | 70 | | 997 | a | 53 | | 12 | | 38 | | 14 | |
| 21. | 688 | a | 823 | a | 708 | a | 879 | a | 682 | a | 30 | | 847 | a | 222 | | 907 | a |
| 22. | 94 | | 13 | | 44 | | 15 | | 13 | | 14 | | 15 | | 16 | | 14 | |
| 23. | 1031 | a | 5 | | 1084 | a | 8 | | 443 | | 7 | | 6 | | 42 | | 10 | |
| 24. | 82 | | 694 | a | 330 | | 10 | | 83 | f | 374 | | 742 | a | 10 | | 11 | f |
| 25. | 4 | | 4 | | 45 | | 4 | | 2 | | 7 | | 4 | | 33 | | 3 | |
| 26. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 27. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 28. | 1 | | 4 | | 2 | | 5 | | 1 | | 3 | | 4 | | 6 | | 6 | |
| 29. | 1088 | a | 9 | | 1026 | a | 9 | | 1063 | a | 214 | | 10 | | 1276 | a | 10 | |
| 30. | 1254 | a | 9 | | 1251 | | 60 | | 229 | | 29 | | 11 | | 64 | | 27 | |
| 31. | 1014 | a | 1068 | a | 995 | a | 1169 | a | 992 | a | 1363 | a | 1099 | a | 1344 | a | 1181 | a |
| 32. | 1021 | a | 12 | | 912 | a | 44 | | 1084 | a | 30 | | 13 | | 93 | | 46 | |
| 33. | 1 | | 2 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 2 | |
| 34. | 356 | | 14 | | 1115 | a | 933 | a | 344 | | 15 | | 14 | | 101 | | 379 | |
| 35. | 2 | | 2 | | 2 | | 2 | | 3 | | 2 | | 2 | | 2 | | 4 | |
| 36. | 4 | | 5 | | 5 | | 4 | | 5 | | 4 | | 6 | | 3 | | 5 | |
| 37. | 44 | | 14 | | 67 | | 15 | | 9 | f | 13 | | 10 | | 33 | | 13 | f |
| 38. | 954 | a | 33 | | 737 | a | 976 | a | 736 | a | 28 | | 36 | | 176 | | 1019 | a |
| 39. | 749 | a | 37 | | 1011 | a | 618 | a | 520 | a | 61 | | 43 | | 728 | a | 1011 | a |
| 40. | 1 | | 2 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |
| 41. | 1092 | a | 23 | | 1127 | a | 28 | | 1107 | a | 87 | | 20 | | 1349 | a | 23 | |
| 42. | 63 | | 3 | | 1104 | a | 3 | | 65 | | 13 | | 3 | | 11 | | 3 | |
| 43. | 1106 | a | 15 | | 542 | | 155 | | 114 | | 12 | | 6 | | 188 | | 68 | |
| 44. | 97 | | 15 | | 8 | | 11 | | 103 | | 7 | | 18 | | 10 | | 12 | |
| 45. | 1133 | a | 25 | | 1206 | a | 44 | | 1150 | a | 77 | | 26 | | 158 | | 47 | |
| 46. | 207 | | 13 | | 15 | | 11 | | 14 | | 22 | | 14 | | 16 | | 12 | |
| 47. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 48. | 1069 | a | 811 | a | 1038 | a | 911 | a | 1049 | a | 24 | | 858 | a | 674 | | 949 | a |
| 49. | 1027 | a | 949 | a | 1087 | a | 900 | | 1011 | a | 154 | | 1036 | a | 357 | | 922 | |
| 50. | 1026 | a | 622 | a | 952 | a | 1001 | a | 998 | a | 1108 | a | 656 | a | 1072 | a | 1033 | a |
| 51. | 2 | f | 2 | f | 3 | f | 2 | f | 2 | f | 2 | f | 2 | f | 2 | f | 2 | f |
| 52. | 356 | | 5 | | 165 | | 6 | | 351 | | 10 | | 6 | | 43 | | 7 | |
| 53. | 1084 | a | 6 | | 25 | | 7 | | 1102 | a | 13 | | 7 | | 15 | | 8 | |
| 54. | 1051 | a | 70 | | 658 | | 1107 | a | 1051 | a | 74 | | 66 | | 349 | | 1091 | a |
| 55. | 658 | | 9 | | 220 | | 571 | | 669 | | 13 | | 10 | | 212 | | 24 | |
| 56. | 1018 | a | 30 | | 812 | a | 246 | | 1036 | a | 43 | | 23 | | 420 | | 146 | |
| 57. | 1055 | a | 23 | | 1231 | a | 1118 | | 1102 | a | 48 | | 17 | | 1596 | | 52 | |
| 58. | 1024 | a | 21 | | 1080 | a | 35 | | 1044 | a | 19 | | 25 | f | 839 | a | 14 | f |
| 59. | 1005 | a | 31 | | 1182 | a | 1104 | | 1038 | a | 43 | | 21 | | 102 | | 23 | |
| 60. | 1247 | a | 20 | | 1345 | a | 105 | | 1247 | a | 27 | f | 16 | | 258 | | 80 | |
| 61. | 255 | | 6 | | 1183 | a | 6 | | 268 | | 9 | | 11 | | 13 | | 11 | |
| 62. | 9 | | 5 | | 46 | | 4 | | 10 | | 7 | | 5 | | 6 | | 5 | |
| 63. | 0 | | 1 | | 0 | | 0 | | 0 | | 1 | | 1 | | 0 | | 0 | |
| 64. | 9 | | 12 | | 11 | | 18 | | 12 | | 23 | | 14 | | 26 | | 20 | |
| 65. | 1057 | a | 12 | | 1033 | a | 1056 | a | 1051 | a | 94 | | 14 | | 1267 | a | 1091 | a |
| 66. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 67. | 1048 | a | 12 | | 68 | | 11 | | 1063 | a | 16 | | 14 | | 56 | | 12 | |
| 68. | 21 | | 8 | | 1319 | a | 14 | | 16 | | 14 | | 10 | | 71 | | 15 | |
| 69. | 1 | | 2 | | 3 | | 2 | | 1 | | 2 | | 3 | | 1 | | 2 | |
| 70. | 4 | | 8 | | 4 | | 5 | | 8 | | 7 | | 13 | | 4 | | 8 | |
| 71. | 672 | | 38 | | 354 | | 29 | | 298 | | 34 | | 38 | | 118 | | 27 | |
| 72. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |
| 73. | 1003 | a | 8 | | 740 | | 20 | | 1127 | a | 17 | | 10 | | 22 | | 21 | |
| 74. | 1216 | a | 15 | | 897 | a | 17 | | 1173 | a | 17 | f | 19 | | 17 | | 19 | f |
| 75. | 1220 | a | 24 | | 1849 | | 1228 | a | 1280 | a | 19 | | 28 | | 105 | | 1284 | a |
| 76. | 110 | | 7 | | 6 | | 7 | | 115 | | 6 | | 9 | | 6 | | 8 | |
| 77. | 22 | | 3 | | 123 | | 8 | | 24 | | 9 | | 4 | | 5 | | 8 | |
| 78. | 969 | a | 13 | | 1300 | | 12 | | 966 | a | 29 | | 27 | | 46 | | 25 | |
| 79. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80. | 1030 | a | 8 | | 37 | | 18 | | 1005 | a | 9 | | 9 | | 104 | | 20 | |
| 81. | 1052 | a | 1116 | a | 305 | | 1464 | a | 1012 | a | 31 | | 29 | | 458 | | 162 | f |
| 82. | 55 | | 12 | | 10 | | 8 | | 56 | | 5 | | 14 | | 13 | | 8 | |
| 83. | 9 | | 11 | | 47 | | 11 | | 10 | | 10 | | 12 | | 17 | | 15 | |
| 84. | 33 | | 5 | | 5 | | 12 | | 12 | | 10 | | 7 | | 18 | | 13 | |
| 85. | 750 | a | 7 | | 1157 | a | 35 | | 744 | a | 15 | | 8 | | 28 | | 36 | |
| 86. | 1102 | a | 965 | a | 3 | f | 1263 | a | 1127 | a | 361 | | 1345 | a | 1865 | a | 1132 | a |
| 87. | 1101 | a | 9 | | 1096 | | 1099 | a | 1134 | a | 27 | | 54 | | 328 | | 1117 | a |
| 88. | 6 | | 738 | | 3 | | 6 | | 5 | | 4 | | 5 | | 9 | | 6 | |
| 89. | 88 | | 10 | | 81 | | 691 | a | 119 | | 217 | | 708 | | 917 | f | 293 | |
| 90. | 1102 | a | 5 | | 1122 | a | 1351 | a | 1060 | a | 17 | | 11 | | 1823 | a | 58 | |
| 91. | 5 | | 2 | | 1 | | 5 | | 5 | | 2 | | 5 | | 5 | | 6 | |
| 92. | 2 | | 1 | | 2 | | 1 | | 2 | | 1 | | 2 | | 2 | | 1 | |
| 93. | 1 | | 5 | | 2 | | 1 | | 1 | | 2 | | 2 | | 2 | | 1 | |
| 94. | 15 | | 4 | | 5 | f | 5 | | 7 | | 7 | | 5 | | 8 | | 6 | |
| 95. | 1043 | a | 117 | | 67 | | 5 | | 1014 | a | 6 | | 5 | | 9 | | 6 | |
| 96. | 877 | a | 8 | | 252 | | 114 | | 942 | a | 42 | | 113 | f | 205 | | 64 | f |
| 97. | 84 | | 6 | | 322 | | 8 | | 51 | | 14 | | 10 | | 22 | | 10 | |
| 98. | 32 | | 14 | | 40 | | 7 | | 35 | | 10 | | 7 | | 13 | | 8 | |
| 99. | 1064 | a | 76 | | 18 | | 991 | a | 1119 | a | 12 | | 16 | | 36 | | 1004 | a |
| 100. | 1011 | a | 29 | | 962 | a | 1080 | a | 15 | | 83 | | 159 | | 997 | a | | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 52295 | | 11016 | | 45557 | | 26715 | | 47010 | | 7188 | | 9989 | | 23589 | | 20540 | |

### B.8.2 Partial Plans

#### B.8.2.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 97 | | 266 | | 84 | | 86 | | 99 | f | 262 | | 167 | | 86 | | 88 | f |
| 2. | 569 | | 114 | | 1351 | | 133 | | 243 | | 127 | | 117 | | 305 | | 125 | |
| 3. | 77 | | 71 | | 21 | | 71 | | 77 | | 26 | | 71 | | 21 | | 71 | |
| 4. | 24 | | 21 | | 141 | | 23 | | 24 | | 20 | | 21 | | 33 | | 23 | |
| 5. | 69 | | 57 | | 19 | | 57 | | 69 | | 33 | | 57 | | 19 | | 57 | |
| 6. | 32 | | 28 | | 51 | | 25 | | 32 | | 24 | | 28 | | 36 | | 25 | |
| 7. | 36 | | 28 | | 32 | | 28 | | 16 | | 19 | | 16 | | 32 | | 16 | |
| 8. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 9. | 47 | | 37 | | 47 | | 35 | | 39 | | 36 | | 34 | | 34 | | 28 | |
| 10. | 154 | | 46 | | 776 | | 77 | | 154 | | 44 | | 46 | | 267 | | 77 | |
| 11. | 557 | | 164 | | 315 | | 172 | | 139 | | 103 | | 155 | | 198 | | 127 | |
| 12. | 504 | | 2001 | a | 928 | | 135 | | 490 | f | 2008 | a | 2001 | a | 96 | | 125 | f |
| 13. | 71 | | 59 | | 317 | | 54 | | 34 | | 39 | | 47 | | 90 | | 33 | |
| 14. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 15. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 16. | 20 | | 63 | | 15 | | 71 | | 20 | | 62 | | 63 | | 46 | | 71 | |
| 17. | 23 | | 26 | | 17 | | 22 | | 23 | | 14 | | 26 | | 10 | | 22 | |
| 18. | 1359 | | 164 | | 2001 | a | 2000 | a | 1288 | | 134 | | 145 | | 468 | | 1037 | |
| 19. | 45 | | 24 | | 25 | | 38 | | 47 | | 27 | | 20 | | 22 | | 45 | |
| 20. | 68 | | 72 | | 57 | | 62 | | 68 | | 28 | | 72 | | 44 | | 62 | |
| 21. | 392 | | 177 | | 239 | | 186 | | 111 | f | 134 | | 116 | | 186 | | 142 | f |
| 22. | 19 | | 22 | | 4 | | 10 | | 19 | | 12 | | 22 | | 4 | | 10 | |
| 23. | 372 | | 32 | | 51 | | 38 | | 372 | | 63 | | 32 | | 44 | | 38 | |
| 24. | 593 | | 220 | | 678 | | 145 | | 593 | | 65 | | 220 | | 135 | | 145 | |
| 25. | 693 | | 159 | | 80 | | 127 | | 145 | | 164 | | 150 | | 99 | | 117 | |
| 26. | 9 | | 9 | | 4 | | 9 | | 9 | | 8 | | 9 | | 6 | | 9 | |
| 27. | 953 | | 66 | | 1830 | | 75 | | 953 | | 285 | | 66 | | 257 | | 75 | |
| 28. | 1357 | | 116 | | 717 | | 1440 | | 1357 | | 65 | | 116 | | 461 | | 214 | |
| 29. | 816 | | 87 | | 81 | | 77 | | 846 | | 80 | | 112 | | 81 | | 110 | |
| 30. | 111 | | 63 | | 92 | | 52 | | 107 | | 48 | | 59 | | 36 | | 48 | |
| 31. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 32. | 124 | | 135 | | 97 | | 166 | | 120 | | 228 | | 130 | | 449 | | 161 | |
| 33. | 8 | | 9 | | 5 | | 9 | | 8 | | 6 | | 9 | | 5 | | 9 | |
| 34. | 210 | | 113 | | 2000 | a | 164 | | 155 | | 79 | | 103 | | 220 | | 148 | |
| 35. | 17 | | 29 | | 26 | | 23 | | 17 | | 19 | | 29 | | 9 | | 23 | |
| 36. | 69 | | 110 | | 54 | | 60 | | 100 | | 70 | | 157 | | 28 | | 90 | |
| 37. | 1639 | | 436 | | 741 | | 285 | | 1111 | | 525 | f | 372 | | 569 | | 230 | |
| 38. | 15 | | 15 | | 9 | | 12 | | 15 | | 5 | | 15 | | 5 | | 12 | |
| 39. | 532 | | 106 | | 167 | | 94 | | 529 | | 186 | | 106 | | 50 | | 85 | |
| 40. | 258 | | 46 | | 334 | | 112 | | 237 | | 172 | | 46 | | 47 | | 93 | |
| 41. | 5 | | 5 | | 2 | | 5 | | 5 | | 2 | | 5 | | 2 | | 5 | |
| 42. | 371 | | 175 | | 194 | | 96 | | 371 | | 76 | | 175 | | 110 | | 83 | |
| 43. | 149 | | 158 | | 104 | | 141 | | 131 | | 576 | | 138 | | 129 | | 148 | |
| 44. | 289 | | 77 | | 406 | | 155 | | 132 | | 49 | | 81 | | 108 | | 142 | |
| 45. | 104 | | 128 | | 36 | | 90 | | 72 | | 124 | | 56 | | 32 | | 63 | |
| 46. | 696 | | 1832 | | 451 | | 2008 | a | 731 | | 364 | | 1734 | | 496 | | 1228 | |
| 47. | 76 | | 71 | | 14 | | 71 | | 76 | | 34 | | 71 | | 14 | | 71 | |
| 48. | 28 | | 33 | | 16 | | 15 | | 28 | | 16 | | 33 | | 9 | | 15 | |
| 49. | 17 | | 13 | | 8 | | 13 | | 17 | | 60 | | 13 | | 9 | | 13 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50. | 169 | | 70 | | 50 | | 73 | | 82 | | 52 | | 63 | | 71 | | 68 | |
| 51. | 564 | | 109 | | 489 | | 105 | | 392 | | 106 | | 105 | | 88 | | 101 | |
| 52. | 313 | | 87 | | 265 | | 93 | | 313 | | 46 | | 85 | | 87 | | 91 | |
| 53. | 45 | | 51 | | 72 | | 45 | | 45 | | 36 | | 43 | | 16 | f | 37 | |
| 54. | 111 | | 435 | | 98 | | 200 | | 111 | | 70 | | 225 | | 86 | | 200 | |
| 55. | 15 | | 11 | | 7 | | 11 | | 15 | | 9 | | 11 | | 5 | | 11 | |
| 56. | 316 | | 146 | | 306 | | 283 | | 314 | | 108 | | 144 | | 126 | | 243 | |
| 57. | 15 | | 12 | | 161 | | 12 | | 15 | | 10 | | 12 | | 10 | | 12 | |
| 58. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 59. | 662 | | 2000 | a | 136 | | 93 | | 168 | | 2000 | a | 2000 | a | 70 | | 87 | |
| 60. | 84 | | 33 | | 87 | | 32 | | 84 | | 33 | | 33 | | 28 | | 32 | |
| 61. | 7 | | 7 | | 3 | | 7 | | 7 | | 3 | | 7 | | 3 | | 7 | |
| 62. | 69 | | 54 | | 80 | | 56 | | 59 | | 48 | | 44 | | 25 | | 46 | |
| 63. | 50 | | 108 | | 22 | | 32 | | 50 | | 64 | | 108 | | 19 | | 32 | |
| 64. | 60 | | 85 | | 25 | | 54 | | 48 | | 41 | | 64 | | 37 | | 47 | |
| 65. | 317 | | 165 | | 373 | | 176 | | 195 | f | 114 | | 115 | | 279 | | 128 | |
| 66. | 51 | | 46 | | 12 | | 46 | | 51 | | 20 | | 46 | | 12 | | 46 | |
| 67. | 185 | | 111 | | 69 | | 140 | | 104 | | 33 | | 49 | | 74 | | 71 | |
| 68. | 753 | | 58 | | 105 | | 65 | | 753 | | 32 | | 58 | | 75 | | 65 | |
| 69. | 46 | | 33 | | 31 | | 39 | | 46 | | 24 | | 35 | | 346 | | 37 | |
| 70. | 784 | | 122 | | 153 | | 112 | | 793 | | 748 | | 131 | | 107 | | 115 | |
| 71. | 536 | | 39 | | 701 | | 70 | | 86 | | 35 | | 39 | | 129 | | 62 | |
| 72. | 176 | | 41 | | 78 | | 50 | | 176 | | 51 | | 41 | | 40 | | 50 | |
| 73. | 116 | | 51 | | 122 | | 105 | | 117 | | 68 | | 54 | | 70 | | 108 | |
| 74. | 91 | | 34 | | 51 | | 84 | | 91 | | 16 | | 34 | | 35 | | 84 | |
| 75. | 72 | | 107 | | 46 | | 72 | | 64 | | 167 | | 94 | | 65 | | 59 | |
| 76. | 66 | | 45 | | 32 | | 51 | | 47 | | 35 | | 22 | | 41 | | 41 | |
| 77. | 10 | | 12 | | 8 | | 12 | | 10 | | 11 | | 12 | | 7 | | 12 | |
| 78. | 575 | | 46 | | 60 | | 66 | | 575 | | 39 | | 46 | | 51 | | 66 | |
| 79. | 68 | | 67 | | 23 | | 67 | | 68 | | 39 | | 67 | | 24 | | 67 | |
| 80. | 1142 | | 45 | | 354 | | 55 | | 1142 | | 33 | | 45 | | 96 | | 55 | |
| 81. | 23 | | 19 | | 13 | | 19 | | 23 | | 5 | | 19 | | 15 | | 19 | |
| 82. | 407 | | 51 | | 130 | | 167 | | 151 | | 74 | | 47 | | 176 | | 135 | |
| 83. | 1875 | | 69 | | 135 | | 170 | | 271 | | 62 | | 58 | | 88 | | 142 | |
| 84. | 67 | | 61 | | 8 | | 61 | | 67 | | 24 | | 61 | | 7 | | 61 | |
| 85. | 27 | | 27 | | 10 | | 27 | | 27 | | 21 | | 27 | | 16 | | 27 | |
| 86. | 46 | | 30 | | 178 | | 47 | | 46 | | 0 | f | 30 | | 90 | | 47 | |
| 87. | 76 | | 70 | | 75 | | 60 | | 70 | | 8 | f | 66 | | 56 | | 54 | |
| 88. | 59 | | 40 | | 73 | | 50 | | 97 | | 29 | | 141 | | 29 | | 84 | |
| 89. | 59 | | 219 | | 235 | | 65 | | 59 | | 14 | | 219 | | 7 | | 65 | |
| 90. | 125 | | 42 | | 53 | | 54 | | 133 | | 30 | | 38 | | 18 | | 50 | |
| 91. | 12 | | 12 | | 10 | | 12 | | 10 | | 10 | | 10 | | 12 | | 10 | |
| 92. | 39 | | 29 | | 25 | | 44 | | 39 | | 8 | | 11 | | 16 | | 44 | |
| 93. | 93 | | 96 | | 706 | | 56 | | 77 | f | 27 | | 76 | f | 106 | | 43 | f |
| 94. | 91 | | 94 | | 57 | | 90 | | 58 | | 34 | | 61 | | 62 | | 54 | |
| 95. | 259 | | 87 | | 43 | | 120 | | 125 | f | 64 | | 60 | f | 51 | | 93 | f |
| 96. | 39 | | 36 | | 12 | | 36 | | 39 | | 12 | | 36 | | 16 | | 36 | |
| 97. | 15 | | 12 | | 8 | | 12 | | 15 | | 5 | | 12 | | 6 | | 12 | |
| 98. | 17 | | 14 | | 7 | | 14 | | 17 | | 12 | | 14 | | 7 | | 14 | |
| 99. | 99 | | 86 | | 367 | | 69 | | 57 | | 46 | f | 52 | | 45 | | 40 | |
| 100. | 675 | | 198 | | 341 | | 151 | | 641 | | 148 | | 132 | | 153 | | 104 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 25249 | | 13209 | | 21146 | | 12401 | | 19276 | | 11249 | | 12267 | | 8509 | | 8777 | |

### B.8.2.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 2. | 2012 | a | 615 | | 2001 | a | 2011 | a | 2001 | a | 611 | | 613 | | 2003 | a | 2003 | a |
| 3. | 2002 | a | 214 | | 2010 | a | 420 | | 2004 | a | 181 | | 202 | | 955 | | 352 | |
| 4. | 97 | | 266 | | 84 | | 86 | | 99 | f | 206 | | 167 | | 77 | | 88 | f |
| 5. | 569 | | 114 | | 1351 | | 133 | | 243 | | 126 | | 117 | | 310 | | 125 | |
| 6. | 77 | | 71 | | 21 | | 71 | | 77 | | 27 | | 71 | | 21 | | 71 | |
| 7. | 24 | | 21 | | 141 | | 23 | | 24 | | 20 | | 21 | | 33 | | 23 | |
| 8. | 2004 | a | 305 | | 78 | f | 239 | | 2001 | a | 157 | | 298 | | 214 | | 225 | |
| 9. | 69 | | 57 | | 19 | | 57 | | 69 | | 33 | | 57 | | 19 | | 57 | |
| 10. | 32 | | 28 | | 51 | | 25 | | 32 | | 24 | | 28 | | 29 | | 25 | |
| 11. | 2003 | a | 2005 | a | 2001 | a | 2001 | a | 2001 | a | 2000 | a | 2000 | a | 2004 | a | 2000 | a |
| 12. | 36 | | 28 | | 32 | | 28 | | 16 | | 14 | | 16 | | 19 | | 16 | |
| 13. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 14. | 2000 | a | 87 | | 2000 | a | 77 | | 791 | | 68 | | 81 | | 201 | | 65 | |
| 15. | 2003 | a | 252 | | 2001 | a | 288 | | 2008 | a | 903 | | 244 | f | 421 | | 276 | f |
| 16. | 2006 | a | 1140 | | 2012 | a | 2006 | a | 2000 | a | 124 | | 373 | | 2005 | a | 2007 | a |
| 17. | 47 | | 37 | | 47 | | 35 | | 39 | | 25 | | 34 | | 24 | | 28 | |
| 18. | 2002 | a | 816 | | 2003 | a | 1480 | | 527 | | 606 | | 538 | | 424 | | 475 | |
| 19. | 154 | | 46 | | 680 | | 77 | | 154 | | 25 | | 46 | | 50 | | 77 | |

|  | $MEA$ |  | $MEA_1$ |  | $MEA_2$ |  | $MEA_3$ |  | $MEA_4$ |  | $MEA_5$ |  | $MEA_6$ |  | $MEA_7$ |  | $MEA_8$ |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20. | 2008 | a | 402 |  | 2003 | a | 404 |  | 2008 | a | 304 |  | 130 |  | 246 |  | 132 |  |
| 21. | 2000 | a | 2001 | a | 2002 | a | 2000 | a | 2000 | a | 191 |  | 2001 | a | 883 |  | 2000 | a |
| 22. | 557 |  | 164 |  | 301 |  | 172 |  | 139 |  | 61 |  | 155 |  | 114 |  | 127 |  |
| 23. | 2000 | a | 60 |  | 2007 | a | 91 |  | 1038 |  | 33 |  | 56 |  | 224 |  | 87 |  |
| 24. | 504 |  | 2001 | a | 919 |  | 135 |  | 490 | f | 1016 |  | 2001 | a | 88 |  | 125 | f |
| 25. | 71 |  | 59 |  | 287 |  | 54 |  | 34 |  | 50 |  | 47 |  | 85 |  | 33 |  |
| 26. | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  |
| 27. | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  |
| 28. | 20 |  | 63 |  | 17 |  | 71 |  | 20 |  | 33 |  | 63 |  | 45 |  | 71 |  |
| 29. | 2002 | a | 112 |  | 2000 | a | 110 |  | 2007 | a | 579 |  | 108 |  | 2000 | a | 106 |  |
| 30. | 2000 | a | 135 |  | 1674 |  | 389 |  | 783 |  | 222 |  | 131 |  | 242 |  | 216 |  |
| 31. | 2002 | a | 2004 | a | 2000 | a | 2008 | a | 2006 | a | 2004 | a | 2002 | a | 2001 | a | 2008 | a |
| 32. | 2000 | a | 145 |  | 2002 | a | 354 |  | 2002 | a | 119 |  | 141 |  | 514 |  | 332 |  |
| 33. | 23 |  | 26 |  | 17 |  | 22 |  | 23 |  | 6 |  | 26 |  | 10 |  | 22 |  |
| 34. | 1359 |  | 164 |  | 2001 | a | 2000 | a | 1288 |  | 123 |  | 145 |  | 575 |  | 1037 |  |
| 35. | 45 |  | 24 |  | 27 |  | 38 |  | 47 |  | 20 |  | 20 |  | 24 |  | 45 |  |
| 36. | 68 |  | 72 |  | 55 |  | 62 |  | 68 |  | 23 |  | 72 |  | 24 |  | 62 |  |
| 37. | 392 |  | 177 |  | 441 |  | 186 |  | 111 | f | 99 |  | 116 |  | 208 |  | 142 | f |
| 38. | 2006 | a | 244 |  | 2006 | a | 2001 | a | 2001 | a | 190 |  | 241 |  | 581 |  | 2003 | a |
| 39. | 2000 | a | 358 |  | 2002 | a | 2005 | a | 2002 | a | 299 |  | 354 |  | 2002 | a | 2004 | a |
| 40. | 19 |  | 22 |  | 4 |  | 10 |  | 19 |  | 4 |  | 22 |  | 4 |  | 10 |  |
| 41. | 2007 | a | 209 |  | 2006 | a | 236 |  | 2004 | a | 260 |  | 171 |  | 2004 | a | 189 |  |
| 42. | 372 |  | 32 |  | 2000 | a | 38 |  | 372 |  | 49 |  | 32 |  | 75 |  | 38 |  |
| 43. | 2002 | a | 162 |  | 1217 |  | 753 |  | 559 |  | 65 |  | 68 |  | 698 |  | 464 |  |
| 44. | 593 |  | 220 |  | 74 |  | 145 |  | 593 |  | 41 |  | 220 |  | 81 |  | 145 |  |
| 45. | 2001 | a | 284 |  | 2000 | a | 389 |  | 2000 | a | 196 |  | 247 |  | 619 |  | 356 |  |
| 46. | 693 |  | 159 |  | 100 |  | 127 |  | 145 |  | 74 |  | 150 |  | 73 |  | 117 |  |
| 47. | 9 |  | 9 |  | 4 |  | 9 |  | 9 |  | 4 |  | 9 |  | 4 |  | 9 |  |
| 48. | 2005 | a | 2000 | a | 2002 | a | 2006 | a | 2005 | a | 76 |  | 2000 | a | 1316 |  | 2006 | a |
| 49. | 2003 | a | 2003 | a | 2000 | a | 1781 |  | 2003 | a | 437 |  | 2003 | a | 904 |  | 1781 |  |
| 50. | 2001 | a | 2001 | a | 2001 | a | 2004 | a | 2001 | a | 2008 | a | 2002 | a | 2002 | a | 2004 | a |
| 51. | 33 | f | 33 | f | 11 | f | 24 | f | 33 | f | 11 | f | 33 | f | 11 | f | 24 | f |
| 52. | 953 |  | 66 |  | 560 |  | 75 |  | 953 |  | 43 |  | 66 |  | 213 |  | 75 |  |
| 53. | 2002 | a | 90 |  | 153 |  | 88 |  | 2000 | a | 59 |  | 88 |  | 86 |  | 80 |  |
| 54. | 2004 | a | 554 |  | 1214 |  | 2006 | a | 2011 | a | 343 |  | 478 |  | 801 |  | 2009 | a |
| 55. | 1357 |  | 116 |  | 701 |  | 1440 |  | 1357 |  | 61 |  | 116 |  | 648 |  | 214 |  |
| 56. | 2001 | a | 329 |  | 2004 | a | 883 |  | 2001 | a | 188 |  | 234 |  | 930 |  | 626 |  |
| 57. | 2006 | a | 236 |  | 2001 | a | 1586 |  | 2006 | a | 160 |  | 168 |  | 1703 |  | 334 |  |
| 58. | 2009 | a | 245 |  | 2001 | a | 304 |  | 2009 | a | 117 |  | 264 | f | 2000 | a | 139 | f |
| 59. | 2006 | a | 291 |  | 2000 | a | 1779 |  | 2006 | a | 205 |  | 201 |  | 366 |  | 205 |  |
| 60. | 2002 | a | 239 |  | 2000 | a | 604 |  | 2003 | a | 174 | f | 154 |  | 754 |  | 415 |  |
| 61. | 816 |  | 87 |  | 2002 | a | 77 |  | 846 |  | 62 |  | 112 |  | 89 |  | 110 |  |
| 62. | 111 |  | 63 |  | 238 |  | 52 |  | 107 |  | 39 |  | 59 |  | 43 |  | 48 |  |
| 63. | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  | 0 |  |
| 64. | 124 |  | 135 |  | 88 |  | 166 |  | 120 |  | 120 |  | 130 |  | 143 |  | 161 |  |
| 65. | 2008 | a | 123 |  | 2004 | a | 2008 | a | 2008 | a | 329 |  | 123 |  | 2003 | a | 2008 | a |
| 66. | 8 |  | 9 |  | 5 |  | 9 |  | 8 |  | 5 |  | 9 |  | 5 |  | 9 |  |
| 67. | 2000 | a | 171 |  | 388 |  | 128 |  | 2000 | a | 78 |  | 167 |  | 295 |  | 126 |  |
| 68. | 210 |  | 113 |  | 2004 | a | 164 |  | 155 |  | 89 |  | 103 |  | 328 |  | 148 |  |
| 69. | 17 |  | 29 |  | 18 |  | 23 |  | 17 |  | 12 |  | 29 |  | 6 |  | 23 |  |
| 70. | 69 |  | 110 |  | 38 |  | 60 |  | 100 |  | 39 |  | 157 |  | 28 |  | 90 |  |
| 71. | 1639 |  | 436 |  | 1346 |  | 285 |  | 1111 |  | 191 |  | 372 |  | 555 |  | 230 |  |
| 72. | 15 |  | 15 |  | 9 |  | 12 |  | 15 |  | 5 |  | 15 |  | 5 |  | 12 |  |
| 73. | 2009 | a | 107 |  | 1422 |  | 190 |  | 2009 | a | 109 |  | 107 |  | 163 |  | 190 |  |
| 74. | 2003 | a | 188 |  | 2000 | a | 189 |  | 2002 | a | 92 | f | 181 |  | 75 |  | 176 | f |
| 75. | 2001 | a | 232 |  | 1844 |  | 2004 | a | 2003 | a | 101 |  | 214 |  | 300 |  | 2001 | a |
| 76. | 532 |  | 106 |  | 62 |  | 94 |  | 529 |  | 40 |  | 106 |  | 39 |  | 85 |  |
| 77. | 258 |  | 46 |  | 602 |  | 112 |  | 237 |  | 38 |  | 46 |  | 29 |  | 93 |  |
| 78. | 2002 | a | 169 |  | 1845 |  | 132 |  | 2002 | a | 141 |  | 232 |  | 165 |  | 198 |  |
| 79. | 5 |  | 5 |  | 2 |  | 5 |  | 5 |  | 2 |  | 5 |  | 5 |  | 5 |  |
| 80. | 2006 | a | 109 |  | 205 |  | 160 |  | 2006 | a | 52 |  | 109 |  | 322 |  | 160 |  |
| 81. | 2006 | a | 2001 | a | 945 |  | 2000 | a | 2006 | a | 148 |  | 230 |  | 766 |  | 680 | f |
| 82. | 371 |  | 175 |  | 102 |  | 96 |  | 371 |  | 34 |  | 175 |  | 55 |  | 83 |  |
| 83. | 149 |  | 158 |  | 290 |  | 141 |  | 131 |  | 72 |  | 138 |  | 75 |  | 148 |  |
| 84. | 289 |  | 77 |  | 45 |  | 155 |  | 132 |  | 79 |  | 81 |  | 102 |  | 142 |  |
| 85. | 2007 | a | 94 |  | 2001 | a | 378 |  | 2007 | a | 93 |  | 88 |  | 115 |  | 366 |  |
| 86. | 2000 | a | 2001 | a | 25 | f | 2000 | a | 2010 | a | 947 |  | 2003 | a | 2003 | a | 2000 | a |
| 87. | 2001 | a | 128 |  | 1981 |  | 2001 | a | 2002 | a | 166 |  | 352 |  | 887 |  | 2004 | a |
| 88. | 104 |  | 1832 |  | 32 |  | 90 |  | 72 |  | 36 |  | 56 |  | 52 |  | 63 |  |
| 89. | 696 |  | 142 |  | 472 |  | 2008 | a | 731 |  | 815 |  | 1734 |  | 1428 | f | 1228 |  |
| 90. | 2004 | a | 71 |  | 2007 | a | 2002 | a | 2002 | a | 95 |  | 127 |  | 2002 | a | 354 |  |
| 91. | 76 |  | 33 |  | 12 |  | 71 |  | 76 |  | 16 |  | 71 |  | 38 |  | 71 |  |
| 92. | 28 |  | 13 |  | 22 |  | 15 |  | 28 |  | 6 |  | 33 |  | 18 |  | 15 |  |
| 93. | 17 |  | 70 |  | 10 |  | 13 |  | 17 |  | 10 |  | 13 |  | 9 |  | 13 |  |
| 94. | 169 |  | 52 |  | 54 | f | 73 |  | 82 |  | 52 |  | 63 |  | 63 |  | 68 |  |
| 95. | 2005 | a | 979 |  | 315 |  | 58 |  | 2005 | a | 36 |  | 52 |  | 61 |  | 58 |  |
| 96. | 2002 | a | 109 |  | 895 |  | 702 |  | 2002 | a | 247 |  | 916 | f | 659 |  | 455 | f |
| 97. | 564 |  | 87 |  | 1011 |  | 105 |  | 392 |  | 72 |  | 105 |  | 106 |  | 101 |  |
| 98. | 313 |  | 143 |  | 257 |  | 93 |  | 313 |  | 86 |  | 85 |  | 72 |  | 91 |  |
| 99. | 2002 | a | 529 |  | 131 |  | 2002 | a | 2002 | a | 72 |  | 143 |  | 218 |  | 2002 | a |
| 100. | 2003 | a | 199 |  | 2005 | a | 2003 | a | 88 |  | 523 |  | 568 |  | 2005 | a |  |  |
|  | **Total** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 106911 |  | 34560 |  | 91274 |  | 59324 |  | 97906 |  | 19811 |  | 30705 |  | 4786 |  | 47525 |  |

### B.8.3 Operators

#### B.8.3.1 Controlled Problem Sample

The table below shows the total number of operators for each of the planning combinations. These can be used to assess differences in the "quality" of the plans that the different combinations generate.

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 13 | | 15 | | 13 | | 13 | | 0 | f | 15 | | 15 | | 13 | | 0 | f |
| 2. | 37 | | 23 | | 37 | | 28 | | 45 | | 23 | | 28 | | 28 | | 34 | |
| 3. | 7 | | 7 | | 7 | | 7 | | 7 | | 10 | | 7 | | 7 | | 7 | |
| 4. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 5. | 7 | | 7 | | 7 | | 7 | | 7 | | 8 | | 7 | | 7 | | 7 | |
| 6. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 7. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 8. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 9. | 14 | | 10 | | 10 | | 10 | | 14 | | 10 | | 10 | | 10 | | 10 | |
| 10. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 11. | 20 | | 20 | | 20 | | 20 | | 20 | | 23 | | 20 | | 20 | | 20 | |
| 12. | 21 | | 0 | a | 21 | | 21 | | 0 | f | 0 | a | 0 | a | 21 | | 0 | f |
| 13. | 12 | | 13 | | 12 | | 12 | | 12 | | 13 | | 16 | | 12 | | 12 | |
| 14. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 15. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 16. | 7 | | 13 | | 7 | | 12 | | 7 | | 16 | | 13 | | 12 | | 12 | |
| 17. | 4 | | 4 | | 5 | | 5 | | 4 | | 4 | | 4 | | 5 | | 5 | |
| 18. | 18 | | 19 | | 0 | a | 0 | a | 18 | | 26 | | 19 | | 18 | | 26 | |
| 19. | 11 | | 9 | | 11 | | 11 | | 14 | | 8 | | 9 | | 11 | | 14 | |
| 20. | 13 | | 13 | | 16 | | 13 | | 13 | | 14 | | 13 | | 14 | | 13 | |
| 21. | 39 | | 30 | | 31 | | 31 | | 0 | f | 30 | | 30 | | 31 | | 0 | f |
| 22. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 23. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 24. | 9 | | 17 | | 9 | | 9 | | 9 | | 18 | | 17 | | 9 | | 9 | |
| 25. | 31 | | 30 | | 31 | | 30 | | 31 | | 43 | | 30 | | 28 | | 30 | |
| 26. | 4 | | 3 | | 3 | | 3 | | 4 | | 3 | | 3 | | 3 | | 3 | |
| 27. | 9 | | 12 | | 9 | | 12 | | 9 | | 12 | | 12 | | 9 | | 12 | |
| 28. | 15 | | 15 | | 15 | | 24 | | 15 | | 14 | | 15 | | 25 | | 26 | |
| 29. | 16 | | 16 | | 16 | | 16 | | 24 | | 16 | | 24 | | 16 | | 24 | |
| 30. | 16 | | 18 | | 16 | | 15 | | 16 | | 18 | | 18 | | 16 | | 15 | |
| 31. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 32. | 22 | | 29 | | 20 | | 29 | | 21 | | 44 | | 29 | | 29 | | 29 | |
| 33. | 5 | | 4 | | 4 | | 4 | | 5 | | 4 | | 4 | | 4 | | 4 | |
| 34. | 35 | | 30 | | 0 | a | 34 | | 38 | | 30 | | 32 | | 34 | | 39 | |
| 35. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 36. | 13 | | 12 | | 13 | | 13 | | 21 | | 12 | | 20 | | 13 | | 21 | |
| 37. | 52 | | 50 | | 52 | | 48 | | 56 | | 0 | f | 50 | | 48 | | 53 | |
| 38. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 39. | 13 | | 16 | | 22 | | 19 | | 13 | | 17 | | 16 | | 20 | | 19 | |
| 40. | 15 | | 11 | | 15 | | 15 | | 15 | | 14 | | 11 | | 15 | | 15 | |
| 41. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 42. | 11 | | 22 | | 11 | | 20 | | 11 | | 21 | | 22 | | 20 | | 16 | |
| 43. | 21 | | 19 | | 30 | | 23 | | 21 | | 30 | | 19 | | 23 | | 31 | |
| 44. | 18 | | 19 | | 20 | | 18 | | 19 | | 13 | | 22 | | 18 | | 19 | |
| 45. | 12 | | 15 | | 12 | | 12 | | 12 | | 17 | | 15 | | 12 | | 12 | |
| 46. | 31 | | 31 | | 31 | | 0 | a | 38 | | 30 | | 31 | | 40 | | 40 | |
| 47. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 48. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 49. | 4 | | 4 | | 4 | | 4 | | 4 | | 14 | | 4 | | 4 | | 4 | |
| 50. | 16 | | 14 | | 16 | | 13 | | 16 | | 14 | | 14 | | 17 | | 13 | |
| 51. | 24 | | 24 | | 26 | | 24 | | 24 | | 35 | | 24 | | 26 | | 24 | |
| 52. | 16 | | 18 | | 16 | | 20 | | 16 | | 16 | | 18 | | 20 | | 20 | |
| 53. | 12 | | 12 | | 25 | | 12 | | 12 | | 27 | | 12 | | 0 | f | 12 | |
| 54. | 21 | | 22 | | 25 | | 24 | | 21 | | 30 | | 22 | | 27 | | 24 | |
| 55. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 56. | 20 | | 23 | | 20 | | 24 | | 20 | | 25 | | 23 | | 22 | | 24 | |
| 57. | 4 | | 4 | | 9 | | 4 | | 4 | | 7 | | 4 | | 4 | | 4 | |
| 58. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 59. | 15 | | 0 | a | 24 | | 15 | | 15 | | 0 | a | 0 | a | 21 | | 15 | |
| 60. | 7 | | 7 | | 16 | | 7 | | 7 | | 17 | | 7 | | 9 | | 7 | |
| 61. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 62. | 13 | | 13 | | 19 | | 13 | | 13 | | 19 | | 13 | | 16 | | 13 | |
| 63. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 64. | 13 | | 13 | | 13 | | 13 | | 14 | | 14 | | 14 | | 13 | | 14 | |
| 65. | 39 | | 29 | | 36 | | 33 | | 0 | f | 30 | | 28 | | 33 | | 35 | |
| 66. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 67. | 12 | | 12 | | 12 | | 12 | | 12 | | 13 | | 12 | | 12 | | 12 | |
| 68. | 10 | | 10 | | 17 | | 10 | | 10 | | 12 | | 10 | | 9 | | 10 | |
| 69. | 16 | | 15 | | 15 | | 14 | | 15 | | 15 | | 14 | | 14 | | 14 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 70. | 18 | | 18 | | 18 | | 18 | | 24 | | 19 | | 24 | | 18 | | 24 | |
| 71. | 16 | | 10 | | 16 | | 16 | | 16 | | 10 | | 10 | | 16 | | 16 | |
| 72. | 10 | | 10 | | 14 | | 10 | | 10 | | 15 | | 10 | | 9 | | 10 | |
| 73. | 28 | | 16 | | 31 | | 26 | | 34 | | 26 | | 19 | | 26 | | 32 | |
| 74. | 8 | | 11 | | 8 | | 8 | | 8 | | 11 | | 11 | | 8 | | 8 | |
| 75. | 14 | | 14 | | 23 | | 15 | | 14 | | 18 | | 14 | | 15 | | 15 | |
| 76. | 18 | | 10 | | 15 | | 16 | | 18 | | 10 | | 10 | | 15 | | 15 | |
| 77. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 78. | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 79. | 9 | | 9 | | 11 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 80. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 9 | | 10 | |
| 81. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 82. | 11 | | 11 | | 11 | | 11 | | 11 | | 21 | | 11 | | 11 | | 11 | |
| 83. | 24 | | 21 | | 24 | | 30 | | 24 | | 21 | | 21 | | 24 | | 30 | |
| 84. | 4 | | 12 | | 4 | | 4 | | 4 | | 15 | | 12 | | 4 | | 4 | |
| 85. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 86. | 7 | | 7 | | 7 | | 7 | | 7 | | 0 | f | 7 | | 6 | | 7 | |
| 87. | 15 | | 13 | | 10 | | 14 | | 18 | | 0 | f | 12 | | 11 | | 16 | |
| 88. | 16 | | 15 | | 16 | | 15 | | 22 | | 15 | | 30 | | 15 | | 21 | |
| 89. | 4 | | 11 | | 18 | | 4 | | 4 | | 4 | | 11 | | 4 | | 4 | |
| 90. | 12 | | 12 | | 12 | | 12 | | 16 | | 12 | | 12 | | 11 | | 12 | |
| 91. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 92. | 10 | | 6 | | 11 | | 10 | | 10 | | 6 | | 6 | | 11 | | 10 | |
| 93. | 11 | | 11 | | 18 | | 11 | | 0 | f | 11 | | 0 | f | 11 | | 0 | f |
| 94. | 18 | | 18 | | 18 | | 18 | | 18 | | 18 | | 18 | | 18 | | 18 | |
| 95. | 25 | | 15 | | 21 | | 25 | | 0 | f | 20 | | 0 | f | 18 | | 0 | f |
| 96. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 97. | 5 | | 5 | | 5 | | 5 | | 5 | | 6 | | 5 | | 5 | | 5 | |
| 98. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 99. | 16 | | 17 | | 25 | | 16 | | 16 | | 0 | f | 20 | | 16 | | 16 | |
| 100. | 27 | | 27 | | 27 | | 25 | | 27 | | 26 | | 27 | | 26 | | 25 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 1285 | | 1217 | | 1328 | | 1247 | | 1203 | | 1283 | | 1245 | | 1290 | | 1276 | |

### B.8.3.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 2. | 0 | a | 39 | | 0 | a | 0 | a | 0 | a | 39 | | 39 | | 0 | a | 0 | a |
| 3. | 0 | a | 28 | | 0 | a | 38 | | 0 | a | 31 | | 28 | | 37 | | 38 | |
| 4. | 13 | | 15 | | 13 | | 13 | | 0 | f | 15 | | 15 | | 13 | | 0 | f |
| 5. | 37 | | 23 | | 37 | | 28 | | 45 | | 23 | | 28 | | 28 | | 34 | |
| 6. | 7 | | 7 | | 7 | | 7 | | 7 | | 10 | | 7 | | 7 | | 7 | |
| 7. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 8. | 0 | a | 28 | | 0 | f | 29 | | 0 | a | 29 | | 28 | | 25 | | 31 | |
| 9. | 7 | | 7 | | 7 | | 7 | | 7 | | 8 | | 7 | | 7 | | 7 | |
| 10. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 11. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a |
| 12. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 13. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 14. | 0 | a | 12 | | 0 | a | 12 | | 12 | | 12 | | 12 | | 12 | | 12 | |
| 15. | 0 | a | 32 | | 0 | a | 32 | | 0 | a | 50 | | 0 | f | 42 | | 0 | f |
| 16. | 0 | a | 19 | | 0 | a | 0 | a | 0 | a | 24 | | 19 | | 0 | a | 0 | a |
| 17. | 14 | | 10 | | 10 | | 10 | | 14 | | 10 | | 10 | | 10 | | 10 | |
| 18. | 0 | a | 22 | | 0 | a | 17 | | 17 | | 17 | | 22 | | 17 | | 17 | |
| 19. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 8 | | 9 | |
| 20. | 0 | a | 24 | | 0 | a | 24 | | 0 | a | 25 | | 24 | | 25 | | 24 | |
| 21. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 42 | | 0 | a | 32 | | 0 | a |
| 22. | 20 | | 20 | | 20 | | 20 | | 20 | | 24 | | 20 | | 20 | | 20 | |
| 23. | 0 | a | 11 | | 0 | a | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 24. | 21 | | 0 | a | 22 | | 21 | | 0 | f | 28 | | 0 | a | 21 | | 0 | f |
| 25. | 12 | | 13 | | 12 | | 12 | | 12 | | 13 | | 16 | | 12 | | 12 | |
| 26. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 27. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 28. | 7 | | 13 | | 7 | | 12 | | 7 | | 13 | | 13 | | 12 | | 12 | |
| 29. | 0 | a | 26 | | 0 | a | 25 | | 0 | a | 28 | | 26 | | 0 | a | 25 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30. | 0 | a | 25 | | 26 | | 26 | | 29 | | 27 | | 25 | | 26 | | 29 | |
| 31. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a |
| 32. | 0 | a | 36 | | 0 | a | 27 | | 0 | a | 32 | | 35 | | 27 | | 27 | |
| 33. | 4 | | 4 | | 5 | | 5 | | 4 | | 4 | | 4 | | 5 | | 5 | |
| 34. | 18 | | 19 | | 0 | a | 0 | a | 18 | | 19 | | 19 | | 18 | | 26 | |
| 35. | 11 | | 9 | | 11 | | 11 | | 14 | | 8 | | 9 | | 11 | | 14 | |
| 36. | 13 | | 13 | | 16 | | 13 | | 13 | | 13 | | 13 | | 16 | | 13 | |
| 37. | 39 | | 30 | | 31 | | 31 | | 0 | f | 30 | | 30 | | 31 | | 0 | f |
| 38. | 0 | a | 25 | | 0 | a | 0 | a | 0 | a | 25 | | 26 | | 23 | | 0 | a |
| 39. | 0 | a | 37 | | 0 | a | 0 | a | 0 | a | 37 | | 41 | | 0 | a | 0 | a |
| 40. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 41. | 0 | a | 24 | | 0 | a | 24 | | 0 | a | 40 | | 24 | | 0 | a | 24 | |
| 42. | 9 | | 9 | | 0 | a | 9 | | 9 | | 15 | | 9 | | 9 | | 9 | |
| 43. | 0 | a | 14 | | 21 | | 19 | | 19 | | 22 | | 14 | | 19 | | 19 | |
| 44. | 9 | | 17 | | 9 | | 9 | | 9 | | 16 | | 17 | | 9 | | 9 | |
| 45. | 0 | a | 39 | | 0 | a | 39 | | 0 | a | 39 | | 41 | | 39 | | 51 | |
| 46. | 31 | | 30 | | 31 | | 30 | | 31 | | 30 | | 30 | | 28 | | 30 | |
| 47. | 4 | | 3 | | 3 | | 3 | | 4 | | 3 | | 3 | | 3 | | 3 | |
| 48. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 15 | | 0 | a | 15 | | 0 | a |
| 49. | 0 | a | 0 | a | 0 | a | 15 | | 0 | a | 22 | | 0 | a | 15 | | 15 | |
| 50. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a |
| 51. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 52. | 9 | | 12 | | 9 | | 12 | | 9 | | 12 | | 12 | | 9 | | 12 | |
| 53. | 0 | a | 16 | | 16 | | 16 | | 0 | a | 16 | | 16 | | 16 | | 16 | |
| 54. | 0 | a | 18 | | 21 | | 0 | a | 0 | a | 18 | | 18 | | 20 | | 0 | a |
| 55. | 15 | | 15 | | 15 | | 24 | | 15 | | 19 | | 15 | | 20 | | 26 | |
| 56. | 0 | a | 35 | | 0 | a | 33 | | 0 | a | 33 | | 36 | | 33 | | 33 | |
| 57. | 0 | a | 29 | | 0 | a | 29 | | 0 | a | 40 | | 29 | | 29 | | 29 | |
| 58. | 0 | a | 31 | | 0 | a | 26 | | 0 | a | 34 | | 0 | f | 0 | a | 0 | f |
| 59. | 0 | a | 34 | | 0 | a | 34 | | 0 | a | 35 | | 37 | | 34 | | 37 | |
| 60. | 0 | a | 31 | | 0 | a | 33 | | 0 | a | 0 | f | 33 | | 33 | | 33 | |
| 61. | 16 | | 16 | | 0 | a | 16 | | 24 | | 21 | | 24 | | 16 | | 24 | |
| 62. | 16 | | 18 | | 16 | | 15 | | 16 | | 17 | | 18 | | 16 | | 15 | |
| 63. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 64. | 22 | | 29 | | 21 | | 29 | | 21 | | 29 | | 29 | | 28 | | 29 | |
| 65. | 0 | a | 18 | | 0 | a | 0 | a | 0 | a | 19 | | 18 | | 0 | a | 0 | a |
| 66. | 5 | | 4 | | 4 | | 4 | | 5 | | 4 | | 4 | | 4 | | 4 | |
| 67. | 0 | a | 29 | | 20 | | 23 | | 0 | a | 29 | | 31 | | 20 | | 26 | |
| 68. | 35 | | 30 | | 0 | a | 34 | | 38 | | 29 | | 32 | | 34 | | 39 | |
| 69. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 70. | 13 | | 12 | | 13 | | 13 | | 21 | | 12 | | 20 | | 13 | | 21 | |
| 71. | 52 | | 50 | | 52 | | 48 | | 56 | | 52 | | 50 | | 47 | | 53 | |
| 72. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 73. | 0 | a | 28 | | 27 | | 28 | | 0 | a | 37 | | 28 | | 23 | | 28 | |
| 74. | 0 | a | 27 | | 0 | a | 29 | | 0 | a | 0 | f | 27 | | 30 | | 0 | f |
| 75. | 0 | a | 37 | | 51 | | 0 | a | 0 | a | 31 | | 39 | | 37 | | 0 | a |
| 76. | 13 | | 16 | | 19 | | 19 | | 13 | | 17 | | 16 | | 19 | | 19 | |
| 77. | 15 | | 11 | | 15 | | 15 | | 15 | | 14 | | 11 | | 15 | | 15 | |
| 78. | 0 | a | 20 | | 21 | | 20 | | 0 | a | 24 | | 25 | | 20 | | 25 | |
| 79. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 80. | 0 | a | 11 | | 14 | | 11 | | 0 | a | 17 | | 11 | | 14 | | 11 | |
| 81. | 0 | a | 0 | a | 30 | | 0 | a | 0 | a | 41 | | 56 | | 33 | | 0 | f |
| 82. | 11 | | 22 | | 21 | | 20 | | 11 | | 15 | | 22 | | 20 | | 16 | |
| 83. | 21 | | 19 | | 29 | | 23 | | 21 | | 31 | | 19 | | 28 | | 31 | |
| 84. | 18 | | 19 | | 18 | | 18 | | 19 | | 24 | | 22 | | 21 | | 19 | |
| 85. | 0 | a | 16 | | 0 | a | 20 | | 0 | a | 33 | | 16 | | 20 | | 20 | |
| 86. | 0 | a | 0 | a | 0 | f | 0 | a | 0 | a | 39 | | 0 | a | 0 | a | 0 | a |
| 87. | 0 | a | 15 | | 33 | | 0 | a | 0 | a | 28 | | 27 | | 28 | | 0 | a |
| 88. | 12 | | 31 | | 12 | | 12 | | 12 | | 15 | | 15 | | 19 | | 12 | |
| 89. | 31 | | 26 | | 31 | | 0 | a | 38 | | 31 | | 31 | | 0 | f | 40 | |
| 90. | 0 | a | 7 | | 0 | a | 0 | a | 0 | a | 28 | | 26 | | 0 | a | 18 | |
| 91. | 7 | | 4 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 92. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 93. | 4 | | 14 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 94. | 16 | | 9 | | 0 | f | 13 | | 16 | | 20 | | 14 | | 14 | | 13 | |
| 95. | 0 | a | 44 | | 12 | | 9 | | 0 | a | 14 | | 9 | | 9 | | 9 | |
| 96. | 0 | a | 24 | | 49 | | 37 | | 0 | a | 45 | | 0 | f | 36 | | 0 | f |
| 97. | 24 | | 18 | | 26 | | 24 | | 24 | | 24 | | 24 | | 26 | | 24 | |
| 98. | 16 | | 17 | | 16 | | 20 | | 16 | | 22 | | 18 | | 20 | | 20 | |
| 99. | 0 | a | 19 | | 17 | | 0 | a | 0 | a | 17 | | 17 | | 21 | | 0 | a |
| 100. | 0 | a | 28 | | 0 | a | 0 | a | 21 | | 19 | | 23 | | 0 | a | | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 703 | | 1620 | | 1011 | | 1353 | | 759 | | 1923 | | 1646 | | 1539 | | 1318 | |

## B.9 Problem Sample

### B.9.1 Controlled Problem Sample

The table below lists the problems in this sample. The tasks are numbered down the left hand side and the first line of each problem is the goal condition $G$ and the second is the initial state. So for example, task number 1 in this sample is:

$$G = \{at(ann, door47, room4), open(door57)\}$$
$$I = \{open(door67), closed(door57), open(door47), closed(door12), open(door56),$$
$$closed(door45), closed(door35), closed(door25), closed(door24),$$
$$open(door23), in\_room(big\_box, room4), in(albert, room2),$$
$$in(ann, room1), in\_room(box2, room3), in\_room(box1, room6)\}$$

The set of problems that could not be solved by all planning applications is:

$$\{1, 12, 21, 34, 37, 46, 53, 59, 65, 86, 87, 93, 95, 99\}$$

A sample of 86 problems that could be solved by all of the planning applications was formed by removing this set of tasks from the sample.

| | Task |
|---|---|
| 1. | at(ann , door47 , room4) , open(door57) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room2) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room6) |
| 2. | at(ann , door45 , room4) , closed(door25) , at_door(big_box , door25 , room5) , at_door(box2 , door56 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room7) , in_room(box2 , room7) , in_room(box1 , room6) |
| 3. | at(albert , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room1) |
| 4. | at_door(box1 , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room5) |
| 5. | at(ann , door47 , room4) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room3) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room7) |
| 6. | at_door(box1 , door12 , room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room1) |
| 7. | at(ann , door25 , room2) , at(albert , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room1) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room7) |
| 8. | in(albert , room1) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room1) , in(ann , room4) , in_room(box2 , room2) , in_room(box1 , room6) |
| 9. | at(albert , door12 , room2) , near(ann , box2) , at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room4) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room4) |
| 10. | at_door(big_box , door25 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room7) , in_room(box2 , room1) , in_room(box1 , room7) |
| 11. | in_room(box1 , room5) , at(ann , door45 , room5) , at_door(box2 , door56 , room5) , at_door(box1 , door57 , room5) , at(albert , door47 , room4) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room4) , in_room(box1 , room5) |
| 12. | in(albert , room6) , closed(door57) , at(ann , door47 , room4) , at_door(box1 , door35 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room4) |
| 13. | in_room(box1 , room4) , at(albert , door12 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room2) |
| 14. | at(ann , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room7) , in_room(box2 , room2) , in_room(box1 , room3) |
| 15. | closed(door67) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room5) , in(ann , room3) , in_room(box2 , room4) , in_room(box1 , room7) |
| 16. | open(door67) , closed(door47) , in(albert , room1) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room5) |
| 17. | closed(door24) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room5) , in_room(box2 , room1) , in_room(box1 , room3) |
| 18. | in_room(box1 , room3) , at(ann , door56 , room6) , at_door(big_box , door67 , room6) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room6) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 19. | closed(door45) , at(ann , door25 , room5) , open(door47) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room6) , in(ann , room2) , in_room(box2 , room5) , in_room(box1 , room2) |
| 20. | in(albert , room1) , at_door(box1 , door45 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room7) , in(ann , room3) , in_room(box2 , room6) , in_room(box1 , room4) |

| | Task |
|---|---|
| 21. | open(door57) , near(ann , box1) , at(albert , door57 , room5) , at_door(box2 , door24 , room4) , in_room(big_box , room4) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room6) |
| 22. | open(door67) , at(ann , door47 , room4) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room1) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room5) |
| 23. | at_door(box1 , door23 , room3) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room4) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room1) |
| 24. | at_door(box1 , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room1) , in_room(box2 , room1) , in_room(box1 , room5) |
| 25. | near(ann , box1) , in_room(big_box , room6) , at_door(box1 , door23 , room3) , at_door(box2 , door56 , room5) , in(albert , room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room7) , in(ann , room3) , in_room(box2 , room2) , in_room(box1 , room5) |
| 26. | in(ann , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room7) , in_room(box2 , room1) , in_room(box1 , room5) |
| 27. | at_door(big_box , door12 , room2) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room4) , in_room(box2 , room2) , in_room(box1 , room2) |
| 28. | closed(door35) , at_door(box1 , door23 , room2) , closed(door57) , near(ann , box2) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room3) , in(albert , room5) , in(ann , room6) , in_room(box2 , room4) , in_room(box1 , room6) |
| 29. | at_door(box1 , door24 , room2) , at(albert , door45 , room5) , at(ann , door47 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room7) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room6) |
| 30. | in(albert , room1) , at_door(box2 , door23 , room3) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room6) |
| 31. | near(ann , box2) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room3) , in_room(box2 , room3) , in_room(box1 , room3) |
| 32. | at_door(box2 , door23 , room3) , at_door(big_box , door35 , room3) , in(albert , room3) , at_door(box1 , door67 , room7) , at(ann , door47 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room7) |
| 33. | at(ann , door56 , room6) , closed(door24) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room7) , in_room(box2 , room4) , in_room(box1 , room1) |
| 34. | at(ann , door56 , room6) , at(albert , door56 , room6) , at_door(box1 , door25 , room2) , in_room(big_box , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room4) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room4) |
| 35. | at_door(box1 , door47 , room4) , open(door67) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room6) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room4) |
| 36. | at(ann , door56 , room6) , at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room2) , in(ann , room2) , in_room(box2 , room6) , in_room(box1 , room5) |
| 37. | at(ann , door57 , room7) , at_door(box1 , door23 , room2) , in_room(box2 , room2) , at_door(big_box , door12 , room1) , closed(door56) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room4) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room7) |
| 38. | near(ann , big_box) , closed(door57) , closed(door25) , in(albert , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room1) , in_room(box2 , room2) , in_room(box1 , room4) |
| 39. | at_door(box1 , door67 , room7) , open(door56) , open(door47) , in_room(box2 , room1) , at(ann , door47 , room4) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room4) |
| 40. | at_door(big_box , door23 , room2) , at(ann , door12 , room1) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room5) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room3) |

| | Task |
|---|---|
| 41. | at(albert , door47 , room7) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room7) |
| 42. | at_door(big_box , door56 , room5) , open(door57) , near(ann , box2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room2) |
| 43. | closed(door45) , in(albert , room6) , at_door(box1 , door25 , room5) , at(ann , door47 , room7) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 44. | closed(door24) , at(albert , door57 , room5) , at_door(big_box , door67 , room6) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room7) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room2) |
| 45. | in(ann , room2) , open(door25) , at(albert , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room2) |
| 46. | in_room(box2 , room6) , at(albert , door45 , room4) , at_door(box1 , door47 , room7) , in_room(big_box , room1) , at(ann , door35 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room4) , in(albert , room6) , in(ann , room5) , in_room(box2 , room7) , in_room(box1 , room6) |
| 47. | at(albert , door57 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room2) |
| 48. | closed(door25) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 49. | at_door(big_box , door24 , room4) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room2) , in(ann , room7) , in_room(box2 , room4) , in_room(box1 , room3) |
| 50. | open(door45) , at_door(box1 , door67 , room7) , at(albert , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room3) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 51. | in_room(box1 , room6) , in(albert , room1) , at_door(box2 , door23 , room3) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room7) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room3) |
| 52. | at_door(box1 , door67 , room6) , closed(door67) , in(albert , room6) , near(ann , box2) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room3) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room5) |
| 53. | at_door(box2 , door56 , room5) , in(albert , room1) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room7) , in(ann , room5) , in_room(box2 , room3) , in_room(box1 , room7) |
| 54. | at_door(box1 , door45 , room5) , at_door(big_box , door47 , room4) , in(ann , room3) , in(albert , room2) , open(door47) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room6) , in_room(box2 , room3) , in_room(box1 , room5) |
| 55. | at(albert , door67 , room7) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room1) , in_room(box2 , room4) , in_room(box1 , room6) |
| 56. | open(door45) , at_door(box1 , door57 , room7) , open(door57) , in_room(big_box , room6) , open(door67) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room7) , in_room(box2 , room1) , in_room(box1 , room4) |
| 57. | at_door(big_box , door67 , room6) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room7) , in(ann , room5) , in_room(box2 , room1) , in_room(box1 , room6) |
| 58. | open(door25) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room5) , in(ann , room5) , in_room(box2 , room2) , in_room(box1 , room5) |
| 59. | at(albert , door12 , room2) , open(door67) , at_door(box1 , door25 , room2) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room6) , in(ann , room4) , in_room(box2 , room5) , in_room(box1 , room3) |
| 60. | at_door(box1 , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room7) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room2) |

| | Task |
|---|---|
| 61. | in(albert , room1) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room1) , in_room(box2 , room2) , in_room(box1 , room3) |
| 62. | at_door(box2 , door56 , room5) , at_door(box1 , door57 , room5) , at(ann , door47 , room4) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room2) , in(ann , room6) , in_room(box2 , room6) , in_room(box1 , room4) |
| 63. | at(albert , door12 , room2) , near(ann , box2) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room2) , in(ann , room3) , in_room(box2 , room4) , in_room(box1 , room3) |
| 64. | at_door(box1 , door35 , room3) , at_door(box2 , door67 , room7) , at(albert , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room5) , in_room(box2 , room7) , in_room(box1 , room3) |
| 65. | in_room(box2 , room2) , at(ann , door47 , room7) , at_door(box1 , door23 , room3) , at(albert , door57 , room5) , in_room(big_box , room6) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room5) , in_room(box2 , room5) , in_room(box1 , room6) |
| 66. | in(ann , room5) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room5) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room7) |
| 67. | at_door(box2 , door56 , room5) , closed(door67) , at(ann , door23 , room2) , at(albert , door57 , room5) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room2) , in_room(box2 , room2) , in_room(box1 , room5) |
| 68. | at_door(big_box , door56 , room5) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room1) , in_room(box2 , room1) , in_room(box1 , room4) |
| 69. | open(door67) , at(ann , door12 , room1) , at(albert , door23 , room3) , in_room(box2 , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room5) , in(ann , room5) , in_room(box2 , room1) , in_room(box1 , room3) |
| 70. | open(door67) , at_door(big_box , door12 , room2) , at(albert , door35 , room5) , near(ann , box2) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room4) , in_room(box1 , room4) |
| 71. | near(ann , box1) , in(albert , room1) , at_door(box2 , door56 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room4) , in_room(box2 , room4) , in_room(box1 , room2) |
| 72. | at_door(box1 , door56 , room6) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room5) , in(ann , room4) , in_room(box2 , room5) , in_room(box1 , room7) |
| 73. | at(ann , door23 , room3) , at_door(box2 , door45 , room5) , at_door(big_box , door56 , room5) , open(door45) , in(albert , room1) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room1) , in(ann , room2) , in_room(box2 , room5) , in_room(box1 , room1) |
| 74. | at(albert , door57 , room7) , open(door67) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 75. | at(albert , door23 , room2) , at_door(big_box , door67 , room6) , near(ann , box2) , at_door(box2 , door25 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room3) , in_room(box2 , room6) , in_room(box1 , room1) |
| 76. | at(ann , door47 , room4) , at_door(box1 , door67 , room6) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room5) , in_room(box2 , room3) , in_room(box1 , room6) |
| 77. | open(door67) , at_door(box1 , door67 , room6) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room6) , in_room(box2 , room6) , in_room(box1 , room6) |
| 78. | in_room(box2 , room2) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room4) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room5) |
| 79. | at(albert , door57 , room5) , closed(door35) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room3) |
| 80. | open(door67) , at_door(box2 , door67 , room7) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room1) |

| | Task |
|---|---|
| 81. | closed(door45) , at(albert , door57 , room5) , closed(door23) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room4) , in_room(box2 , room2) , in_room(box1 , room4) |
| 82. | in(ann , room4) , at(albert , door12 , room2) , at_door(big_box , door47 , room7) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room2) , in(ann , room5) , in_room(box2 , room5) , in_room(box1 , room4) |
| 83. | at(albert , door56 , room6) , near(ann , box2) , at_door(box1 , door25 , room5) , at_door(box2 , door35 , room5) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room5) , in(ann , room6) , in_room(box2 , room3) , in_room(box1 , room7) |
| 84. | at_door(box2 , door25 , room5) , open(door67) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room3) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room1) |
| 85. | in(albert , room1) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room7) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room1) |
| 86. | at_door(big_box , door25 , room5) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room3) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room1) |
| 87. | at(albert , door57 , room5) , at_door(box1 , door12 , room2) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room4) , in(albert , room4) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room5) |
| 88. | at(albert , door12 , room1) , open(door24) , at(ann , door57 , room7) , in_room(big_box , room6) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room2) , in(ann , room4) , in_room(box2 , room1) , in_room(box1 , room2) |
| 89. | at_door(big_box , door25 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room7) |
| 90. | at(albert , door35 , room5) , in_room(box1 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room7) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room5) |
| 91. | in(albert , room1) , at(ann , door23 , room3) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room2) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room4) |
| 92. | at(albert , door45 , room4) , at_door(box1 , door23 , room2) , open(door57) , open(door67) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room7) , in_room(box2 , room2) , in_room(box1 , room2) |
| 93. | at_door(box1 , door24 , room4) , at(ann , door35 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room2) , in_room(box1 , room2) |
| 94. | at_door(box1 , door67 , room7) , at(albert , door57 , room5) , at(ann , door12 , room1) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room2) , in_room(box2 , room5) , in_room(box1 , room5) |
| 95. | at(ann , door47 , room4) , at(albert , door12 , room2) , at_door(big_box , door25 , room5) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room4) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room5) |
| 96. | at(albert , door12 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room5) |
| 97. | in(albert , room1) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room3) |
| 98. | closed(door67) , in(ann , room4) , at(albert , door45 , room4) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room2) , in(ann , room4) , in_room(box2 , room5) , in_room(box1 , room6) |
| 99. | in(ann , room6) , at_door(box1 , door56 , room5) , at(albert , door12 , room2) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room4) , in_room(box2 , room7) , in_room(box1 , room6) |
| 100. | closed(door12) , at_door(big_box , door12 , room1) , at(albert , door47 , room7) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room6) , in_room(box2 , room5) , in_room(box1 , room3) |

### B.9.2 Uncontrolled Problem Sample

| | Task |
|---|---|
| 1. | open(door67) |
| | open(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room3) |
| 2. | closed(door25) , in(ann , room3) , at(albert , door25 , room5) , at_door(big_box , door67 , room6) , at_door(box1 , door12 , room1) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room4) , in(ann , room4) , in_room(box2 , room4) , in_room(box1 , room7) |
| 3. | closed(door25) , at_door(box2 , door23 , room2) , in(ann , room3) , at_door(box1 , door23 , room2) , in_room(big_box , room6) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room2) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room4) |
| 4. | at(ann , door47 , room4) , open(door57) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room2) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room6) |
| 5. | at(ann , door45 , room4) , closed(door25) , at_door(big_box , door25 , room5) , at_door(box2 , door56 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room7) , in_room(box2 , room7) , in_room(box1 , room6) |
| 6. | at(albert , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room1) |
| 7. | at_door(box1 , door23 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room5) |
| 8. | closed(door47) , at(albert , door57 , room7) , open(door12) , at(ann , door57 , room7) , in_room(big_box , room6) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room5) , in(ann , room6) , in_room(box2 , room4) , in_room(box1 , room2) |
| 9. | at(ann , door47 , room4) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room3) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room7) |
| 10. | at_door(box1 , door12 , room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room1) |
| 11. | at_door(box2 , door56 , room5) , at_door(box1 , door23 , room3) , open(door12) , at_door(big_box , door25 , room5) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room4) , in(ann , room6) , in_room(box2 , room5) , in_room(box1 , room1) |
| 12. | at(ann , door25 , room2) , at(albert , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room1) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room7) |
| 13. | in(albert , room1) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room1) , in(ann , room4) , in_room(box2 , room2) , in_room(box1 , room6) |
| 14. | at(albert , door57 , room5) , at_door(box2 , door24 , room2) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room4) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room4) |
| 15. | at_door(box1 , door47 , room7) , closed(door56) , at_door(box2 , door47 , room7) , in_room(big_box , room3) , at(albert , door57 , room5) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room4) |
| 16. | closed(door24) , at_door(big_box , door35 , room3) , at(ann , door12 , room2) , at_door(box2 , door23 , room3) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room3) , in(ann , room2) , in_room(box2 , room6) , in_room(box1 , room7) |
| 17. | at(albert , door12 , room2) , near(ann , box2) , at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room4) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room4) |
| 18. | in(albert , room1) , at(ann , door67 , room6) , at_door(big_box , door25 , room5) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room6) , in_room(box2 , room4) , in_room(box1 , room4) |
| 19. | at_door(big_box , door25 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room7) , in_room(box2 , room1) , in_room(box1 , room7) |
| 20. | at_door(box2 , door67 , room6) , at(albert , door57 , room5) , in_room(big_box , room1) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room6) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |

| | Task |
|---|---|
| 21. | at_door(big_box , door25 , room5) , in_room(box2 , room2) , closed(door47) , at(albert , door12 , room2) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room1) , in_room(box2 , room6) , in_room(box1 , room6) |
| 22. | in_room(box1 , room5) , at(ann , door45 , room5) , at_door(box2 , door56 , room5) , at_door(box1 , door57 , room5) , at(albert , door47 , room4) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room4) , in_room(box1 , room5) |
| 23. | in(ann , room4) , at_door(big_box , door47 , room4) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room7) , in_room(box2 , room3) , in_room(box1 , room1) |
| 24. | in(albert , room6) , closed(door57) , at(ann , door47 , room4) , at_door(box1 , door35 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room1) , in_room(box2 , room5) , in_room(box1 , room4) |
| 25. | in_room(box1 , room4) , at(albert , door12 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room6) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room2) |
| 26. | at(ann , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room7) , in_room(box2 , room2) , in_room(box1 , room3) |
| 27. | closed(door67) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room5) , in(ann , room3) , in_room(box2 , room4) , in_room(box1 , room7) |
| 28. | open(door67) , closed(door47) , in(albert , room1) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room5) |
| 29. | in(albert , room1) , at_door(box1 , door67 , room6) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room2) , in_room(box2 , room3) , in_room(box1 , room1) |
| 30. | at(albert , door12 , room2) , near(ann , box2) , closed(door67) , at_door(big_box , door25 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room3) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room6) |
| 31. | at_door(box2 , door45 , room4) , at(albert , door12 , room2) , near(ann , box1) , in_room(big_box , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room1) , in(albert , room2) , in(ann , room2) , in_room(box2 , room6) , in_room(box1 , room2) |
| 32. | at(albert , door45 , room4) , at_door(big_box , door67 , room6) , at_door(box2 , door67 , room6) , in_room(big_box , room6) , at_door(box1 , door47 , room7) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room5) , in_room(box2 , room7) , in_room(box1 , room7) |
| 33. | closed(door24) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room5) , in_room(box2 , room1) , in_room(box1 , room3) |
| 34. | in_room(box1 , room3) , at(ann , door56 , room6) , at_door(big_box , door67 , room6) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room6) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 35. | closed(door45) , at(ann , door25 , room5) , open(door47) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room6) , in(ann , room2) , in_room(box2 , room5) , in_room(box1 , room2) |
| 36. | in(albert , room1) , at_door(box1 , door45 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room7) , in(ann , room3) , in_room(box2 , room6) , in_room(box1 , room4) |
| 37. | open(door57) , near(ann , box1) , at(albert , door57 , room5) , at_door(box2 , door24 , room4) , in_room(big_box , room4) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room6) |
| 38. | closed(door45) , at_door(box1 , door57 , room7) , at(albert , door12 , room2) , at_door(big_box , door57 , room5) , at(ann , door45 , room4) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room2) , in(ann , room7) , in_room(box2 , room4) , in_room(box1 , room6) |
| 39. | at_door(box2 , door24 , room2) , at_door(box1 , door57 , room5) , in(albert , room1) , at_door(big_box , door35 , room3) , at(ann , door23 , room2) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room7) |
| 40. | open(door67) , at(ann , door47 , room4) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room1) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room5) |

| | Task |
|---|---|
| 41. | in_room(big_box , room6) , at_door(box1 , door67 , room6) , open(door67) , at(ann , door47 , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room5) , in_room(box2 , room5) , in_room(box1 , room1) |
| 42. | at_door(box1 , door23 , room3) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room4) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room1) |
| 43. | at(ann , door24 , room4) , at_door(box1 , door56 , room6) , near(albert , box2) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room4) , in(ann , room5) , in_room(box2 , room4) , in_room(box1 , room5) |
| 44. | at_door(box1 , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room1) , in_room(box2 , room1) , in_room(box1 , room5) |
| 45. | in(albert , room4) , at(ann , door45 , room5) , at_door(big_box , door12 , room1) , in_room(box2 , room2) , in_room(box1 , room6) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room3) , in_room(box2 , room6) , in_room(box1 , room2) |
| 46. | near(ann , box1) , in_room(big_box , room6) , at_door(box1 , door23 , room3) , at_door(box2 , door56 , room5) , in(albert , room1) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room5) , in(albert , room7) , in(ann , room3) , in_room(box2 , room2) , in_room(box1 , room5) |
| 47. | in(ann , room4) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room7) , in_room(box2 , room1) , in_room(box1 , room5) |
| 48. | at_door(big_box , door25 , room5) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room6) , in(ann , room3) , in_room(box2 , room7) , in_room(box1 , room4) |
| 49. | at_door(box1 , door67 , room6) , at_door(box2 , door25 , room2) , near(ann , box2) , in(albert , room1) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room2) , in_room(box2 , room2) , in_room(box1 , room4) |
| 50. | at_door(box2 , door12 , room1) , at_door(box1 , door67 , room7) , in(albert , room1) , at(ann , door57 , room5) |
| | open(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room2) , in(ann , room5) , in_room(box2 , room4) , in_room(box1 , room3) |
| 51. | in(ann , room4) , near(ann , box2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room1) |
| 52. | at_door(big_box , door12 , room2) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room1) , in(ann , room4) , in_room(box2 , room2) , in_room(box1 , room2) |
| 53. | at(albert , door12 , room1) , at_door(box1 , door25 , room2) , in(ann , room5) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room7) |
| 54. | near(ann , box2) , at_door(box1 , door23 , room3) , at_door(box2 , door56 , room5) , at(albert , door67 , room6) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room3) , in(albert , room6) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room4) |
| 55. | closed(door35) , at_door(box1 , door23 , room2) , closed(door57) , near(ann , box2) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room3) , in(albert , room5) , in(ann , room6) , in_room(box2 , room4) , in_room(box1 , room6) |
| 56. | at_door(big_box , door47 , room4) , closed(door47) , at(albert , door67 , room7) , in_room(box2 , room1) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room1) , in(ann , room5) , in_room(box2 , room3) , in_room(box1 , room2) |
| 57. | at_door(big_box , door67 , room7) , at_door(box1 , door67 , room7) , in(albert , room6) , in(ann , room4) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room3) , in(ann , room1) , in_room(box2 , room7) , in_room(box1 , room4) |
| 58. | at_door(box2 , door35 , room3) , at(ann , door12 , room1) , at_door(box1 , door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room7) , in(ann , room6) , in_room(box2 , room6) , in_room(box1 , room6) |
| 59. | at_door(big_box , door45 , room4) , at_door(box2 , door35 , room3) , closed(door67) , in(albert , room1) , at_door(box1 , door67 , room6) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room3) , in(albert , room2) , in(ann , room5) , in_room(box2 , room7) , in_room(box1 , room6) |
| 60. | at(albert , door57 , room5) , closed(door35) , at_door(box1 , door45 , room5) , in_room(box2 , room2) , at_door(big_box , door12 , room1) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room3) , in(ann , room3) , in_room(box2 , room3) , in_room(box1 , room1) |

| | Task |
|---|---|
| 61. | at_door(box1 , door24 , room2) , at(albert , door45 , room5) , at(ann , door47 , room4) |
| | closed(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room6) , in(albert , room7) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room6) |
| 62. | in(albert , room1) , at_door(box2 , door23 , room3) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room6) |
| 63. | near(ann , box2) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room6) , in(albert , room6) , in(ann , room3) , in_room(box2 , room3) , in_room(box1 , room3) |
| 64. | at_door(box2 , door23 , room3) , at_door(big_box , door35 , room3) , in(albert , room3) , at_door(box1 , door67 , room7) , at(ann , door47 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room7) |
| 65. | at_door(box1 , door67 , room6) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room2) , in(ann , room4) , in_room(box2 , room4) , in_room(box1 , room1) |
| 66. | at(ann , door56 , room6) , closed(door24) |
| | closed(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room1) , in(ann , room7) , in_room(box2 , room4) , in_room(box1 , room1) |
| 67. | at_door(box2 , door35 , room5) , at_door(big_box , door47 , room7) , at(ann , door23 , room3) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room1) |
| 68. | at(ann , door56 , room6) , at(albert , door56 , room6) , at_door(box1 , door25 , room2) , in_room(big_box , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room6) , in(albert , room4) , in(ann , room7) , in_room(box2 , room5) , in_room(box1 , room4) |
| 69. | at_door(box1 , door47 , room4) , open(door67) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , open(door25) , open(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room6) , in(ann , room1) , in_room(box2 , room3) , in_room(box1 , room4) |
| 70. | at(ann , door56 , room6) , at_door(box2 , door57 , room7) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room1) , in(albert , room2) , in(ann , room2) , in_room(box2 , room6) , in_room(box1 , room5) |
| 71. | at(ann , door57 , room7) , at_door(box1 , door23 , room2) , in_room(box2 , room2) , at_door(big_box , door12 , room1) , closed(door56) |
| | closed(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room4) , in(ann , room4) , in_room(box2 , room3) , in_room(box1 , room7) |
| 72. | near(ann , big_box) , closed(door57) , closed(door25) , in(albert , room2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room1) , in_room(box2 , room2) , in_room(box1 , room4) |
| 73. | near(ann , box1) , in_room(box2 , room4) , near(albert , box2) , at_door(box1 , door35 , room3) |
| | closed(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room7) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room4) |
| 74. | open(door56) , at_door(box1 , door23 , room2) , at_door(box2 , door67 , room6) , in_room(big_box , room6) , at(albert , door25 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room2) |
| 75. | at(ann , door56 , room6) , at_door(box1 , door12 , room1) , in(albert , room1) , at_door(box2 , door24 , room2) , at_door(big_box , door23 , room2) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room3) , in(albert , room7) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room1) |
| 76. | at_door(box1 , door67 , room7) , open(door56) , open(door47) , in_room(box2 , room1) , at(ann , door47 , room4) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room7) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room4) |
| 77. | at_door(big_box , door23 , room2) , at(ann , door12 , room1) |
| | open(door67) , open(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , closed(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room5) , in(ann , room6) , in_room(box2 , room2) , in_room(box1 , room3) |
| 78. | at_door(box2 , door35 , room3) , in_room(big_box , room1) , at(ann , door12 , room1) , at(albert , door57 , room5) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room6) , in_room(box2 , room4) , in_room(box1 , room5) |
| 79. | at(albert , door47 , room7) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room4) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room7) |
| 80. | at_door(big_box , door23 , room3) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room2) , in(ann , room3) , in_room(box2 , room5) , in_room(box1 , room1) |

| | Task |
|---|---|
| 81. | at_door(box2 , door12 , room2) , in(ann , room1) , at_door(big_box , door67 , room6) , open(door47) , at(albert , door45 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , open(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room5) , in_room(box2 , room6) , in_room(box1 , room5) |
| 82. | at_door(big_box , door56 , room5) , open(door57) , near(ann , box2) |
| | open(door67) , closed(door57) , closed(door47) , open(door12) , open(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room5) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room2) |
| 83. | closed(door45) , in(albert , room6) , at_door(box1 , door25 , room5) , at(ann , door47 , room7) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 84. | closed(door24) , at(albert , door57 , room5) , at_door(big_box , door67 , room6) |
| | closed(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room7) , in(albert , room7) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room2) |
| 85. | closed(door23) , at_door(box2 , door12 , room2) , at_door(big_box , door25 , room5) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room3) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room5) |
| 86. | at(albert , door56 , room5) , open(door24) , near(ann , box2) , at_door(big_box , door45 , room5) , at_door(box1 , door23 , room3) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room2) , in(albert , room4) , in(ann , room3) , in_room(box2 , room6) , in_room(box1 , room1) |
| 87. | at_door(big_box , door67 , room7) , in_room(box2 , room4) , at_door(box1 , door47 , room7) , at(albert , door35 , room5) , at(ann , door12 , room1) |
| | open(door67) , open(door57) , open(door47) , open(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room2) , in(albert , room5) , in(ann , room2) , in_room(box2 , room4) , in_room(box1 , room2) |
| 88. | in(ann , room2) , open(door25) , at(albert , door57 , room5) |
| | open(door67) , closed(door57) , open(door47) , closed(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , open(door24) , open(door23) , in_room(big_box , room1) , in(albert , room1) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room2) |
| 89. | in_room(box2 , room6) , at(albert , door45 , room4) , at_door(box1 , door47 , room7) , in_room(big_box , room1) , at(ann , door35 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , closed(door35) , open(door25) , open(door24) , open(door23) , in_room(big_box , room4) , in(albert , room6) , in(ann , room5) , in_room(box2 , room7) , in_room(box1 , room6) |
| 90. | open(door67) , at_door(box2 , door35 , room3) , in_room(box1 , room4) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , closed(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room4) , in(ann , room6) , in_room(box2 , room1) , in_room(box1 , room2) |
| 91. | at(albert , door57 , room5) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room2) |
| 92. | closed(door25) |
| | open(door67) , open(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room1) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 93. | at_door(big_box , door24 , room4) |
| | open(door67) , closed(door57) , open(door47) , open(door12) , closed(door56) , closed(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room7) , in(albert , room2) , in(ann , room7) , in_room(box2 , room4) , in_room(box1 , room3) |
| 94. | open(door45) , at_door(box1 , door67 , room7) , at(albert , door57 , room5) |
| | closed(door67) , open(door57) , open(door47) , open(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room7) , in(albert , room3) , in(ann , room6) , in_room(box2 , room7) , in_room(box1 , room6) |
| 95. | at_door(box1 , door67 , room6) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , closed(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room4) , in(albert , room2) , in(ann , room5) , in_room(box2 , room5) , in_room(box1 , room3) |
| 96. | at_door(box2 , door56 , room6) , at_door(box1 , door23 , room3) , at(ann , door57 , room5) , open(door47) , at_door(big_box , door67 , room6) |
| | closed(door67) , closed(door57) , open(door47) , closed(door12) , open(door56) , closed(door45) , open(door35) , closed(door25) , closed(door24) , closed(door23) , in_room(big_box , room4) , in(albert , room6) , in(ann , room1) , in_room(box2 , room7) , in_room(box1 , room7) |
| 97. | in_room(box1 , room6) , in(albert , room1) , at_door(box2 , door23 , room3) |
| | closed(door67) , open(door57) , closed(door47) , open(door12) , open(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , closed(door23) , in_room(big_box , room1) , in(albert , room7) , in(ann , room2) , in_room(box2 , room1) , in_room(box1 , room3) |
| 98. | at_door(box1 , door67 , room6) , closed(door67) , in(albert , room6) , near(ann , box2) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , open(door25) , closed(door24) , open(door23) , in_room(big_box , room5) , in(albert , room3) , in(ann , room4) , in_room(box2 , room6) , in_room(box1 , room5) |
| 99. | in(ann , room3) , in_room(box2 , room4) |
| | open(door67) , open(door57) , closed(door47) , closed(door12) , closed(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room6) , in(ann , room3) , in_room(box2 , room1) , in_room(box1 , room2) |
| 100. | at_door(box1 , door24 , room2) , at_door(big_box , door45 , room5) , in(albert , room1) , at(ann , door56 , room5) |
| | open(door67) , closed(door57) , closed(door47) , closed(door12) , open(door56) , open(door45) , open(door35) , closed(door25) , closed(door24) , open(door23) , in_room(big_box , room2) , in(albert , room1) , in(ann , room2) , in_room(box2 , room3) , in_room(box1 , room7) |

# Appendix C

# *WW*: The Warehouse World

This world is a refinement of the warehouse world used in [46]. A driver, crane, truck and crates populate a warehouse that requires shelves to be stacked. The warehouse is densely populated and much of the planning problem is in moving crates to clear paths. The figure below is a representation of the planning domain.

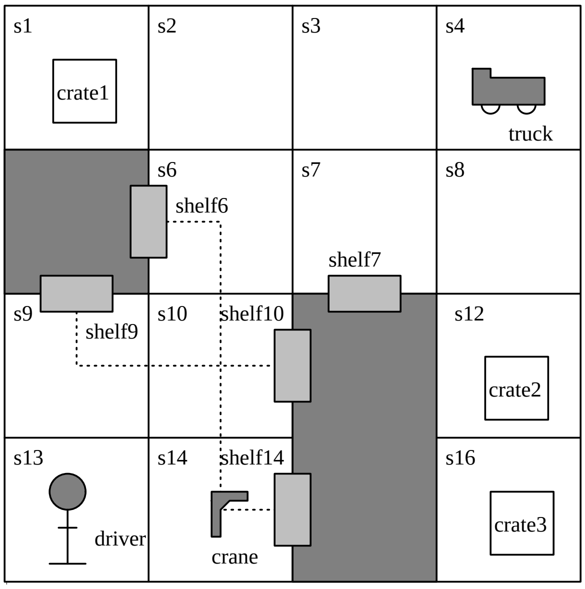


Figure C.1: *WW*: The Warehouse World

The state of the world that is shown in the figure is expressed, in the FM input language, as follows:

$$\{crane\_at(s14), truck\_at(s4), truck\_unloaded, clear(s2), clear(s3), clear(s6), clear(s7), clear(s8), clear(s9), clear(s10), clear(s14), driver\_on\_floor(s13), driver\_at(s13), at(crate3, s16), on\_floor(crate3, s16), at(crate2, s12), on\_floor(crate2, s12), at(crate1, s1), on\_floor(crate1, s1), crane\_unloaded\}$$

The table below gives information on the relative size and structure of the modelled planning domain. Column **ops** gives the number of operator instances, **preds** the number of predicate instances, and **states** the approximate number of world states. The final column **rules** gives the combined number of parameterised rule and axiom schemas that were posed by the modeller to specify relations between literals, and to specify inconsistent states.

| | ops | preds | states | rules |
|---|---|---|---|---|
| WW: warehouse | 482 | 290 | $2 \times 10^6$ | 53 |

## C.1 *OS*: Domain Operators

$(load(B, X, Y),$
$\quad\{at(B, X)\},$
$\quad\{type\_of(X, space), type\_of(Y, space), type\_of(B, crate), next(X, Y)\},$
$\quad\{in\_truck, on\_floor(B, X), truck\_unloaded, truck\_at(Y)\},$
$\quad\{at(B, Y), truck\_loaded(B, Y), clear(X)\},$
$\quad\{\},$
$\quad\{on\_floor(B, X), at(B, X), truck\_unloaded\})$

$(unload(B, Y, X),$
$\quad\{\},$
$\quad\{type\_of(X, space), type\_of(Y, space), type\_of(B, crate), next(X, Y)\},$
$\quad\{clear(X), in\_truck, truck\_loaded(B, Y)\},$
$\quad\{truck\_unloaded, on\_floor(B, X), at(B, X)\},$
$\quad\{\},$
$\quad\{truck\_loaded(B, Y), clear(X), on\_floor(B, Y), at(B, Y)\})$

$(get\_in(X, Y),$
$\quad\{truck\_at(Y)\},$
$\quad\{type\_of(X, space), type\_of(Y, space), next(X, Y)\},$
$\quad\{driver\_at(X), driver\_on\_floor(X), truck\_at(Y)\},$
$\quad\{in\_truck\},$
$\quad\{clear(X), driver\_at(Y)\},$
$\quad\{driver\_on\_floor(X), driver\_at(X)\})$

$(get\_out(Y, X),$
$\quad\{\},$
$\quad\{type\_of(X, space), type\_of(Y, space), next(X, Y)\},$
$\quad\{in\_truck, clear(X), truck\_at(Y)\},$
$\quad\{driver\_on\_floor(X)\},$
$\quad\{driver\_at(X)\},$
$\quad\{in\_truck, clear(X), driver\_at(Y)\})$

$(drive\_load(B, X, Y),$

$\{\},$

$\{type\_of(X, space), type\_of(Y, space), type\_of(B, crate), next(X, Y)\},$

$\{in\_truck, truck\_loaded(B, X), clear(Y)\},$

$\{truck\_loaded(B, Y), at(B, Y)\},$

$\{driver\_at(Y), clear(X), truck\_at(Y)\},$

$\{driver\_at(X), truck\_loaded(B, X), at(B, X), truck\_at(X), clear(Y)\})$

$(drive(X, Y),$

$\{\},$

$\{type\_of(X, space), type\_of(Y, space), next(X, Y)\},$

$\{in\_truck, truck\_at(X), truck\_unloaded, clear(Y)\},$

$\{truck\_at(Y), clear(X)\},$

$\{driver\_at(Y)\},$

$\{driver\_at(X), truck\_at(X), clear(Y)\})$

$(walk(X, Y),$

$\{\},$

$\{type\_of(X, space), type\_of(Y, space), next(X, Y)\},$

$\{driver\_at(X), driver\_on\_floor(X), clear(Y)\},$

$\{driver\_at(Y)\},$

$\{driver\_on\_floor(Y), clear(X)\},$

$\{driver\_at(X), driver\_on\_floor(X), clear(Y)\})$

$(lift\_up(B, Y),$

$\{\},$

$\{type\_of(B, crate), pickup\_point(Y)\},$

$\{crane\_unloaded, crane\_at(Y), on\_floor(B, Y)\},$

$\{above\_floor(B, Y), crane\_loaded(B, Y)\},$

$\{\},$

$\{crane\_unloaded, on\_floor(B, Y)\})$

$(lift\_down(B, Y),$

$\{above\_floor(B, Y), at(B, Y)\},$

$\{type\_of(B, crate), pickup\_point(Y)\},$

$\{above\_floor(B, Y), crane\_loaded(B, Y)\},$

$\{on\_floor(B, Y), crane\_unloaded\},$

$\{\},$

$\{crane\_loaded(B, Y), above\_floor(B, Y)\})$

$(move\_crane(X, Y),$
$\quad \{\},$
$\quad \{pickup\_point(X), pickup\_point(Y), connect(X, Y)\},$
$\quad \{crane\_at(X), crane\_unloaded\},$
$\quad \{crane\_at(Y)\},$
$\quad \{\},$
$\quad \{crane\_at(X)\})$

$(crane\_stack(B, X, S),$
$\quad \{\},$
$\quad \{pickup\_point(X), connect(X, S), type\_of(S, shelf), type\_of(B, crate)\},$
$\quad \{crane\_loaded(B, X)\},$
$\quad \{stacked(B, S), crane\_unloaded\},$
$\quad \{above\_floor(B, S)\},$
$\quad \{crane\_loaded(B, X), above\_floor(B, X)\})$

$(crane\_unstack(B, Y, S),$
$\quad \{stacked(B, S), above\_floor(B, Y)\},$
$\quad \{type\_of(S, shelf), type\_of(Y, space), type\_of(B, crate), connect(S, Y)\},$
$\quad \{crane\_unloaded, stacked(B, S), above\_floor(B, Y), crane\_at(Y)\},$
$\quad \{crane\_loaded(B, Y)\},$
$\quad \{\},$
$\quad \{crane\_unloaded, stacked(B, S)\})$

$(truck\_stack(B, X, Y, Z),$
$\quad \{\},$
$\quad \{type\_of(X, space), type\_of(Y, space), type\_of(Z, shelf),$
$\qquad type\_of(B, crate), between(X, Y, Z)\},$
$\quad \{truck\_at(X), truck\_unloaded, in\_truck, on\_floor(B, Y)\},$
$\quad \{stacked(B, Z)\},$
$\quad \{above\_floor(B, Y)\},$
$\quad \{on\_floor(B, Y)\})$

$(truck\_unstack(B, X, Y, Z),$
$\quad \{stacked(B, Z)\},$
$\quad \{type\_of(X, space), type\_of(Y, space), type\_of(Z, shelf),$
$\qquad type\_of(B, crate), between(X, Y, Z)\},$
$\quad \{truck\_at(X), truck\_unloaded, in\_truck, stacked(B, Z),$
$\qquad above\_floor(B, Z)\},$
$\quad \{on\_floor(B, Y)\},$
$\quad \{\},$
$\quad \{stacked(B, Z), above\_floor(B, Y)\})$

## C.2 *ES*: Environmental Information

The set of environmental information, $ES$, for this domain is given below. For ease of presentation in this section we have divided it into sub-sets, given below is the set of unchanging predicates and later are a set of rules that specify relations between predicates in the domain model.

$\{type\_of(crate1, crate), type\_of(crate2, crate), type\_of(crate3, crate),$
$type\_of(s1, space), type\_of(s2, space), type\_of(s3, space),$
$type\_of(s4, space), type\_of(s6, space), type\_of(s7, space),$
$type\_of(s8, space), type\_of(s9, space), type\_of(s10, space),$
$type\_of(s12, space), type\_of(s13, space), type\_of(s14, space),$
$type\_of(s16, space), type\_of(shelf6, shelf), type\_of(shelf7, shelf),$
$type\_of(shelf9, shelf), type\_of(shelf10, shelf), type\_of(shelf14, shelf),$
$pickup\_point(s6), pickup\_point(s9), pickup\_point(s10), pickup\_point(s14),$
$connect(s6, shelf6), connect(shelf6, s6), connect(s10, s6), connect(s6, s10),$
$connect(s10, s9), connect(s9, s10), connect(s9, shelf9), connect(shelf9, s9),$
$connect(s14, s10), connect(s10, s14), connect(s10, shelf10),$
$connect(shelf10, s10), connect(s14, shelf14), connect(shelf14, s14),$
$next(s1, s2), next(s2, s1), next(s2, s3), next(s3, s2), next(s4, s3),$
$next(s3, s4), next(s6, s7), next(s7, s6), next(s8, s7), next(s7, s8),$
$next(s9, s10), next(s10, s9), next(s14, s13), next(s13, s14), next(s13, s9),$
$next(s9, s13), next(s6, s2), next(s2, s6), next(s6, s10), next(s10, s6),$
$next(s14, s10), next(s10, s14), next(s3, s7), next(s7, s3), next(s8, s4),$
$next(s4, s8), next(s8, s12), next(s12, s8), next(s16, s12), next(s12, s16),$
$between(s7, s6, shelf6), between(s3, s7, shelf7), between(s13, s9, shelf9),$
$between(s13, s14, shelf14), between(s9, s10, shelf10)\}$

There are also rules which specify the relation between certain predicates in the domain. Each rule is expressed as a pair $(a, b)$ where $b$ can be inferred from $a$. For this modelled planning domain the rules for achievable predicates (as defined on page 15) in $ES$ is:

$\{(truck\_loaded(C, S), at(C, S)),$
$(truck\_loaded(C, S), truck\_at(C, S)),$
$(on\_floor(C, S), at(C, S)),$
$(driver\_on\_floor(S), driver\_at(S)),$
$(crane\_loaded(C, S), crane\_at(S)),$
$(crane\_loaded(C, S), above\_floor(C, S)),$
$(crane\_loaded(C, S), at(C, S)),$
$(stacked(C, Shelf), at(C, Shelf)),$
$(above\_floor(C, S), at(C, S)),$
$(\{in\_truck, truck\_at(S)\}, driver\_at(S))\}$

Finally, there are rules that relate unchanging predicates as follows:

$\{(next(X, Y), ne(X, Y)),$
$(next(X3, Y3), next(Y3, X3)),$
$(between(X1, X2, \_), next(X1, X2)),$
$(connect(X4, X5), connect(X5, X4))\}$

## C.3 *IN*: Negative State Invariant

The state invariant for this domain model was supplied by the domain modeller. The set *IN* is as follows:

$\{(truck\_loaded(C, \_), stacked(C, \_)),$
$(truck\_loaded(C, \_), truck\_unloaded),$
$(truck\_loaded(C, \_), truck\_loaded(D, \_)),$
$(truck\_loaded(C, S), truck\_loaded(C, T)),$
$(truck\_at(S), on\_floor(\_, S)),$
$(truck\_at(C), truck\_loaded(\_, D)),$
$(truck\_at(C), truck\_at(D)),$
$(truck\_at(S), on\_floor(C, S)),$
$(on\_floor(C, S), on\_floor(C, T)),$
$(on\_floor(C, S), on\_floor(D, S)),$
$(at(C, S), on\_floor(D, S)),$
$(at(C, S), at(D, S)),$
$(at(C, S), at(C, T)),$
$(at(C, S), truck\_loaded(C, T)),$
$(crane\_at(S), crane\_loaded(\_, T)),$
$(crane\_at(S), crane\_at(T)),$
$(crane\_loaded(C, \_), crane\_loaded(D, \_)),$
$(crane\_loaded(C, S), crane\_loaded(C, T)),$
$(crane\_loaded(C, \_), on\_floor(C, \_)),$
$(stacked(C, \_), on\_floor(C, \_)),$
$(stacked(C, S), stacked(C, T)),$
$(above\_floor(C, \_), on\_floor(C, \_)),$
$(above\_floor(C, S), above\_floor(C, T)),$
$(driver\_at(S), driver\_at(T)),$
$(driver\_at(S), on\_floor(C, S)),$
$(driver\_on\_floor(S), driver\_on\_floor(T)),$
$(in\_truck, driver\_on\_floor(\_)),$
$(in\_truck, truck\_at(S), driver\_at(T)),$
$(clear(S), on\_floor(\_, S)),$
$(clear(S), at(\_, S)),$
$(clear(S), driver\_at(S)),$

$(clear(S), truck\_at(S)),$
$(clear(S), truck\_loaded(\_, S))\}$

## C.4 $IP_{sort}$: Positive State Invariant

The positive state invariant for this modelled planning domain is as follows:

$$IP_{crate} = \{\{on\_floor(C,S), at(C,S)\},$$
$$\{truck\_loaded(C,S), at(C,S)\},$$
$$\{crane\_loaded(C,S), above\_floor(C,S), at(C,S)\},$$
$$\{stacked(C,S'), above\_floor(C,S'), at(C,S')\}\}$$

$$IP_{truck} = \{\{truck\_loaded(C,S), truck\_at(S)\}, \{truck\_unloaded, truck\_at(s)\}\}$$

$$IP_{driver} = \{\{driver\_on\_floor(S), driver\_at(S)\}, \{in\_truck, driver\_at(S)\}\}$$

$$IP_{crane} = \{\{crane\_loaded(C,S), crane\_at(S)\}, \{crane\_unloaded, crane\_at(S)\}\}$$

## C.5 PRECEDE Goal Order: $P$

$$P = \{$$
$$(clear(S), truck\_loaded(C,S)),$$
$$(truck\_loaded(C,S), truck\_at(S)),$$
$$(truck\_at(S), driver\_at(S)), (driver\_at(S), driver\_on\_floor(S)),$$
$$(in\_truck, driver\_at(S)),$$
$$(crane\_unloaded, crane\_at(S)), (crane\_at(S), crane\_loaded(C,S))\}$$

## C.6 $\mathcal{M}$: PROPOSE Macros

For this modelled planning domain, the set $\mathcal{M}$ is made up of a set $M_s$ where $s$ can be one of the sorts listed in section C.4. The elements of each $M_s$ are pairs of the form $((a,b),c)$ where $a$ and $b$ are a task configuration, $a$ is a start situation for objects of sort $s$, $b$ is the end situation and $c$ is the set of macros for this task configuration.

For simplicity, the anonymous variable "_" is used to represent all variables in predicate arguments in the macro. When the macros are used during planning, these variable arguments must be instantiated to objects of the correct sort. The planning application that accepts a domain model including these compiled macros must be able to bind the variables correctly.

$M_{crane} = \{$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{crane\_at}(\_)\}, \{\mathit{crane\_loaded}(\_,\_), \mathit{crane\_at}(\_)\}), \{$

$[\mathit{crane\_stack}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*, \mathit{crane\_unstack}(\_,\_,\_)],$

$[\mathit{crane\_stack}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*, \mathit{lift\_up}(\_,\_)],$

$[\mathit{lift\_down}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*, \mathit{crane\_unstack}(\_,\_,\_)],$

$[\mathit{lift\_down}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*, \mathit{lift\_up}(\_,\_)]\}),$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{crane\_at}(\_)\}, \{\mathit{crane\_unloaded}, \mathit{crane\_at}(\_)\}), \{$

$[\mathit{crane\_stack}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*],$

$[\mathit{lift\_down}(\_,\_,\_,\_), [\mathit{move\_crane}(\_,\_)]^*]\}),$

$((\{\mathit{crane\_unloaded}, \mathit{crane\_at}(\_)\}, \{\mathit{crane\_loaded}(\_,\_), \mathit{crane\_at}(\_)\}), \{$

$[[\mathit{move\_crane}(\_,\_)]^*, \mathit{lift\_up}(\_,\_)],$

$[[\mathit{move\_crane}(\_,\_)]^*, \mathit{crane\_unstack}(\_,\_,\_)]\}),$

$((\{\mathit{crane\_unloaded}, \mathit{crane\_at}(\_)\}, \{\mathit{crane\_unloaded}, \mathit{crane\_at}(\_)\}), \{$

$[[\mathit{move\_crane}(\_,\_)]^*]\})\}$

$M_{crate} = \{$

$((\{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_)]\}),$

$((\{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*]\}),$

$((\{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{lift\_up}(\_,\_)]\}),$

$((\{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{stacked}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{truck\_stack}(\_,\_,\_,\_)],$

$[\mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{lift\_up}(\_,\_),$

$\mathit{crane\_stack}(\_,\_,\_,\_)]\}),$

$((\{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[[\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_)]\}),$

$((\{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}\}, \{$

$[[\mathit{drive\_load}(\_,\_,\_)]^*]\}),$

$((\{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[[\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{lift\_up}(\_,\_)]\}),$

$((\{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{stacked}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[[\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{truck\_stack}(\_,\_,\_,\_)],$

$[[\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_), \mathit{lift\_up}(\_,\_), \mathit{crane\_stack}(\_,\_,\_,\_)]\}),$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{on\_floor}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{lift\_down}(\_,\_,\_,\_), \mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_)]\}),$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{truck\_loaded}(\_,\_), \mathit{at}(\_,\_)\}), \{$

$[\mathit{lift\_down}(\_,\_,\_,\_), \mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*]\}),$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{crane\_loaded}(\_,\_),$

$\mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)]), \{$

$[\mathit{lift\_down}(\_,\_,\_,\_), \mathit{load}(\_,\_,\_), [\mathit{drive\_load}(\_,\_,\_)]^*, \mathit{unload}(\_,\_,\_),$

$[\mathit{move\_crane}(\_,\_)]^*, \mathit{lift\_up}(\_,\_)]\}),$

$((\{\mathit{crane\_loaded}(\_,\_), \mathit{above\_floor}(\_,\_), \mathit{at}(\_,\_)\}, \{\mathit{stacked}(\_,\_),$

$above\_floor(\_,\_), at(\_,\_)\}), \{$
$[lift\_down(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*, unload(\_,\_,\_),$
$truck\_stack(\_,\_,\_,\_)],$
$[crane\_stack(\_,\_,\_,\_)]\}),$
$((\{stacked(\_,\_), above\_floor(\_,\_), at(\_,\_)\}, \{on\_floor(\_,\_), at(\_,\_)\}), \{$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*,$
$unload(\_,\_,\_)]\}),$
$((\{stacked(\_,\_), above\_floor(\_,\_), at(\_,\_)\}, \{truck\_loaded(\_,\_), at(\_,\_)\}), \{$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*]\}),$
$((\{stacked(\_,\_), above\_floor(\_,\_), at(\_,\_)\}, \{crane\_loaded(\_,\_),$
$above\_floor(\_,\_), at(\_,\_)\}), \{$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*,$
$unload(\_,\_,\_), lift\_up(\_,\_)],$
$[crane\_unstack(\_,\_,\_,\_), lift\_down(\_,\_,\_,\_), load(\_,\_,\_),$
$[drive\_load(\_,\_,\_)]^*, unload(\_,\_,\_), lift\_up(\_,\_)],$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*,$
$unload(\_,\_,\_), lift\_up(\_,\_)]\}),$
$((\{stacked(\_,\_), above\_floor(\_,\_), at(\_,\_)\}, \{stacked(\_,\_),$
$above\_floor(\_,\_), at(\_,\_)]\}, \{$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*,$
$unload(\_,\_,\_), truck\_stack(\_,\_,\_,\_)],$
$[truck\_unstack(\_,\_,\_,\_), load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*,$
$unload(\_,\_,\_), lift\_up(\_,\_), crane\_stack(\_,\_,\_,\_)]\})\}$

$M_{driver} = \{$
$((\{driver\_on\_floor(\_), driver\_at(\_)\}, \{driver\_on\_floor(\_), driver\_at(\_)\}), \{$
$[[walk(\_,\_)]^*, get\_in(\_,\_), [drive(\_,\_)]^*, get\_out(\_,\_)],$
$[[walk(\_,\_)]^*, get\_in(\_,\_), [drive\_load(\_,\_,\_)]^*, get\_out(\_,\_)]\}),$
$((\{driver\_on\_floor(\_), driver\_at(\_)\}, \{in\_truck, driver\_at(\_)\}), \{$
$[[walk(\_,\_)]^*, get\_in(\_,\_), [drive(\_,\_)]^*],$
$[[walk(\_,\_)]^*, get\_in(\_,\_), [drive\_load(\_,\_,\_)]^*]\}),$
$((\{in\_truck, driver\_at(\_)\}, \{driver\_on\_floor(\_), driver\_at(\_)\}), \{$
$[[drive(\_,\_)]^*, get\_out(\_,\_), [walk(\_,\_)]^*],$
$[[drive\_load(\_,\_,\_)]^*, get\_out(\_,\_), [walk(\_,\_)]^*]\})\}$

$M_{truck} = \{$
$((\{in\_truck, driver\_at(\_)\}, \{in\_truck, driver\_at(\_)\}), \{$
$[[drive(\_,\_)]^*, get\_out(\_,\_), [walk(\_,\_)]^*, get\_in(\_,\_)],$
$[[drive\_load(\_,\_,\_)]^*, get\_out(\_,\_), [walk(\_,\_)]^*, get\_in(\_,\_)]\}),$
$((\{truck\_loaded(\_,\_), truck\_at(\_)\}, \{truck\_loaded(\_,\_), truck\_at(\_)\}), \{$
$[[drive\_load(\_,\_,\_)]^*, unload(\_,\_,\_), [drive(\_,\_)]^*, load(\_,\_,\_)]\}),$
$((\{truck\_loaded(\_,\_), truck\_at(\_)\}, \{truck\_unloaded, truck\_at(\_)\}), \{$
$[[drive\_load(\_,\_,\_)]^*, unload(\_,\_,\_), [drive(\_,\_)]^*]\}),$
$((\{truck\_unloaded, truck\_at(\_)], \{truck\_loaded(\_,\_), truck\_at(\_)\}), \{$

$[[drive(\_,\_)]^*, load(\_,\_,\_), [drive\_load(\_,\_,\_)]^*]\}),$
$((\{truck\_unloaded, truck\_at(\_)], \{truck\_unloaded, truck\_at(\_)\}), \{$
$[[drive(\_,\_)]^*]\}))\}$

## C.7 $\mathcal{EM}$: PROPOSE Enumerated Macros

There is only one static sort that is changed by iterative macros in this modelled planning domain, which is:

$\{space\}$

Below we list part of the set of enumerated sequences for this static sort:

$EM_{space} = \{$
$((s1, s2), \{[s1, s2]\}),$
$((s1, s3), \{[s1, s2, s3]\}),$
$((s1, s4), \{[s1, s2, s3, s4]\}),$
$((s1, s6), \{[s1, s2, s6]\}),$
$((s1, s7), \{[s1, s2, s3, s7], [s1, s2, s6, s7]\}),$
$((s1, s8), \{[s1, s2, s3, s4, s8], [s1, s2, s3, s7, s8], [s1, s2, s6, s7, s8]\}),$
$((s1, s9), \{[s1, s2, s6, s10, s9]\}),$
$((s1, s10), \{[s1, s2, s6, s10]\}),$
$((s1, s12), \{[s1, s2, s3, s4, s8, s12], [s1, s2, s3, s7, s8, s12\},$
$[s1, s2, s6, s7, s8, s12]\}),$
$((s1, s13), \{[s1, s2, s6, s10, s14, s13], [s1, s2, s6, s10, s9, s13]\}),$
$((s1, s14), \{[s1, s2, s6, s10, s14]\}),$
$((s1, s16), \{[s1, s2, s3, s4, s8, s12, s16], [s1, s2, s3, s7, s8, s12, s16\},$
$[s1, s2, s6, s7, s8, s12, s16]\})\},$
$((s2, s1), \{[s2, s1]\}),$
$((s2, s3), \{[s2, s3]\}),$
$((s2, s4), \{[s2, s3, s4]\}),$
$((s2, s6), \{[s2, s6]\}),$
$((s2, s7), \{[s2, s3, s7\}, \{s2, s6, s7]\}),$
$((s2, s8), \{[s2, s3, s4, s8], [s2, s3, s7, s8], [s2, s6, s7, s8]\}),$
$((s2, s9), \{[s2, s6, s10, s9]\}),$
$((s2, s10), \{[s3, s2, s6, s10], [s3, s7, s6, s10]\}),$
$((s2, s12), \{[s3, s4, s8, s12], [s3, s7, s8, s12]\}),$
$((s2, s13), \{[s3, s2, s6, s10, s14, s13], [s3, s2, s6, s10, s9, s13\},$
$[s3, s7, s6, s10, s14, s13\}, \{s3, s7, s6, s10, s9, s13]\}),$
$((s2, s14), \{[s3, s2, s6, s10, s14], [s3, s7, s6, s10, s14]\}),$
$((s2, s16), \{[s3, s4, s8, s12, s16], [s3, s7, s8, s12, s16]\})),$
$((s3, s1), \{[s3, s2, s1]\}),$
$((s3, s2), \{[s3, s2]\}),$
$((s3, s4), \{[s3, s4]\}),$

$((s3, s6), \{[s3, s2, s6], [s3, s7, s6]\}),$

$((s3, s7), \{[s3, s7]\}),$

$((s3, s8), \{[s3, s4, s8], [s3, s7, s8]\}),$

$((s3, s9), \{[s3, s2, s6, s10, s9], [s3, s7, s6, s10, s9]\})),$

$((s3, s10), \{[s3, s2, s6, s10], [s3, s7, s6, s10]\}),$

$((s3, s12), \{[s3, s4, s8, s12], [s3, s7, s8, s12]\}),$

$((s3, s13), \{[s3, s2, s6, s10, s14, s13], [s3, s2, s6, s10, s9, s13\},$
$\quad [s3, s7, s6, s10, s14, s13], [s3, s7, s6, s10, s9, s13]\}),$

$((s3, s14), \{[s3, s2, s6, s10, s14], [s3, s7, s6, s10, s14]\}),$

$((s3, s16), \{[s3, s4, s8, s12, s16], [s3, s7, s8, s12, s16]\})),$

$\ldots,$

$\ldots,$

$((s16, s1), \{[s16, s12, s8, s4, s3, s2, s1], [s16, s12, s8, s7, s3, s2, s1\},$
$\quad [s16, s12, s8, s7, s6, s2, s1]\}),$

$((s16, s2), \{[s16, s12, s8, s4, s3, s2], [s16, s12, s8, s7, s3, s2\},$
$\quad [s16, s12, s8, s7, s6, s2]\}),$

$((s16, s3), \{[s16, s12, s8, s4, s3], [s16, s12, s8, s7, s3]\}),$

$((s16, s4), \{[s16, s12, s8, s4]\}),$

$((s16, s6), \{[s16, s12, s8, s7, s6]\}),$

$((s16, s7), \{[s16, s12, s8, s7]\}),$

$((s16, s8), \{[s16, s12, s8]\}),$

$((s16, s9), \{[s16, s12, s8, s7, s6, s10, s9]\}),$

$((s16, s10), \{[s16, s12, s8, s7, s6, s10]\}),$

$((s16, s12), \{[s16, s12]\}),$

$((s16, s13), \{[s16, s12, s8, s7, s6, s10, s14, s13\},$
$\quad \{s16, s12, s8, s7, s6, s10, s9, s13]\}\}),$

$((s16, s14), \{[s16, s12, s8, s7, s6, s10, s14]\})\}$

## C.8 Empirical Results

In this section we list the results of the empirical tests that were summarised in chapter 7. The section is divided into three subsections which correspond to the performance metrics that were used in the tests: cpu time, the number of partial plans specialised and the number of operators. Each section is subdivided into two, the first part presents the results for a randomly generated problem sample that had been controlled for correctness by removing all tasks that the basic non-compiled planning function *MEA* could not solve. The second section is the results for an uncontrolled problem sample which had been randomly generated.

In the tables the *columns* give the results for each of the 9 different planning applications which were tested (for further explanation of these different applications see section 7.1). Down the left hand side of the tables the *rows* are numbered with the task number. The tasks that were used are listed on page 218.

### C.8.1 Cumulative CPU

#### C.8.1.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 67 | | 39 | | 66 | | 97 | | 28 | | 25 | | 15 | | 94 | | 43 | |
| 2. | 20 | | 21 | | 184 | | 8 | | 26 | | 234 | | 28 | | 193 | | 13 | |
| 3. | 36 | | 68 | | 51 | | 25 | | 49 | | 78 | | 91 | | 49 | | 36 | |
| 4. | 1 | | 2 | | 2 | | 1 | | 2 | | 2 | | 3 | | 2 | | 2 | |
| 5. | 7 | | 13 | | 17 | | 10 | | 11 | | 20 | | 18 | | 27 | | 16 | |
| 6. | 333 | | 492 | | 137 | | 35 | | 217 | | 152 | | 307 | | 1144 | | 40 | |
| 7. | 6 | | 11 | | 5 | | 2 | | 9 | | 16 | | 14 | | 3 | | 3 | |
| 8. | 7 | | 8 | | 5 | | 4 | | 10 | | 5 | | 11 | | 5 | | 6 | |
| 9. | 11 | | 14 | | 3 | | 2 | | 16 | | 4 | | 16 | | 2 | | 3 | |
| 10. | 437 | | 1175 | a | 297 | | 636 | | 397 | | 1269 | | 1340 | | 317 | f | 465 | |
| 11. | 1241 | | 1656 | a | 485 | | 24 | | 1481 | | 870 | | 1944 | a | 42 | | 35 | |
| 12. | 19 | | 64 | | 150 | | 26 | f | 26 | | 44 | | 79 | | 10 | | 40 | f |
| 13. | 3 | | 4 | | 7 | | 2 | | 3 | | 8 | | 4 | | 2 | | 2 | |
| 14. | 44 | | 46 | | 27 | f | 23 | | 33 | | 27 | f | 34 | | 18 | | 27 | |
| 15. | 392 | | 748 | | 329 | | 11 | | 113 | | 547 | | 185 | | 186 | | 16 | |
| 16. | 9 | | 12 | | 9 | | 14 | | 13 | | 12 | | 17 | | 7 | | 21 | |
| 17. | 20 | | 19 | | 3 | | 3 | | 28 | | 19 | | 26 | | 223 | | 4 | |
| 18. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 19. | 55 | | 77 | | 54 | | 24 | | 29 | | 64 | | 36 | | 25 | | 33 | |
| 20. | 1 | | 2 | | 1 | | 1 | | 1 | | 2 | | 2 | | 1 | | 1 | |
| 21. | 20 | | 134 | | 15 | | 49 | | 49 | | 32 | | 118 | | 32 | | 92 | |
| 22. | 74 | | 68 | | 6 | | 3 | | 91 | | 2 | | 82 | | 3 | | 5 | |
| 23. | 34 | | 125 | | 23 | | 24 | | 52 | | 79 | | 282 | | 26 | | 34 | |
| 24. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 25. | 34 | | 34 | | 13 | | 3 | | 46 | | 5 | | 44 | | 3 | | 4 | |
| 26. | 0 | | 1 | | 0 | | 0 | | 6 | | 1 | | 6 | | 0 | | 5 | |
| 27. | 76 | | 228 | | 101 | | 13 | | 101 | | 113 | | 280 | | 13 | | 18 | |
| 28. | 10 | | 12 | | 26 | | 11 | | 14 | | 35 | f | 15 | | 55 | | 14 | |
| 29. | 70 | | 69 | | 31 | f | 17 | | 34 | | 39 | | 32 | | 26 | | 16 | |
| 30. | 2 | | 3 | | 3 | | 3 | | 2 | | 4 | | 3 | | 2 | | 3 | |
| 31. | 416 | | 1419 | a | 173 | | 179 | | 208 | | 544 | | 1554 | a | 126 | | 49 | |
| 32. | 33 | | 35 | | 16 | | 3 | | 24 | | 10 | | 24 | | 3 | | 4 | |
| 33. | 68 | | 50 | | 4 | | 4 | | 86 | | 21 | | 62 | | 4 | | 5 | |
| 34. | 34 | | 63 | | 40 | | 52 | | 49 | | 73 | | 81 | | 57 | | 73 | |
| 35. | 40 | | 43 | | 18 | | 13 | | 49 | | 44 | | 51 | | 7 | | 17 | |
| 36. | 6 | | 12 | | 6 | | 11 | | 8 | | 10 | | 12 | | 4 | | 18 | |
| 37. | 1 | | 2 | | 2 | | 1 | | 1 | | 5 | | 2 | | 2 | | 2 | |
| 38. | 42 | | 30 | | 5 | | 4 | | 29 | | 14 | | 22 | | 5 | | 6 | |
| 39. | 15 | | 25 | | 6 | | 14 | | 22 | | 6 | | 34 | | 4 | | 21 | |
| 40. | 35 | | 241 | | 9 | | 7 | | 46 | | 15 | | 258 | | 6 | | 9 | |
| 41. | 8 | | 10 | | 11 | f | 26 | | 12 | | 15 | f | 13 | | 114 | f | 36 | |
| 42. | 24 | | 39 | | 29 | f | 4 | | 35 | | 4 | | 50 | | 4 | | 6 | |
| 43. | 247 | | 664 | | 55 | | 16 | | 300 | | 94 | | 762 | | 148 | | 23 | |
| 44. | 51 | | 176 | | 28 | | 104 | f | 50 | | 57 | | 199 | | 91 | f | 147 | f |
| 45. | 52 | | 295 | | 13 | | 26 | | 31 | | 19 | | 130 | | 50 | | 31 | |
| 46. | 13 | | 26 | | 9 | | 15 | | 19 | | 13 | | 34 | | 6 | | 21 | |
| 47. | 90 | | 299 | | 37 | | 39 | | 72 | | 25 | | 196 | | 17 | | 51 | |
| 48. | 4 | | 5 | | 9 | | 3 | | 11 | | 9 | | 12 | | 4 | | 6 | |
| 49. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 2 | | 1 | | 1 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50. | 5 | | 5 | | 3 | | 20 | | 6 | | 4 | | 6 | | 18 | f | 21 | |
| 51. | 13 | | 14 | | 2 | | 3 | | 18 | | 4 | | 18 | | 2 | | 4 | |
| 52. | 76 | | 250 | | 32 | | 17 | | 63 | | 58 | | 166 | | 80 | f | 21 | |
| 53. | 22 | | 112 | | 19 | | 8 | | 30 | | 39 | | 127 | | 8 | | 11 | |
| 54. | 21 | | 23 | | 14 | | 10 | | 29 | | 23 | | 30 | | 10 | | 13 | |
| 55. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 56. | 42 | | 85 | | 5 | | 31 | | 55 | | 6 | | 97 | | 4 | | 42 | |
| 57. | 240 | | 1248 | | 86 | | 1254 | | 160 | | 607 | | 658 | | 60 | | 696 | |
| 58. | 92 | | 95 | | 10 | | 8 | | 114 | | 9 | | 111 | | 7 | | 11 | |
| 59. | 61 | | 217 | | 16 | | 30 | | 79 | | 24 | | 241 | | 531 | | 41 | |
| 60. | 5 | | 8 | | 13 | | 6 | | 8 | | 10 | | 11 | | 6 | | 8 | |
| 61. | 22 | | 22 | | 31 | f | 42 | | 28 | f | 129 | f | 27 | f | 19 | f | 6 | |
| 62. | 27 | | 29 | | 11 | f | 21 | | 39 | | 14 | f | 45 | | 2 | f | 33 | |
| 63. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 2 | | 0 | | 1 | |
| 64. | 9 | | 8 | | 8 | | 4 | | 14 | | 21 | | 12 | | 3 | | 6 | |
| 65. | 23 | | 33 | | 16 | | 231 | f | 142 | f | 58 | | 267 | f | 108 | f | 271 | f |
| 66. | 4 | | 7 | | 24 | | 10 | | 6 | | 32 | | 10 | | 10 | | 13 | |
| 67. | 372 | | 631 | | 58 | | 94 | | 421 | | 113 | | 676 | | 35 | | 109 | |
| 68. | 18 | | 32 | | 4 | | 7 | | 30 | | 16 | | 23 | | 6 | | 7 | |
| 69. | 288 | | 209 | | 62 | | 32 | | 341 | | 146 | | 252 | | 8 | f | 42 | |
| 70. | 131 | | 352 | | 63 | | 18 | | 83 | | 69 | | 181 | | 21 | | 23 | |
| 71. | 24 | | 169 | | 7 | | 12 | | 61 | | 21 | | 258 | | 39 | f | 17 | |
| 72. | 6 | | 7 | | 7 | | 3 | | 9 | | 9 | | 10 | | 3 | | 4 | |
| 73. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 74. | 12 | | 24 | | 9 | | 11 | | 50 | | 20 | | 70 | | 4 | | 46 | |
| 75. | 19 | | 41 | | 6 | | 15 | | 20 | | 44 | | 35 | | 6 | | 18 | |
| 76. | 6 | | 6 | | 2 | | 3 | | 8 | | 3 | | 8 | | 2 | | 3 | |
| 77. | 12 | | 19 | | 11 | | 17 | | 18 | | 13 | | 27 | | 15 | | 23 | |
| 78. | 13 | | 14 | | 7 | | 4 | | 18 | | 6 | | 19 | | 5 | | 6 | |
| 79. | 301 | | 886 | | 77 | | 19 | | 168 | | 225 | | 340 | | 13 | | 14 | |
| 80. | 353 | | 760 | | 218 | | 75 | | 376 | | 336 | f | 805 | | 44 | | 92 | |
| 81. | 5 | | 6 | | 7 | | 5 | | 7 | | 9 | | 9 | | 5 | | 7 | |
| 82. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 83. | 68 | | 72 | | 49 | f | 14 | | 82 | | 53 | f | 87 | | 38 | f | 18 | |
| 84. | 25 | | 86 | | 31 | | 59 | | 31 | | 44 | f | 102 | | 23 | f | 72 | |
| 85. | 22 | | 40 | | 79 | | 30 | | 18 | | 309 | | 28 | | 13 | f | 32 | |
| 86. | 419 | | 581 | | 150 | | 31 | | 494 | | 260 | | 680 | | 95 | | 43 | |
| 87. | 3 | | 4 | | 3 | | 3 | | 5 | | 4 | | 5 | | 3 | | 5 | |
| 88. | 2 | | 3 | | 3 | | 3 | | 4 | | 3 | | 5 | | 2 | | 4 | |
| 89. | 363 | | 1440 | a | 61 | | 269 | | 160 | | 152 | | 2183 | | 244 | | 173 | |
| 90. | 27 | | 15 | | 21 | | 4 | | 35 | | 19 | | 20 | | 4 | | 6 | |
| 91. | 16 | | 24 | | 23 | | 15 | | 23 | | 18 | | 33 | | 15 | | 22 | |
| 92. | 90 | | 659 | | 18 | | 158 | | 103 | | 55 | | 703 | | 43 | | 187 | |
| 93. | 352 | | 698 | | 136 | | 392 | | 178 | | 303 | | 405 | | 33 | | 231 | |
| 94. | 3 | | 3 | | 3 | | 3 | | 4 | | 3 | | 5 | | 3 | | 4 | |
| 95. | 42 | | 95 | | 8 | | 116 | | 53 | | 8 | | 112 | | 11 | | 125 | |
| 96. | 4 | | 7 | | 3 | | 5 | | 5 | | 4 | | 10 | | 5 | | 7 | |
| 97. | 27 | | 29 | | 18 | | 25 | | 19 | | 19 | | 21 | | 30 | | 33 | |
| 98. | 4 | | 6 | | 3 | | 11 | | 6 | | 4 | | 8 | | 2 | | 15 | |
| 99. | 3 | | 5 | | 3 | | 4 | | 5 | | 5 | | 6 | | 9 | | 5 | |
| 100. | 317 | | 397 | | 35 | | 12 | | 228 | | 229 | f | 273 | | 11 | | 18 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 8218 | | 18060 | | 3996 | | 4757 | | 7798 | | 8251 | | 17760 | | 4821 | | 4135 | |

### C.8.1.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 70 | | 27 | | 65 | | 98 | | 28 | | 25 | | 16 | | 94 | | 44 | |
| 2. | 126 | f | 415 | f | 100 | f | 110 | f | 165 | f | 146 | f | 459 | f | 105 | f | 148 | f |
| 3. | 21 | | 23 | | 185 | | 8 | | 26 | | 237 | | 28 | | 193 | | 14 | |
| 4. | 150 | f | 38 | | 320 | f | 72 | f | 198 | f | 838 | a | 83 | | 73 | f | 115 | f |
| 5. | 175 | f | 403 | f | 120 | f | 99 | f | 221 | f | 171 | f | 494 | f | 99 | f | 155 | f |
| 6. | 39 | | 74 | | 55 | | 25 | | 49 | | 79 | | 91 | | 49 | | 41 | |
| 7. | 1 | | 2 | | 2 | | 1 | | 2 | | 2 | | 3 | | 2 | | 2 | |
| 8. | 8 | | 14 | | 18 | | 10 | | 11 | | 21 | | 18 | | 27 | | 18 | |
| 9. | 344 | | 510 | | 144 | | 35 | | 217 | | 155 | | 311 | | 1145 | | 42 | |
| 10. | 9 | | 17 | | 9 | | 3 | | 8 | | 34 | | 14 | | 3 | | 4 | |
| 11. | 11 | f | 15 | f | 15 | f | 18 | f | 19 | f | 31 | f | 23 | f | 17 | f | 26 | f |
| 12. | 14 | f | 16 | f | 34 | f | 7 | f | 20 | f | 26 | f | 21 | f | 8 | f | 11 | f |
| 13. | 13 | | 15 | | 10 | | 4 | | 10 | | 11 | | 11 | | 5 | | 5 | f |
| 14. | 12 | | 14 | | 8 | | 2 | | 16 | | 9 | | 16 | | 2 | | 4 | |
| 15. | 448 | | 1210 | a | 263 | f | 636 | | 397 | | 1165 | a | 1350 | | 354 | f | 488 | |
| 16. | 1282 | | 1746 | a | 76 | f | 24 | | 1478 | | 1569 | a | 1935 | a | 43 | | 35 | |
| 17. | 20 | | 70 | | 192 | | 26 | f | 26 | | 36 | | 78 | | 23 | f | 40 | f |
| 18. | 3 | | 4 | | 4 | | 2 | | 3 | | 6 | | 4 | | 2 | | 2 | |
| 19. | 968 | a | 1097 | a | 841 | a | 43 | f | 1199 | a | 800 | a | 1215 | a | 62 | f | 61 | f |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20. | 862 | a | 915 | a | 836 | a | 184 | | 933 | a | 974 | a | 951 | a | 82 | | 227 | |
| 21. | 46 | | 48 | | 12 | | 23 | | 33 | | 13 | | 34 | | 17 | | 29 | |
| 22. | 1512 | a | 1387 | a | 207 | f | 842 | f | 1578 | a | 1623 | f | 1365 | a | 385 | f | 914 | f |
| 23. | 11 | f | 13 | f | 11 | f | 20 | f | 17 | f | 12 | f | 18 | f | 17 | f | 29 | f |
| 24. | 400 | | 784 | | 241 | | 11 | | 112 | | 476 | | 184 | | 173 | | 17 | |
| 25. | 9 | | 12 | | 6 | | 14 | | 13 | | 9 | | 17 | | 7 | | 21 | |
| 26. | 21 | | 19 | | 20 | | 3 | | 28 | | 4 | | 25 | | 2 | | 4 | |
| 27. | 0 | | 1 | | 0 | | 0 | | 1 | | 1 | | 1 | | 0 | | 1 | |
| 28. | 56 | | 78 | | 69 | | 25 | | 29 | | 55 | | 36 | | 23 | | 33 | |
| 29. | 1 | | 2 | | 1 | | 1 | | 1 | | 2 | | 2 | | 1 | | 1 | |
| 30. | 20 | | 139 | | 15 | | 51 | | 49 | | 32 | | 117 | | 34 | | 91 | |
| 31. | 75 | | 77 | | 35 | | 3 | | 92 | | 9 | | 81 | | 3 | | 5 | |
| 32. | 35 | | 129 | | 16 | | 24 | | 53 | | 73 | | 281 | | 13 | | 34 | |
| 33. | 1203 | a | 1326 | a | 575 | f | 188 | f | 1270 | a | 1543 | f | 1355 | a | 101 | f | 160 | f |
| 34. | 37 | f | 53 | f | 31 | f | 33 | f | 48 | f | 24 | f | 70 | f | 18 | f | 40 | f |
| 35. | 297 | f | 335 | | 214 | f | 24 | | 138 | | 821 | a | 160 | | 40 | | 32 | |
| 36. | 1399 | a | 1451 | a | 1353 | a | 1414 | a | 1590 | a | 1531 | a | 1608 | a | 1307 | a | 1698 | a |
| 37. | 0 | | 0 | | 0 | a | 0 | | 0 | | 0 | a | 0 | a | 0 | a | 0 | a |
| 38. | 39 | | 35 | | 37 | | 3 | | 47 | | 15 | | 44 | | 3 | | 4 | |
| 39. | 97 | f | 103 | f | 62 | f | 32 | f | 82 | f | 38 | f | 63 | f | 32 | f | 38 | f |
| 40. | 752 | f | 862 | f | 560 | f | 1507 | a | 919 | f | 804 | f | 995 | f | 618 | f | 1821 | a |
| 41. | 0 | | 1 | | 0 | | 0 | | 6 | | 1 | | 6 | | 0 | | 5 | |
| 42. | 16 | f | 16 | f | 13 | f | 22 | f | 23 | f | 14 | f | 20 | f | 14 | f | 32 | f |
| 43. | 77 | | 239 | | 86 | | 14 | | 101 | | 51 | | 277 | | 12 | | 18 | |
| 44. | 51 | f | 91 | f | 37 | f | 74 | f | 58 | f | 72 | f | 100 | f | 23 | f | 82 | f |
| 45. | 31 | f | 127 | f | 17 | f | 32 | f | 38 | f | 23 | f | 140 | f | 14 | f | 39 | f |
| 46. | 62 | f | 189 | f | 55 | f | 70 | f | 88 | f | 155 | f | 235 | f | 58 | f | 94 | f |
| 47. | 1268 | a | 1320 | a | 1302 | a | 1187 | a | 1303 | a | 1344 | a | 1300 | a | 1561 | a | 1248 | a |
| 48. | 13 | f | 19 | f | 15 | f | 12 | f | 19 | f | 20 | f | 26 | f | 12 | f | 18 | f |
| 49. | 46 | f | 81 | f | 19 | f | 49 | f | 45 | f | 33 | f | 79 | f | 17 | f | 48 | f |
| 50. | 388 | f | 247 | f | 373 | f | 28 | f | 412 | f | 411 | f | 266 | f | 29 | f | 34 | f |
| 51. | 113 | f | 716 | f | 50 | | 51 | f | 123 | | 60 | | 468 | | 42 | f | 59 | |
| 52. | 11 | | 13 | | 5 | | 12 | | 14 | | 10 | | 15 | | 37 | | 14 | |
| 53. | 273 | f | 506 | f | 130 | f | 680 | f | 194 | f | 102 | f | 306 | f | 197 | f | 311 | f |
| 54. | 71 | | 73 | | 61 | | 18 | | 35 | | 26 | | 33 | | 16 | | 16 | |
| 55. | 2 | | 3 | | 3 | | 3 | | 2 | | 4 | | 3 | | 2 | | 3 | |
| 56. | 4 | f | 4 | f | 5 | f | 3 | f | 6 | f | 5 | f | 7 | f | 3 | f | 5 | f |
| 57. | 429 | | 1504 | a | 332 | | 189 | | 212 | | 476 | | 1551 | a | 99 | | 50 | |
| 58. | 1063 | a | 1183 | a | 895 | f | 20 | | 1369 | a | 1527 | a | 1417 | a | 20 | | 24 | |
| 59. | 37 | | 38 | | 15 | | 4 | | 25 | | 7 | | 25 | | 3 | | 4 | |
| 60. | 72 | | 52 | | 40 | | 4 | | 89 | | 4 | | 63 | | 4 | | 5 | |
| 61. | 37 | | 69 | | 35 | | 55 | | 53 | | 74 | | 81 | | 56 | | 73 | |
| 62. | 42 | | 45 | | 10 | | 14 | | 53 | | 12 | | 51 | | 7 | | 18 | |
| 63. | 7 | | 13 | | 3 | | 12 | | 8 | | 5 | | 12 | | 4 | | 18 | |
| 64. | 1 | | 2 | | 2 | | 2 | | 2 | | 5 | | 2 | | 2 | | 2 | |
| 65. | 45 | | 37 | | 431 | f | 5 | | 31 | | 8 | | 22 | | 4 | | 6 | |
| 66. | 143 | f | 663 | f | 522 | a | 131 | f | 184 | f | 991 | a | 649 | f | 413 | f | 142 | f |
| 67. | 16 | | 26 | | 5 | | 15 | | 23 | | 6 | | 34 | | 4 | | 21 | |
| 68. | 247 | f | 311 | f | 219 | f | 1068 | a | 181 | f | 259 | f | 226 | f | 492 | f | 473 | f |
| 69. | 86 | f | 133 | f | 141 | f | 70 | f | 114 | f | 93 | f | 180 | f | 90 | f | 93 | f |
| 70. | 36 | | 247 | | 13 | | 7 | | 45 | | 19 | | 259 | | 6 | | 9 | |
| 71. | 1445 | a | 1313 | a | 345 | f | 279 | f | 1493 | a | 378 | f | 1298 | a | 144 | f | 313 | f |
| 72. | 344 | f | 727 | a | 601 | f | 1165 | a | 453 | f | 1101 | a | 823 | a | 164 | f | 353 | f |
| 73. | 9 | | 10 | | 69 | f | 28 | | 12 | | 13 | f | 13 | | 60 | f | 36 | |
| 74. | 25 | | 37 | | 45 | | 4 | | 35 | | 37 | f | 50 | | 24 | f | 6 | |
| 75. | 1400 | a | 1436 | a | 432 | f | 1395 | a | 1709 | a | 1738 | a | 1707 | a | 182 | f | 1686 | a |
| 76. | 187 | f | 899 | f | 233 | f | 47 | | 107 | f | 423 | f | 452 | f | 146 | f | 62 | |
| 77. | 68 | f | 81 | f | 31 | f | 74 | f | 52 | f | 84 | f | 61 | f | 26 | f | 95 | f |
| 78. | 402 | f | 1327 | a | 139 | f | 244 | f | 321 | f | 402 | f | 1325 | a | 39 | f | 408 | f |
| 79. | 251 | | 700 | | 220 | | 16 | | 301 | | 181 | | 762 | | 63 | | 24 | |
| 80. | 54 | | 183 | | 113 | | 109 | f | 49 | | 96 | | 199 | | 91 | f | 147 | f |
| 81. | 304 | f | 441 | f | 112 | f | 31 | f | 844 | f | 283 | f | 985 | f | 104 | f | 42 | f |
| 82. | 20 | f | 28 | | 25 | | 16 | | 27 | f | 25 | | 29 | | 16 | | 23 | |
| 83. | 703 | f | 513 | f | 441 | f | 529 | f | 437 | f | 697 | a | 385 | f | 50 | f | 376 | f |
| 84. | 53 | | 310 | | 23 | | 30 | | 32 | | 15 | f | 129 | | 30 | | 32 | |
| 85. | 13 | | 27 | | 6 | | 17 | | 19 | | 11 | | 34 | | 6 | | 21 | |
| 86. | 96 | | 308 | | 23 | f | 43 | | 72 | | 20 | | 192 | | 19 | | 51 | |
| 87. | 4 | | 5 | | 4 | | 4 | | 11 | | 8 | | 12 | | 5 | | 6 | |
| 88. | 36 | f | 60 | f | 50 | f | 16 | f | 48 | f | 66 | f | 74 | f | 15 | f | 23 | f |
| 89. | 1005 | a | 1068 | a | 709 | a | 17 | f | 597 | f | 1073 | a | 1069 | a | 19 | f | 24 | f |
| 90. | 62 | f | 192 | f | 17 | f | 8 | f | 52 | f | 48 | f | 158 | f | 20 | f | 22 | f |
| 91. | 1 | | 2 | | 1 | | 1 | | 1 | | 2 | | 2 | | 1 | | 2 | |
| 92. | 5 | | 6 | | 5 | | 21 | | 6 | | 3 | | 6 | | 7 | | 21 | |
| 93. | 388 | f | 1132 | f | 542 | f | 1197 | f | 196 | f | 667 | f | 320 | f | 474 | f | 301 | f |
| 94. | 13 | | 15 | | 4 | | 3 | | 18 | | 5 | | 19 | | 2 | | 4 | |
| 95. | 11 | f | 12 | f | 17 | f | 4 | f | 31 | f | 10 | f | 28 | f | 15 | f | 38 | f |
| 96. | 78 | | 274 | | 135 | f | 18 | | 63 | | 252 | | 173 | | 16 | | 21 | |
| 97. | 23 | | 120 | | 16 | | 8 | | 30 | | 8 | | 136 | | 7 | | 11 | |
| 98. | 22 | | 25 | | 13 | | 10 | | 29 | | 18 | | 32 | | 53 | | 13 | |
| 99. | 133 | f | 113 | f | 457 | f | 21 | f | 163 | f | 16 | f | 142 | f | 12 | f | 28 | f |
| 100. | 0 | | 0 | | 0 | f | 0 | | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 22428 | | 32811 | | 16424 | | 14831 | | 23190 | | 26917 | | 32045 | | 10233 | | 13607 | |

## C.8.2 Partial Plans

### C.8.2.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 293 | | 160 | | 290 | | 425 | | 145 | | 155 | | 84 | | 417 | | 214 | |
| 2. | 150 | | 149 | | 640 | | 48 | | 128 | | 680 | | 136 | | 563 | | 58 | |
| 3. | 228 | | 305 | | 275 | | 137 | | 228 | | 353 | | 305 | | 190 | | 137 | |
| 4. | 11 | | 11 | | 9 | | 11 | | 11 | | 9 | | 11 | | 9 | | 11 | |
| 5. | 45 | | 73 | | 86 | | 59 | | 45 | | 99 | | 73 | | 141 | | 59 | |
| 6. | 797 | | 961 | | 410 | | 180 | | 526 | | 424 | | 662 | | 1616 | | 156 | |
| 7. | 57 | | 82 | | 39 | | 22 | | 57 | | 95 | | 82 | | 20 | | 22 | |
| 8. | 55 | | 55 | | 29 | | 30 | | 55 | | 29 | | 55 | | 26 | | 30 | |
| 9. | 96 | | 95 | | 20 | | 22 | | 96 | | 22 | | 95 | | 15 | | 22 | |
| 10. | 1081 | | 2002 | a | 859 | | 1277 | | 793 | | 1872 | | 1597 | | 877 | f | 839 | |
| 11. | 1575 | | 2004 | a | 884 | | 96 | | 1575 | | 1170 | | 2004 | a | 122 | | 96 | |
| 12. | 131 | | 306 | | 458 | | 149 | f | 131 | | 193 | | 306 | | 52 | | 159 | f |
| 13. | 27 | | 27 | | 54 | | 16 | | 17 | | 54 | | 17 | | 16 | | 13 | |
| 14. | 311 | | 311 | | 168 | f | 164 | | 195 | | 170 | f | 195 | | 117 | | 144 | |
| 15. | 1041 | | 1414 | | 770 | | 67 | | 441 | | 1075 | | 559 | | 526 | | 75 | |
| 16. | 55 | | 63 | | 48 | | 64 | | 55 | | 56 | | 63 | | 26 | | 64 | |
| 17. | 140 | | 124 | | 20 | | 23 | | 140 | | 54 | | 124 | | 608 | | 23 | |
| 18. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 19. | 353 | | 419 | | 278 | | 146 | | 144 | | 332 | | 172 | | 131 | | 139 | |
| 20. | 7 | | 7 | | 6 | | 7 | | 7 | | 6 | | 7 | | 6 | | 7 | |
| 21. | 117 | | 396 | | 38 | | 160 | | 193 | | 81 | | 331 | | 78 | | 212 | |
| 22. | 343 | | 309 | | 19 | | 26 | | 343 | | 6 | | 309 | | 16 | | 26 | |
| 23. | 225 | | 532 | | 104 | | 130 | | 245 | | 277 | | 753 | | 105 | | 133 | |
| 24. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 25. | 196 | | 188 | | 68 | | 24 | | 196 | | 24 | | 188 | | 17 | | 24 | |
| 26. | 0 | | 0 | | 0 | | 0 | | 36 | | 0 | | 32 | | 0 | | 35 | |
| 27. | 335 | | 674 | | 340 | | 77 | | 335 | | 323 | | 674 | | 63 | | 77 | |
| 28. | 93 | | 93 | | 82 | | 82 | | 93 | | 108 | f | 93 | | 199 | | 82 | |
| 29. | 381 | | 381 | | 96 | f | 104 | | 163 | | 173 | | 163 | | 112 | | 73 | |
| 30. | 20 | | 20 | | 21 | | 27 | | 13 | | 21 | | 13 | | 14 | | 20 | |
| 31. | 1148 | | 2002 | a | 484 | | 600 | | 620 | | 1043 | | 2000 | a | 438 | | 147 | |
| 32. | 195 | | 195 | | 93 | | 27 | | 120 | | 61 | | 120 | | 22 | | 24 | |
| 33. | 314 | | 251 | | 22 | | 29 | | 314 | | 110 | | 251 | | 19 | | 29 | |
| 34. | 203 | | 355 | | 198 | | 294 | | 203 | | 360 | | 355 | | 280 | | 294 | |
| 35. | 246 | | 246 | | 116 | | 91 | | 245 | | 198 | | 245 | | 49 | | 91 | |
| 36. | 58 | | 88 | | 37 | | 72 | | 51 | | 51 | | 67 | | 23 | | 81 | |
| 37. | 13 | | 13 | | 10 | | 13 | | 10 | | 34 | | 10 | | 13 | | 10 | |
| 38. | 247 | | 184 | | 24 | | 17 | | 149 | | 65 | | 111 | | 17 | | 14 | |
| 39. | 104 | | 141 | | 24 | | 75 | | 104 | | 28 | | 141 | | 11 | | 75 | |
| 40. | 201 | | 638 | | 45 | | 46 | | 201 | | 76 | | 638 | | 33 | | 46 | |
| 41. | 62 | | 60 | | 38 | f | 133 | | 62 | | 53 | f | 60 | | 376 | f | 133 | |
| 42. | 146 | | 179 | | 98 | f | 15 | | 139 | | 12 | | 172 | | 15 | | 15 | |
| 43. | 643 | | 1082 | | 182 | | 60 | | 643 | | 282 | | 1082 | | 487 | | 60 | |
| 44. | 331 | | 624 | | 135 | | 353 | f | 250 | | 261 | | 603 | | 265 | f | 349 | f |
| 45. | 268 | | 756 | | 64 | | 144 | | 146 | | 91 | | 416 | | 156 | | 128 | |
| 46. | 88 | | 151 | | 38 | | 99 | | 85 | | 59 | | 148 | | 16 | | 96 | |
| 47. | 380 | | 744 | | 150 | | 213 | | 290 | | 102 | | 540 | | 58 | | 213 | |
| 48. | 27 | | 24 | | 41 | | 21 | | 59 | | 42 | | 59 | | 16 | | 23 | |
| 49. | 4 | | 4 | | 4 | | 4 | | 4 | | 3 | | 4 | | 3 | | 4 | |
| 50. | 43 | | 39 | | 18 | | 124 | | 32 | | 15 | | 28 | | 59 | f | 106 | |
| 51. | 102 | | 100 | | 13 | | 24 | | 102 | | 23 | | 100 | | 12 | | 24 | |
| 52. | 344 | | 663 | | 116 | | 106 | | 245 | | 222 | | 464 | | 214 | f | 95 | |
| 53. | 160 | | 437 | | 108 | | 53 | | 160 | | 185 | | 437 | | 36 | | 53 | |
| 54. | 166 | | 166 | | 89 | | 77 | | 154 | | 141 | | 154 | | 52 | | 65 | |
| 55. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 56. | 236 | | 350 | | 23 | | 156 | | 236 | | 18 | | 350 | | 11 | | 156 | |
| 57. | 695 | | 1553 | | 307 | | 1641 | | 467 | | 978 | | 1058 | | 206 | | 1081 | |
| 58. | 379 | | 381 | | 39 | | 48 | | 379 | | 47 | | 381 | | 31 | | 48 | |
| 59. | 282 | | 599 | | 61 | | 155 | | 282 | | 99 | | 599 | | 915 | | 155 | |
| 60. | 28 | | 39 | | 55 | | 27 | | 28 | | 33 | | 39 | | 15 | | 27 | |
| 61. | 157 | | 156 | | 93 | f | 202 | | 122 | f | 381 | f | 125 | f | 53 | f | 35 | |
| 62. | 193 | | 196 | | 46 | f | 127 | | 208 | | 40 | f | 243 | | 8 | f | 148 | |
| 63. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 64. | 70 | | 55 | | 37 | | 21 | | 70 | | 91 | | 55 | | 10 | | 21 | |
| 65. | 150 | | 183 | | 64 | | 705 | f | 567 | f | 177 | | 833 | f | 419 | f | 701 | f |
| 66. | 37 | | 54 | | 92 | | 66 | | 37 | | 124 | | 54 | | 53 | | 66 | |
| 67. | 878 | | 1153 | | 221 | | 360 | | 878 | | 371 | | 1153 | | 134 | | 360 | |
| 68. | 125 | | 198 | | 32 | | 40 | | 153 | | 98 | | 121 | | 30 | | 42 | |
| 69. | 748 | | 599 | | 230 | | 167 | | 748 | | 451 | | 599 | | 31 | f | 167 | |
| 70. | 510 | | 839 | | 268 | | 75 | | 311 | | 274 | | 439 | | 59 | | 72 | |
| 71. | 164 | | 572 | | 26 | | 67 | | 272 | | 106 | | 661 | | 148 | f | 81 | |
| 72. | 51 | | 51 | | 33 | | 17 | | 51 | | 44 | | 51 | | 12 | | 17 | |
| 73. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 74. | 94 | | 142 | | 54 | | 59 | | 94 | | 103 | | 142 | | 20 | | 59 | |
| 75. | 142 | | 246 | | 43 | | 86 | | 115 | | 167 | | 171 | | 28 | | 79 | |
| 76. | 55 | | 44 | | 9 | | 18 | | 55 | | 8 | | 44 | | 11 | | 18 | |
| 77. | 92 | | 130 | | 55 | | 106 | | 92 | | 70 | | 130 | | 79 | | 106 | |
| 78. | 97 | | 95 | | 40 | | 26 | | 97 | | 26 | | 95 | | 15 | | 26 | |
| 79. | 1050 | | 1711 | | 324 | | 114 | | 637 | | 658 | | 925 | | 64 | | 68 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80. | 1185 | | 1735 | | 702 | | 332 | | 1113 | | 1049 | f | 1664 | | 159 | | 342 | |
| 81. | 25 | | 25 | | 19 | | 19 | | 25 | | 25 | | 25 | | 19 | | 19 | |
| 82. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 83. | 328 | | 328 | | 155 | f | 92 | | 328 | | 181 | f | 328 | | 85 | f | 92 | |
| 84. | 159 | | 350 | | 116 | | 250 | | 159 | | 170 | f | 350 | | 62 | f | 250 | |
| 85. | 160 | | 244 | | 347 | | 182 | | 87 | | 756 | | 122 | | 52 | f | 143 | |
| 86. | 865 | | 996 | | 397 | | 120 | | 865 | | 555 | | 996 | | 279 | | 120 | |
| 87. | 8 | | 8 | | 7 | | 8 | | 8 | | 4 | | 8 | | 7 | | 8 | |
| 88. | 19 | | 18 | | 18 | | 18 | | 19 | | 15 | | 18 | | 11 | | 18 | |
| 89. | 858 | | 2005 | a | 241 | | 709 | | 439 | | 402 | | 1924 | | 576 | | 400 | |
| 90. | 156 | | 87 | | 92 | | 26 | | 156 | | 85 | | 87 | | 15 | | 26 | |
| 91. | 87 | | 122 | | 102 | | 70 | | 87 | | 81 | | 122 | | 51 | | 70 | |
| 92. | 365 | | 1115 | | 80 | | 486 | | 365 | | 160 | | 1115 | | 164 | | 486 | |
| 93. | 839 | | 1209 | | 452 | | 985 | | 520 | | 719 | | 835 | | 129 | | 604 | |
| 94. | 4 | | 4 | | 3 | | 4 | | 4 | | 3 | | 4 | | 4 | | 4 | |
| 95. | 217 | | 373 | | 27 | | 396 | | 217 | | 25 | | 373 | | 37 | | 396 | |
| 96. | 38 | | 57 | | 18 | | 22 | | 38 | | 23 | | 57 | | 30 | | 22 | |
| 97. | 180 | | 186 | | 102 | | 119 | | 86 | | 97 | | 92 | | 114 | | 123 | |
| 98. | 34 | | 37 | | 17 | | 58 | | 34 | | 12 | | 37 | | 9 | | 58 | |
| 99. | 11 | | 11 | | 7 | | 11 | | 11 | | 9 | | 11 | | 22 | | 11 | |
| 100. | 878 | | 956 | | 182 | | 78 | | 699 | | 502 | f | 756 | | 57 | | 85 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 26308 | | 39518 | | 13497 | | 14534 | | 21956 | | 20618 | | 34503 | | 12989 | | 11938 | |

### C.8.2.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | | | | | | | | | | | | | | | | | | |
| 1. | 293 | | 160 | | 290 | | 425 | | 145 | | 155 | | 84 | | 417 | | 214 | |
| 2. | 383 | f | 761 | f | 387 | f | 414 | f | 383 | f | 479 | f | 761 | f | 380 | f | 414 | f |
| 3. | 150 | | 149 | | 640 | | 48 | | 128 | | 680 | | 136 | | 563 | | 58 | |
| 4. | 727 | f | 271 | | 1230 | f | 339 | f | 850 | f | 2000 | a | 417 | | 333 | f | 406 | f |
| 5. | 679 | f | 1010 | f | 491 | f | 369 | f | 679 | f | 644 | f | 1010 | f | 339 | f | 369 | f |
| 6. | 228 | | 305 | | 275 | | 137 | | 228 | | 353 | | 305 | | 190 | | 137 | |
| 7. | 11 | | 11 | | 9 | | 11 | | 11 | | 9 | | 11 | | 9 | | 11 | |
| 8. | 45 | | 73 | | 86 | | 59 | | 45 | | 99 | | 73 | | 141 | | 59 | |
| 9. | 797 | | 961 | | 410 | | 180 | | 526 | | 424 | | 662 | | 1616 | | 156 | |
| 10. | 84 | | 122 | | 68 | | 25 | | 57 | | 176 | | 82 | | 23 | | 22 | |
| 11. | 67 | f | 77 | f | 64 | f | 96 | f | 81 | f | 146 | f | 91 | f | 81 | f | 85 | f |
| 12. | 101 | f | 107 | f | 181 | f | 47 | f | 97 | f | 141 | f | 103 | f | 45 | f | 43 | f |
| 13. | 99 | | 99 | | 51 | | 26 | | 55 | | 60 | | 55 | | 22 | | 23 | f |
| 14. | 96 | | 95 | | 49 | | 22 | | 96 | | 57 | | 95 | | 16 | | 22 | |
| 15. | 1081 | | 2002 | a | 841 | f | 1277 | | 793 | | 2003 | a | 1597 | | 951 | f | 839 | |
| 16. | 1575 | | 2004 | a | 298 | f | 96 | | 1575 | | 2001 | a | 2004 | a | 124 | | 96 | |
| 17. | 131 | | 306 | | 527 | | 149 | f | 131 | | 166 | | 306 | | 109 | f | 159 | f |
| 18. | 27 | | 27 | | 30 | | 16 | | 17 | | 41 | | 17 | | 16 | | 13 | |
| 19. | 2000 | a | 2000 | a | 2000 | a | 229 | f | 2001 | a | 2003 | a | 2056 | a | 226 | f | 226 | f |
| 20. | 2002 | a | 2001 | a | 2024 | a | 600 | | 2000 | a | 2002 | a | 2000 | a | 310 | | 599 | |
| 21. | 311 | | 311 | | 78 | | 164 | | 195 | | 100 | | 195 | | 116 | | 144 | |
| 22. | 2002 | a | 2001 | a | 631 | f | 1628 | f | 2002 | a | 1884 | f | 2000 | a | 992 | f | 1416 | f |
| 23. | 62 | f | 62 | f | 44 | f | 102 | f | 62 | f | 52 | f | 62 | f | 78 | f | 102 | f |
| 24. | 1041 | | 1414 | | 664 | | 67 | | 441 | | 1014 | | 559 | | 512 | | 75 | |
| 25. | 55 | | 63 | | 29 | | 64 | | 55 | | 35 | | 63 | | 26 | | 64 | |
| 26. | 140 | | 124 | | 56 | | 23 | | 140 | | 24 | | 124 | | 12 | | 23 | |
| 27. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 28. | 353 | | 419 | | 303 | | 146 | | 144 | | 283 | | 172 | | 119 | | 139 | |
| 29. | 7 | | 7 | | 7 | | 7 | | 7 | | 6 | | 7 | | 7 | | 7 | |
| 30. | 117 | | 396 | | 38 | | 160 | | 193 | | 84 | | 331 | | 87 | | 212 | |
| 31. | 343 | | 309 | | 160 | | 26 | | 343 | | 53 | | 309 | | 21 | | 26 | |
| 32. | 225 | | 532 | | 77 | | 130 | | 245 | | 265 | | 753 | | 51 | | 133 | |
| 33. | 2000 | a | 2001 | a | 1032 | f | 657 | f | 2001 | a | 1820 | f | 2000 | a | 388 | f | 500 | f |
| 34. | 282 | f | 338 | f | 238 | f | 185 | f | 276 | f | 198 | f | 332 | f | 116 | f | 183 | f |
| 35. | 849 | f | 934 | | 650 | f | 116 | | 464 | | 2001 | a | 497 | | 145 | | 106 | |
| 36. | 2001 | a | 2001 | a | 2000 | a | 2001 | a | 2000 | a | 2000 | a | 2000 | a | 2000 | a | 2000 | a |
| 37. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 38. | 196 | | 188 | | 163 | | 24 | | 196 | | 81 | | 188 | | 18 | | 24 | |
| 39. | 515 | f | 521 | f | 309 | f | 173 | f | 403 | f | 238 | f | 358 | f | 173 | f | 161 | f |
| 40. | 1298 | f | 1332 | f | 990 | f | 2003 | a | 1291 | f | 1477 | f | 1325 | f | 1148 | f | 2003 | a |
| 41. | 0 | | 0 | | 0 | | 0 | | 36 | | 0 | | 32 | | 0 | | 35 | |
| 42. | 136 | f | 130 | f | 120 | f | 109 | f | 133 | f | 111 | f | 127 | f | 71 | f | 118 | f |
| 43. | 335 | | 674 | | 285 | | 77 | | 335 | | 163 | | 674 | | 52 | | 77 | |
| 44. | 361 | f | 489 | f | 304 | f | 408 | f | 357 | f | 444 | f | 485 | f | 138 | f | 408 | f |
| 45. | 193 | f | 480 | f | 96 | f | 157 | f | 193 | f | 150 | f | 480 | f | 81 | f | 157 | f |
| 46. | 260 | f | 524 | f | 200 | f | 166 | f | 260 | f | 475 | f | 524 | f | 133 | f | 166 | f |
| 47. | 2002 | a | 2000 | a | 2001 | a | 2000 | a | 2002 | a | 2002 | a | 2000 | a | 2000 | a | 2000 | a |
| 48. | 80 | f | 113 | f | 62 | f | 67 | f | 80 | f | 117 | f | 113 | f | 55 | f | 67 | f |
| 49. | 227 | f | 331 | f | 64 | f | 215 | f | 184 | f | 114 | f | 288 | f | 49 | f | 175 | f |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50. | 1161 | f | 955 | f | 1126 | f | 147 | f | 1161 | f | 1353 | f | 955 | f | 104 | f | 145 | f |
| 51. | 414 | f | 1101 | f | 154 | | 177 | f | 358 | | 150 | | 767 | | 114 | f | 141 | |
| 52. | 93 | | 93 | | 28 | | 82 | | 93 | | 68 | | 93 | | 118 | | 82 | |
| 53. | 778 | f | 1082 | f | 412 | f | 1293 | f | 544 | f | 316 | f | 759 | f | 544 | f | 783 | f |
| 54. | 381 | | 381 | | 233 | | 104 | | 163 | | 113 | | 163 | | 58 | | 73 | |
| 55. | 20 | | 20 | | 20 | | 27 | | 13 | | 18 | | 13 | | 13 | | 20 | |
| 56. | 34 | f | 34 | f | 33 | f | 26 | f | 34 | f | 34 | f | 34 | f | 25 | f | 26 | f |
| 57. | 1148 | | 2002 | a | 852 | | 600 | | 620 | | 938 | | 2000 | a | 332 | | 147 | |
| 58. | 2001 | a | 2000 | a | 1516 | f | 123 | | 2000 | a | 2000 | a | 2000 | a | 103 | | 120 | |
| 59. | 195 | | 195 | | 88 | | 27 | | 120 | | 45 | | 120 | | 22 | | 24 | |
| 60. | 314 | | 251 | | 168 | | 29 | | 314 | | 22 | | 251 | | 22 | | 29 | |
| 61. | 203 | | 355 | | 151 | | 294 | | 203 | | 374 | | 355 | | 272 | | 294 | |
| 62. | 246 | | 246 | | 68 | | 91 | | 245 | | 80 | | 245 | | 47 | | 91 | |
| 63. | 58 | | 88 | | 23 | | 72 | | 51 | | 27 | | 67 | | 24 | | 81 | |
| 64. | 13 | | 13 | | 13 | | 13 | | 10 | | 34 | | 10 | | 10 | | 10 | |
| 65. | 247 | | 184 | | 1174 | f | 17 | | 149 | | 41 | | 111 | | 17 | | 14 | |
| 66. | 706 | f | 1571 | f | 2013 | a | 623 | f | 706 | f | 2001 | a | 1571 | f | 1806 | f | 623 | f |
| 67. | 104 | | 141 | | 18 | | 75 | | 104 | | 28 | | 141 | | 9 | | 75 | |
| 68. | 807 | f | 953 | f | 599 | f | 2000 | a | 548 | f | 825 | f | 672 | f | 1198 | f | 1130 | f |
| 69. | 428 | f | 521 | f | 441 | f | 310 | f | 428 | f | 316 | f | 521 | f | 330 | f | 310 | f |
| 70. | 201 | | 638 | | 63 | | 46 | | 201 | | 91 | | 638 | | 32 | | 46 | |
| 71. | 2001 | a | 2000 | a | 911 | f | 961 | f | 2001 | a | 865 | f | 2000 | a | 519 | f | 961 | f |
| 72. | 1231 | f | 2065 | a | 1935 | f | 2003 | a | 1406 | f | 2001 | a | 2000 | a | 626 | f | 900 | f |
| 73. | 62 | | 60 | | 217 | f | 133 | | 62 | | 48 | f | 60 | | 221 | f | 133 | |
| 74. | 146 | | 179 | | 168 | | 15 | | 139 | | 127 | f | 172 | | 62 | f | 15 | |
| 75. | 2000 | a | 2000 | a | 869 | f | 2000 | a | 2000 | a | 2000 | a | 2000 | a | 507 | f | 2000 | a |
| 76. | 705 | f | 1710 | f | 578 | f | 185 | | 362 | f | 1026 | f | 922 | f | 423 | f | 176 | |
| 77. | 257 | f | 281 | f | 86 | f | 281 | f | 155 | f | 216 | f | 170 | f | 83 | f | 276 | f |
| 78. | 1226 | f | 2002 | a | 513 | f | 664 | f | 960 | f | 983 | f | 2000 | a | 161 | f | 763 | f |
| 79. | 643 | | 1082 | | 538 | | 60 | | 643 | | 473 | | 1082 | | 236 | | 60 | |
| 80. | 331 | | 624 | | 372 | | 353 | f | 250 | | 368 | | 603 | | 256 | f | 349 | f |
| 81. | 743 | f | 1029 | f | 312 | f | 199 | f | 1200 | f | 685 | f | 1488 | f | 233 | f | 198 | f |
| 82. | 128 | f | 140 | | 111 | | 83 | | 128 | f | 91 | | 140 | | 65 | | 83 | |
| 83. | 1722 | f | 1601 | f | 1371 | f | 1166 | f | 1367 | f | 2003 | a | 1396 | f | 210 | f | 921 | f |
| 84. | 268 | | 756 | | 109 | | 144 | | 146 | | 54 | f | 416 | | 129 | | 128 | |
| 85. | 88 | | 151 | | 25 | | 99 | | 85 | | 42 | | 148 | | 16 | | 96 | |
| 86. | 380 | | 744 | | 72 | f | 213 | | 290 | | 75 | | 540 | | 64 | | 213 | |
| 87. | 27 | | 24 | | 17 | | 21 | | 59 | | 34 | | 59 | | 16 | | 23 | |
| 88. | 192 | f | 277 | f | 215 | f | 94 | f | 192 | f | 261 | f | 277 | f | 59 | f | 94 | f |
| 89. | 2000 | a | 2003 | a | 2006 | a | 101 | f | 1389 | f | 2005 | a | 2069 | a | 79 | f | 99 | f |
| 90. | 297 | f | 544 | f | 42 | f | 42 | f | 218 | f | 201 | f | 400 | f | 70 | f | 76 | f |
| 91. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 3 | | 4 | |
| 92. | 43 | | 39 | | 24 | | 124 | | 32 | | 14 | | 28 | | 32 | | 106 | |
| 93. | 965 | f | 1761 | f | 1088 | f | 1826 | f | 726 | f | 1251 | f | 865 | f | 1096 | f | 923 | f |
| 94. | 102 | | 100 | | 27 | | 24 | | 102 | | 28 | | 100 | | 11 | | 24 | |
| 95. | 96 | f | 84 | f | 120 | f | 36 | f | 187 | f | 61 | f | 154 | f | 100 | f | 190 | f |
| 96. | 344 | | 663 | | 496 | f | 106 | | 245 | | 678 | | 464 | | 52 | | 95 | |
| 97. | 160 | | 437 | | 91 | | 53 | | 160 | | 33 | | 437 | | 31 | | 53 | |
| 98. | 166 | | 166 | | 88 | | 77 | | 154 | | 114 | | 154 | | 224 | | 65 | |
| 99. | 481 | f | 419 | f | 1044 | f | 126 | f | 481 | f | 72 | f | 419 | f | 47 | f | 126 | f |
| 100. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 52329 | | 66006 | | 43196 | | 32811 | | 47146 | | 53519 | | 59918 | | 25332 | | 27888 | |

## C.8.3 Operators

### C.8.3.1 Controlled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 2. | 26 | | 26 | | 28 | | 16 | | 26 | | 26 | | 26 | | 20 | | 18 | |
| 3. | 11 | | 11 | | 11 | | 9 | | 11 | | 11 | | 11 | | 13 | | 9 | |
| 4. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 5. | 5 | | 5 | | 7 | | 5 | | 5 | | 7 | | 5 | | 9 | | 5 | |
| 6. | 10 | | 10 | | 12 | | 10 | | 12 | | 10 | | 10 | | 12 | | 10 | |
| 7. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 8. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 9. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 10. | 23 | | 0 | a | 23 | | 21 | | 23 | | 23 | | 23 | | 0 | f | 21 | |
| 11. | 13 | | 0 | a | 15 | | 13 | | 13 | | 15 | | 0 | a | 13 | | 13 | |
| 12. | 10 | | 10 | | 10 | | 0 | f | 10 | | 10 | | 10 | | 10 | | 0 | f |
| 13. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 14. | 16 | | 16 | | 0 | f | 18 | | 16 | | 0 | f | 16 | | 18 | | 18 | |
| 15. | 11 | | 9 | | 9 | | 11 | | 13 | | 9 | | 13 | | 9 | | 13 | |
| 16. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 17. | 7 | | 7 | | 7 | | 7 | | 7 | | 13 | | 7 | | 7 | | 7 | |
| 18. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 19. | 25 | | 25 | | 25 | | 21 | | 20 | | 25 | | 20 | | 21 | | 23 | |
| 20. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 21. | 12 | | 12 | | 12 | | 12 | | 14 | | 12 | | 14 | | 16 | | 16 | |
| 22. | 8 | | 6 | | 8 | | 6 | | 8 | | 6 | | 6 | | 6 | | 6 | |
| 23. | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | | 25 | | 17 | |
| 24. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 25. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 26. | 1 | | 1 | | 1 | | 1 | | 3 | | 1 | | 3 | | 1 | | 5 | |
| 27. | 13 | | 11 | | 13 | | 15 | | 13 | | 11 | | 11 | | 15 | | 15 | |
| 28. | 6 | | 6 | | 20 | | 6 | | 6 | | 0 | f | 6 | | 6 | | 6 | |
| 29. | 12 | | 12 | | 0 | f | 12 | | 14 | | 16 | | 14 | | 12 | | 14 | |
| 30. | 3 | | 3 | | 3 | | 5 | | 3 | | 3 | | 3 | | 5 | | 5 | |
| 31. | 12 | | 0 | a | 12 | | 14 | | 12 | | 10 | | 0 | a | 14 | | 14 | |
| 32. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 33. | 10 | | 8 | | 8 | | 8 | | 10 | | 8 | | 8 | | 8 | | 8 | |
| 34. | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | |
| 35. | 12 | | 12 | | 12 | | 13 | | 12 | | 14 | | 12 | | 13 | | 13 | |
| 36. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 6 | | 4 | |
| 37. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 38. | 6 | | 6 | | 8 | | 6 | | 6 | | 12 | | 6 | | 6 | | 6 | |
| 39. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 40. | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 41. | 8 | | 8 | | 0 | f | 8 | | 8 | | 0 | f | 8 | | 0 | f | 8 | |
| 42. | 19 | | 19 | | 0 | f | 7 | | 19 | | 7 | | 19 | | 7 | | 7 | |
| 43. | 10 | | 10 | | 10 | | 12 | | 10 | | 10 | | 10 | | 12 | | 12 | |
| 44. | 11 | | 11 | | 11 | | 0 | f | 11 | | 11 | | 11 | | 0 | f | 0 | f |
| 45. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 46. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 47. | 11 | | 11 | | 17 | | 11 | | 11 | | 11 | | 11 | | 13 | | 11 | |
| 48. | 5 | | 5 | | 6 | | 7 | | 5 | | 5 | | 5 | | 6 | | 7 | |
| 49. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 50. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 0 | f | 3 | |
| 51. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 52. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 0 | f | 8 | |
| 53. | 7 | | 7 | | 7 | | 11 | | 7 | | 7 | | 7 | | 11 | | 11 | |
| 54. | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | |
| 55. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| 56. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 57. | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | | 17 | | 15 | |
| 58. | 13 | | 11 | | 13 | | 15 | | 13 | | 11 | | 11 | | 15 | | 15 | |
| 59. | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | | 15 | |
| 60. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 9 | | 7 | |
| 61. | 10 | | 10 | | 0 | f | 9 | | 0 | f | 0 | f | 0 | f | 0 | f | 9 | |
| 62. | 10 | | 10 | | 0 | f | 10 | | 12 | | 0 | f | 12 | | 0 | f | 12 | |
| 63. | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | | 2 | |
| 64. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 65. | 14 | | 14 | | 18 | | 0 | f | 0 | f | 18 | | 0 | f | 0 | f | 0 | f |
| 66. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 67. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 68. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 69. | 17 | | 17 | | 17 | | 7 | | 17 | | 17 | | 17 | | 0 | f | 7 | |
| 70. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 71. | 6 | | 6 | | 8 | | 6 | | 8 | | 8 | | 8 | | 0 | f | 8 | |
| 72. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 73. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 74. | 9 | | 9 | | 9 | | 5 | | 9 | | 9 | | 9 | | 5 | | 5 | |
| 75. | 6 | | 6 | | 8 | | 8 | | 8 | | 6 | | 8 | | 6 | | 8 | |
| 76. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 77. | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 78. | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 79. | 19 | | 19 | | 29 | | 11 | | 23 | | 29 | | 23 | | 13 | | 11 | |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80. | 24 | | 24 | | 24 | | 12 | | 26 | | 0 | f | 26 | | 26 | | 14 | |
| 81. | 8 | | 8 | | 10 | | 8 | | 8 | | 10 | | 8 | | 8 | | 8 | |
| 82. | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | | 1 | |
| 83. | 11 | | 11 | | 0 | f | 11 | | 11 | | 0 | f | 11 | | 0 | f | 11 | |
| 84. | 8 | | 8 | | 8 | | 8 | | 8 | | 0 | f | 8 | | 0 | f | 8 | |
| 85. | 10 | | 10 | | 14 | | 14 | | 10 | | 14 | | 10 | | 0 | f | 14 | |
| 86. | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | |
| 87. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 88. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 89. | 11 | | 0 | a | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 90. | 9 | | 9 | | 11 | | 9 | | 9 | | 9 | | 9 | | 9 | | 9 | |
| 91. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 92. | 13 | | 13 | | 13 | | 13 | | 13 | | 13 | | 13 | | 13 | | 13 | |
| 93. | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 94. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 95. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 96. | 4 | | 4 | | 4 | | 6 | | 4 | | 4 | | 4 | | 6 | | 6 | |
| 97. | 15 | | 15 | | 17 | | 21 | | 15 | | 17 | | 15 | | 21 | | 23 | |
| 98. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 99. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 100. | 20 | | 20 | | 20 | | 14 | | 22 | | 0 | f | 22 | | 14 | | 16 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 913 | | 844 | | 882 | | 836 | | 908 | | 822 | | 873 | | 786 | | 862 | |

### C.8.3.2 Uncontrolled Problem Sample

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | | | | | | | | | | | | | | | | | | |
| 1. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 2. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 3. | 26 | | 26 | | 28 | | 16 | | 26 | | 26 | | 26 | | 20 | | 18 | |
| 4. | 0 | f | 14 | | 0 | f | 0 | f | 0 | f | 0 | a | 14 | | 0 | f | 0 | f |
| 5. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 6. | 11 | | 11 | | 11 | | 9 | | 11 | | 11 | | 11 | | 13 | | 9 | |
| 7. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 8. | 5 | | 5 | | 7 | | 5 | | 5 | | 7 | | 5 | | 9 | | 5 | |
| 9. | 10 | | 10 | | 12 | | 10 | | 12 | | 10 | | 10 | | 12 | | 10 | |
| 10. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 11. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 12. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 13. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 0 | f |
| 14. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 15. | 23 | | 0 | a | 0 | f | 21 | | 23 | | 0 | a | 23 | | 0 | f | 21 | |
| 16. | 13 | | 0 | a | 0 | f | 13 | | 13 | | 0 | a | 0 | a | 13 | | 13 | |
| 17. | 10 | | 10 | | 12 | | 0 | f | 10 | | 10 | | 10 | | 0 | f | 0 | f |
| 18. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 19. | 0 | a | 0 | a | 0 | a | 0 | f | 0 | a | 0 | a | 0 | a | 0 | f | 0 | f |
| 20. | 0 | a | 0 | a | 0 | a | 22 | | 0 | a | 0 | a | 0 | a | 26 | | 22 | |
| 21. | 16 | | 16 | | 16 | | 18 | | 16 | | 16 | | 16 | | 18 | | 18 | |
| 22. | 0 | a | 0 | a | 0 | f | 0 | f | 0 | a | 0 | f | 0 | a | 0 | f | 0 | f |
| 23. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 24. | 11 | | 9 | | 9 | | 11 | | 13 | | 9 | | 13 | | 9 | | 13 | |
| 25. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 26. | 7 | | 7 | | 13 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 27. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 28. | 25 | | 25 | | 25 | | 21 | | 20 | | 25 | | 20 | | 21 | | 23 | |
| 29. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 30. | 12 | | 12 | | 12 | | 12 | | 14 | | 12 | | 14 | | 16 | | 16 | |
| 31. | 8 | | 6 | | 8 | | 6 | | 8 | | 6 | | 6 | | 6 | | 6 | |
| 32. | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | | 17 | |
| 33. | 0 | a | 0 | a | 0 | f | 0 | f | 0 | a | 0 | f | 0 | a | 0 | f | 0 | f |
| 34. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 35. | 0 | f | 16 | | 0 | f | 20 | | 20 | | 0 | a | 16 | | 20 | | 20 | |
| 36. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a |
| 37. | 0 | | 0 | | 0 | a | 0 | | 0 | | 0 | a | 0 | a | 0 | a | 0 | a |
| 38. | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 39. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |

| | $MEA$ | | $MEA_1$ | | $MEA_2$ | | $MEA_3$ | | $MEA_4$ | | $MEA_5$ | | $MEA_6$ | | $MEA_7$ | | $MEA_8$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40. | 0 | f | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f | 0 | f | 0 | f | 0 | a |
| 41. | 1 | | 1 | | 1 | | 1 | | 3 | | 1 | | 3 | | 1 | | 5 | |
| 42. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 43. | 13 | | 11 | | 13 | | 15 | | 13 | | 11 | | 11 | | 15 | | 15 | |
| 44. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 45. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 46. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 47. | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a | 0 | a |
| 48. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 49. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 50. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 51. | 0 | f | 0 | f | 18 | | 0 | f | 18 | | 18 | | 18 | | 0 | f | 20 | |
| 52. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 53. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 54. | 12 | | 12 | | 12 | | 12 | | 14 | | 16 | | 14 | | 16 | | 14 | |
| 55. | 3 | | 3 | | 3 | | 5 | | 3 | | 3 | | 3 | | 3 | | 5 | |
| 56. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 57. | 12 | | 0 | a | 12 | | 14 | | 12 | | 10 | | 0 | a | 14 | | 14 | |
| 58. | 0 | a | 0 | a | 0 | f | 28 | | 0 | a | 0 | a | 0 | a | 28 | | 28 | |
| 59. | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | | 6 | |
| 60. | 10 | | 8 | | 10 | | 8 | | 10 | | 8 | | 8 | | 8 | | 8 | |
| 61. | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | | 14 | |
| 62. | 12 | | 12 | | 12 | | 13 | | 12 | | 12 | | 12 | | 13 | | 13 | |
| 63. | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | | 4 | |
| 64. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 65. | 6 | | 6 | | 0 | f | 6 | | 6 | | 8 | | 6 | | 6 | | 6 | |
| 66. | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f | 0 | f |
| 67. | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | | 7 | |
| 68. | 0 | f | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 69. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 70. | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | | 11 | |
| 71. | 0 | a | 0 | a | 0 | f | 0 | f | 0 | a | 0 | f | 0 | a | 0 | f | 0 | f |
| 72. | 0 | f | 0 | a | 0 | f | 0 | a | 0 | f | 0 | a | 0 | a | 0 | f | 0 | f |
| 73. | 8 | | 8 | | 0 | f | 8 | | 8 | | 0 | f | 8 | | 0 | f | 8 | |
| 74. | 19 | | 19 | | 21 | | 7 | | 19 | | 0 | f | 19 | | 0 | f | 7 | |
| 75. | 0 | a | 0 | a | 0 | f | 0 | a | 0 | a | 0 | a | 0 | a | 0 | f | 0 | a |
| 76. | 0 | f | 0 | f | 0 | f | 24 | | 0 | f | 0 | f | 0 | f | 0 | f | 24 | |
| 77. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 78. | 0 | f | 0 | a | 0 | f | 0 | f | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f |
| 79. | 10 | | 10 | | 12 | | 12 | | 10 | | 10 | | 10 | | 12 | | 12 | |
| 80. | 11 | | 11 | | 24 | | 0 | f | 11 | | 11 | | 11 | | 0 | f | 0 | f |
| 81. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 82. | 0 | f | 16 | | 16 | | 16 | | 0 | f | 18 | | 16 | | 16 | | 16 | |
| 83. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | a | 0 | f | 0 | f | 0 | f |
| 84. | 8 | | 8 | | 8 | | 8 | | 8 | | 0 | f | 8 | | 8 | | 8 | |
| 85. | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | | 10 | |
| 86. | 11 | | 11 | | 0 | f | 11 | | 11 | | 11 | | 11 | | 13 | | 11 | |
| 87. | 5 | | 5 | | 4 | | 7 | | 5 | | 5 | | 5 | | 4 | | 7 | |
| 88. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 89. | 0 | a | 0 | a | 0 | a | 0 | f | 0 | f | 0 | a | 0 | a | 0 | f | 0 | f |
| 90. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 91. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 92. | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | | 3 | |
| 93. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 94. | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | | 5 | |
| 95. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 96. | 8 | | 8 | | 0 | f | 8 | | 8 | | 8 | | 8 | | 8 | | 8 | |
| 97. | 7 | | 7 | | 7 | | 11 | | 7 | | 7 | | 7 | | 11 | | 11 | |
| 98. | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 32 | | 16 | |
| 99. | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f | 0 | f |
| 100. | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | |
| | **Total** | | | | | | | | | | | | | | | | | |
| | 492 | | 482 | | 485 | | 564 | | 535 | | 455 | | 528 | | 541 | | 592 | |

## C.9 Problem Sample

### C.9.1 Controlled Problem Sample

The table below lists the problems in this sample. The tasks are numbered down the left hand side and the first line of each problem is the goal condition $G$ and the second is the initial state. So for example task number 1 in this sample is:

$$G = \{crane\_at(s6), on\_floor(crate2, s13), clear(s2)\}$$
$$I = \{crane\_at(s14), clear(s9), clear(s7), clear(s13), clear(s1), clear(s6),$$
$$clear(s16), clear(s3), clear(s8), clear(s4), truck\_at(s2), driver\_at(s2),$$
$$at(crate3, s12), on\_floor(crate3, s12), at(crate2, s10),$$
$$on\_floor(crate2, s10), at(crate1, s14), on\_floor(crate1, s14),$$
$$in\_truck, truck\_unloaded, crane\_unloaded\}$$

The set of problems that could not be solved by all planning applications is:

$$\{10, 11, 12, 14, 28, 29, 31, 41, 42, 44, 50, 52, 61, 62, 65, 69, 71, 80, 83, 84, 85, 89, 100\}$$

A sample of 73 problems that could be solved by all of the planning applications was formed by removing this set of tasks from the sample.

| | Task |
|---|---|
| 1. | crane_at(s6) , on_floor(crate2 , s13) , clear(s2) |
| | crane_at(s14) , clear(s9) , clear(s7) , clear(s13) , clear(s1) , clear(s6) , clear(s16) , clear(s3) , clear(s8) , clear(s4) , truck_at(s2) , driver_at(s2) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 2. | truck_at(s2) , on_floor(crate1 , s8) , on_floor(crate3 , s16) |
| | crane_at(s10) , clear(s9) , clear(s7) , clear(s4) , clear(s6) , clear(s14) , clear(s13) , clear(s12) , clear(s8) , clear(s16) , truck_at(s2) , driver_at(s2) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 3. | clear(s12) , on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s14) , clear(s1) , clear(s7) , clear(s8) , clear(s12) , clear(s2) , clear(s10) , clear(s13) , clear(s4) , truck_at(s16) , driver_at(s16) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 4. | truck_at(s1) |
| | crane_at(s6) , clear(s7) , clear(s9) , clear(s1) , clear(s8) , clear(s4) , clear(s10) , clear(s6) , clear(s2) , clear(s13) , truck_at(s12) , driver_at(s12) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s16) , on_floor(crate1 , s16) , in_truck , truck_unloaded , crane_unloaded |
| 5. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s14) , clear(s8) , clear(s10) , clear(s7) , clear(s13) , clear(s16) , clear(s6) , clear(s4) , clear(s9) , truck_at(s3) , driver_at(s3) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 6. | at(crate3 , s6) |
| | crane_at(s9) , clear(s3) , clear(s16) , clear(s6) , clear(s10) , clear(s2) , clear(s1) , clear(s14) , clear(s7) , clear(s4) , truck_at(s12) , driver_at(s12) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 7. | on_floor(crate2 , s2) |
| | crane_at(s14) , clear(s7) , clear(s10) , clear(s4) , clear(s16) , clear(s12) , clear(s3) , clear(s13) , clear(s6) , clear(s9) , truck_at(s2) , driver_at(s2) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 8. | on_floor(crate2 , s16) , clear(s12) |
| | crane_at(s9) , clear(s4) , clear(s13) , clear(s6) , clear(s16) , clear(s8) , clear(s7) , clear(s3) , clear(s10) , clear(s2) , truck_at(s14) , driver_at(s14) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 9. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s16) , clear(s8) , clear(s10) , clear(s12) , clear(s6) , clear(s14) , clear(s1) , clear(s3) , clear(s2) , truck_at(s9) , driver_at(s9) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 10. | clear(s4) , at(crate1 , s2) |
| | crane_at(s9) , clear(s10) , clear(s1) , clear(s6) , clear(s12) , clear(s3) , clear(s13) , clear(s7) , clear(s9) , clear(s14) , truck_at(s16) , driver_at(s16) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |
| 11. | above_floor(crate3 , s10) |
| | crane_at(s9) , clear(s14) , clear(s8) , clear(s3) , clear(s12) , clear(s6) , clear(s9) , clear(s7) , clear(s16) , clear(s1) , truck_at(s13) , driver_at(s13) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 12. | crane_loaded(crate3 , s10) , truck_at(s1) |
| | crane_at(s10) , clear(s4) , clear(s12) , clear(s14) , clear(s13) , clear(s9) , clear(s2) , clear(s6) , clear(s8) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 13. | at(crate3 , s6) |
| | crane_at(s9) , clear(s10) , clear(s1) , clear(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s16) , clear(s2) , clear(s8) , truck_at(s9) , driver_at(s9) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 14. | truck_at(s1) , above_floor(crate1 , s10) , at(crate2 , s6) |
| | crane_at(s6) , clear(s16) , clear(s14) , clear(s9) , clear(s8) , clear(s10) , clear(s3) , clear(s7) , clear(s12) , clear(s2) , truck_at(s1) , driver_at(s1) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 15. | clear(s4) , at(crate3 , s2) , at(crate1 , s6) |
| | crane_at(s9) , clear(s7) , clear(s8) , clear(s3) , clear(s10) , clear(s16) , clear(s4) , clear(s9) , clear(s13) , clear(s1) , truck_at(s6) , driver_at(s6) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 16. | above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s14) , clear(s4) , clear(s7) , clear(s9) , clear(s8) , clear(s13) , clear(s3) , clear(s10) , clear(s16) , truck_at(s2) , driver_at(s2) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 17. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s6) , clear(s9) , clear(s4) , clear(s8) , clear(s10) , clear(s13) , clear(s2) , clear(s16) , clear(s12) , truck_at(s7) , driver_at(s7) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 18. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s8) , clear(s16) , clear(s13) , clear(s2) , clear(s14) , clear(s3) , clear(s12) , clear(s1) , clear(s10) , truck_at(s6) , driver_at(s6) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 19. | on_floor(crate1 , s4) , at(crate2 , s2) , on_floor(crate3 , s16) |
| | crane_at(s9) , clear(s16) , clear(s9) , clear(s4) , clear(s7) , clear(s6) , clear(s3) , clear(s8) , clear(s2) , clear(s12) , truck_at(s10) , driver_at(s10) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 20. | truck_at(s1) |
| | crane_at(s6) , clear(s7) , clear(s10) , clear(s16) , clear(s2) , clear(s12) , clear(s3) , clear(s6) , clear(s1) , clear(s4) , truck_at(s8) , driver_at(s8) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 21. | clear(s12) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s3) , clear(s14) , clear(s13) , clear(s6) , clear(s16) , clear(s10) , clear(s9) , clear(s1) , clear(s2) , truck_at(s4) , driver_at(s4) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 22. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s8) , clear(s16) , clear(s12) , clear(s14) , clear(s4) , clear(s6) , clear(s10) , clear(s3) , clear(s1) , truck_at(s7) , driver_at(s7) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 23. | on_floor(crate1 , s1) , stacked(crate2 , shelf6) |
| | crane_at(s14) , clear(s9) , clear(s16) , clear(s1) , clear(s14) , clear(s6) , clear(s2) , clear(s13) , clear(s10) , clear(s7) , truck_at(s12) , driver_at(s12) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 24. | clear(s9) |
| | crane_at(s9) , clear(s9) , clear(s14) , clear(s8) , clear(s6) , clear(s12) , clear(s7) , clear(s10) , clear(s4) , clear(s16) , truck_at(s3) , driver_at(s3) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 25. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s14) , clear(s3) , clear(s10) , clear(s13) , clear(s6) , clear(s7) , clear(s16) , clear(s4) , clear(s2) , truck_at(s8) , driver_at(s8) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 26. | at(crate2 , s2) |
| | crane_at(s14) , clear(s12) , clear(s16) , clear(s1) , clear(s8) , clear(s9) , clear(s4) , clear(s13) , clear(s7) , clear(s3) , truck_at(s2) , driver_at(s2) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 27. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s12) , clear(s4) , clear(s7) , clear(s2) , clear(s14) , clear(s6) , clear(s13) , clear(s1) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 28. | crane_loaded(crate2 , s9) |
| | crane_at(s10) , clear(s14) , clear(s12) , clear(s1) , clear(s16) , clear(s8) , clear(s13) , clear(s7) , clear(s9) , clear(s6) , truck_at(s2) , driver_at(s2) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 29. | at(crate3 , s16) , at(crate1 , s6) |
| | crane_at(s9) , clear(s12) , clear(s16) , clear(s8) , clear(s13) , clear(s2) , clear(s14) , clear(s4) , clear(s6) , clear(s1) , truck_at(s10) , driver_at(s10) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 30. | at(crate2 , s2) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s10) , clear(s2) , clear(s14) , clear(s9) , clear(s7) , clear(s3) , clear(s16) , truck_at(s4) , driver_at(s4) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 31. | at(crate3 , s9) , clear(s12) |
| | crane_at(s10) , clear(s12) , clear(s3) , clear(s6) , clear(s7) , clear(s13) , clear(s1) , clear(s8) , clear(s14) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 32. | at(crate2 , s2) |
| | crane_at(s9) , clear(s6) , clear(s7) , clear(s3) , clear(s1) , clear(s8) , clear(s12) , clear(s14) , clear(s10) , clear(s2) , truck_at(s16) , driver_at(s16) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 33. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s13) , clear(s4) , clear(s8) , clear(s9) , clear(s3) , clear(s10) , clear(s2) , clear(s16) , clear(s6) , truck_at(s14) , driver_at(s14) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 34. | driver_on_floor(s9) , above_floor(crate1 , s10) , clear(s12) |
| | crane_at(s10) , clear(s7) , clear(s1) , clear(s4) , clear(s14) , clear(s16) , clear(s8) , clear(s9) , clear(s2) , clear(s6) , truck_at(s12) , driver_at(s12) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 35. | above_floor(crate1 , s10) , on_floor(crate3 , s16) |
| | crane_at(s6) , clear(s7) , clear(s3) , clear(s16) , clear(s10) , clear(s13) , clear(s8) , clear(s9) , clear(s1) , clear(s12) , truck_at(s4) , driver_at(s4) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 36. | at(crate3 , s2) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s7) , clear(s10) , clear(s2) , clear(s9) , clear(s16) , clear(s3) , clear(s8) , truck_at(s6) , driver_at(s6) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 37. | at(crate1 , s6) |
| | crane_at(s6) , clear(s12) , clear(s2) , clear(s8) , clear(s10) , clear(s7) , clear(s1) , clear(s16) , clear(s9) , clear(s13) , truck_at(s6) , driver_at(s6) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 38. | at(crate3 , s3) , in_truck , truck_at(s1) |
| | crane_at(s9) , clear(s10) , clear(s13) , clear(s2) , clear(s16) , clear(s14) , clear(s7) , clear(s12) , clear(s9) , clear(s3) , truck_at(s8) , driver_at(s8) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 39. | above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s9) , clear(s10) , clear(s16) , clear(s4) , clear(s14) , clear(s8) , truck_at(s2) , driver_at(s2) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 40. | crane_loaded(crate1 , s14) , clear(s12) |
| | crane_at(s6) , clear(s14) , clear(s1) , clear(s3) , clear(s13) , clear(s8) , clear(s7) , clear(s6) , clear(s12) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 41. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s6) , clear(s14) , clear(s7) , clear(s12) , clear(s9) , clear(s10) , clear(s13) , clear(s8) , clear(s16) , truck_at(s1) , driver_at(s1) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 42. | truck_at(s1) , on_floor(crate2 , s3) |
| | crane_at(s9) , clear(s7) , clear(s10) , clear(s13) , clear(s16) , clear(s2) , clear(s8) , clear(s6) , clear(s14) , clear(s12) , truck_at(s4) , driver_at(s4) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 43. | above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s16) , clear(s13) , clear(s7) , clear(s8) , clear(s9) , clear(s6) , clear(s2) , clear(s3) , clear(s4) , truck_at(s14) , driver_at(s14) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 44. | at(crate2 , s10) , in_truck , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s4) , clear(s10) , clear(s14) , clear(s2) , clear(s7) , clear(s13) , clear(s1) , clear(s9) , clear(s12) , truck_at(s8) , driver_at(s8) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 45. | on_floor(crate1 , s1) , at(crate2 , s2) |
| | crane_at(s10) , clear(s13) , clear(s12) , clear(s6) , clear(s7) , clear(s10) , clear(s3) , clear(s14) , clear(s9) , clear(s2) , truck_at(s16) , driver_at(s16) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 46. | crane_loaded(crate1 , s10) , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s13) , clear(s3) , clear(s10) , clear(s14) , clear(s9) , clear(s12) , clear(s4) , clear(s1) , clear(s16) , truck_at(s7) , driver_at(s7) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 47. | crane_at(s14) , in_truck , on_floor(crate3 , s6) |
| | crane_at(s9) , clear(s8) , clear(s2) , clear(s13) , clear(s10) , clear(s14) , clear(s6) , clear(s12) , clear(s1) , clear(s16) , truck_at(s9) , driver_at(s9) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 48. | stacked(crate2 , shelf6) |
| | crane_at(s6) , clear(s1) , clear(s2) , clear(s7) , clear(s12) , clear(s6) , clear(s14) , clear(s13) , clear(s8) , clear(s10) , truck_at(s3) , driver_at(s3) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 49. | driver_on_floor(s1) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s2) , clear(s3) , clear(s10) , clear(s6) , clear(s9) , clear(s16) , clear(s13) , truck_at(s7) , driver_at(s7) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 50. | at(crate3 , s2) |
| | crane_at(s6) , clear(s7) , clear(s1) , clear(s13) , clear(s16) , clear(s9) , clear(s2) , clear(s8) , clear(s12) , clear(s3) , truck_at(s10) , driver_at(s10) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 51. | in_truck , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s14) , clear(s8) , clear(s10) , clear(s16) , clear(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s7) , truck_at(s9) , driver_at(s9) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 52. | at(crate2 , s2) |
| | crane_at(s14) , clear(s14) , clear(s16) , clear(s13) , clear(s3) , clear(s2) , clear(s1) , clear(s10) , clear(s4) , clear(s7) , truck_at(s8) , driver_at(s8) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 53. | above_floor(crate1 , s10) |
| | crane_at(s14) , clear(s4) , clear(s9) , clear(s8) , clear(s13) , clear(s16) , clear(s14) , clear(s3) , clear(s6) , clear(s2) , truck_at(s10) , driver_at(s10) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 54. | on_floor(crate1 , s4) , driver_on_floor(s9) , at(crate2 , s2) |
| | crane_at(s9) , clear(s4) , clear(s10) , clear(s16) , clear(s12) , clear(s6) , clear(s3) , clear(s9) , clear(s2) , clear(s8) , truck_at(s14) , driver_at(s14) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 55. | at(crate2 , s2) |
| | crane_at(s14) , clear(s1) , clear(s3) , clear(s4) , clear(s10) , clear(s14) , clear(s9) , clear(s7) , clear(s16) , clear(s12) , truck_at(s8) , driver_at(s8) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 56. | on_floor(crate3 , s6) |
| | crane_at(s9) , clear(s7) , clear(s8) , clear(s4) , clear(s9) , clear(s3) , clear(s12) , clear(s13) , clear(s14) , clear(s16) , truck_at(s6) , driver_at(s6) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 57. | at(crate3 , s2) |
| | crane_at(s14) , clear(s3) , clear(s14) , clear(s2) , clear(s9) , clear(s7) , clear(s12) , clear(s1) , clear(s4) , clear(s8) , truck_at(s13) , driver_at(s13) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 58. | above_floor(crate2 , s10) , truck_at(s1) |
| | crane_at(s9) , clear(s8) , clear(s3) , clear(s16) , clear(s4) , clear(s2) , clear(s9) , clear(s13) , clear(s6) , clear(s10) , truck_at(s12) , driver_at(s12) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 59. | on_floor(crate3 , s14) , truck_at(s1) |
| | crane_at(s10) , clear(s3) , clear(s16) , clear(s10) , clear(s7) , clear(s9) , clear(s14) , clear(s4) , clear(s6) , clear(s13) , truck_at(s2) , driver_at(s2) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 60. | clear(s12) |
| | crane_at(s9) , clear(s10) , clear(s9) , clear(s1) , clear(s3) , clear(s16) , clear(s4) , clear(s6) , clear(s13) , clear(s8) , truck_at(s14) , driver_at(s14) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 61. | at(crate3 , s13) , on_floor(crate1 , s1) |
| | crane_at(s9) , clear(s14) , clear(s12) , clear(s13) , clear(s16) , clear(s8) , clear(s9) , clear(s1) , clear(s6) , clear(s4) , truck_at(s7) , driver_at(s7) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 62. | on_floor(crate1 , s1) , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s8) , clear(s10) , clear(s16) , clear(s14) , clear(s3) , clear(s12) , clear(s2) , clear(s4) , clear(s1) , truck_at(s13) , driver_at(s13) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 63. | clear(s12) , crane_loaded(crate2 , s9) |
| | crane_at(s10) , clear(s7) , clear(s6) , clear(s3) , clear(s13) , clear(s8) , clear(s1) , clear(s16) , clear(s12) , clear(s2) , truck_at(s10) , driver_at(s10) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 64. | stacked(crate3 , shelf7) |
| | crane_at(s6) , clear(s13) , clear(s6) , clear(s2) , clear(s4) , clear(s14) , clear(s9) , clear(s12) , clear(s7) , clear(s8) , truck_at(s3) , driver_at(s3) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s16) , on_floor(crate2 , s16) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 65. | clear(s8) , crane_loaded(crate3 , s10) , driver_at(s2) |
| | crane_at(s10) , clear(s3) , clear(s7) , clear(s13) , clear(s1) , clear(s2) , clear(s6) , clear(s10) , clear(s16) , clear(s9) , truck_at(s14) , driver_at(s14) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |
| 66. | on_floor(crate1 , s1) |
| | crane_at(s9) , clear(s9) , clear(s7) , clear(s16) , clear(s14) , clear(s6) , clear(s2) , clear(s1) , clear(s8) , clear(s4) , truck_at(s10) , driver_at(s10) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 67. | on_floor(crate1 , s4) |
| | crane_at(s6) , clear(s14) , clear(s16) , clear(s4) , clear(s3) , clear(s13) , clear(s12) , clear(s2) , clear(s6) , clear(s7) , truck_at(s8) , driver_at(s8) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 68. | at(crate1 , s2) |
| | crane_at(s10) , clear(s8) , clear(s2) , clear(s7) , clear(s9) , clear(s10) , clear(s1) , clear(s12) , clear(s16) , clear(s13) , truck_at(s4) , driver_at(s4) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 69. | stacked(crate3 , shelf7) |
| | crane_at(s9) , clear(s3) , clear(s9) , clear(s6) , clear(s4) , clear(s12) , clear(s7) , clear(s2) , clear(s13) , clear(s8) , truck_at(s14) , driver_at(s14) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s16) , on_floor(crate2 , s16) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 70. | clear(s4) , at(crate1 , s6) |
| | crane_at(s14) , clear(s9) , clear(s4) , clear(s8) , clear(s7) , clear(s16) , clear(s1) , clear(s14) , clear(s3) , clear(s10) , truck_at(s13) , driver_at(s13) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 71. | at(crate1 , s6) |
| | crane_at(s6) , clear(s6) , clear(s9) , clear(s16) , clear(s4) , clear(s10) , clear(s1) , clear(s8) , clear(s14) , clear(s13) , truck_at(s12) , driver_at(s12) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 72. | on_floor(crate3 , s16) , clear(s2) |
| | crane_at(s14) , clear(s16) , clear(s8) , clear(s9) , clear(s14) , clear(s4) , clear(s12) , clear(s3) , clear(s2) , clear(s6) , truck_at(s10) , driver_at(s10) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 73. | truck_at(s1) |
| | crane_at(s14) , clear(s8) , clear(s7) , clear(s12) , clear(s14) , clear(s9) , clear(s1) , clear(s13) , clear(s6) , clear(s16) , truck_at(s2) , driver_at(s2) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 74. | truck_at(s1) , truck_loaded(crate2 , s1) |
| | crane_at(s6) , clear(s10) , clear(s7) , clear(s1) , clear(s8) , clear(s3) , clear(s4) , clear(s16) , clear(s9) , clear(s2) , truck_at(s6) , driver_at(s6) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 75. | at(crate1 , s6) |
| | crane_at(s10) , clear(s1) , clear(s9) , clear(s13) , clear(s4) , clear(s16) , clear(s7) , clear(s3) , clear(s10) , clear(s8) , truck_at(s12) , driver_at(s12) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 76. | crane_loaded(crate3 , s10) , clear(s12) |
| | crane_at(s10) , clear(s2) , clear(s1) , clear(s16) , clear(s4) , clear(s10) , clear(s14) , clear(s12) , clear(s3) , clear(s8) , truck_at(s6) , driver_at(s6) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 77. | on_floor(crate2 , s14) , clear(s12) , above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s14) , clear(s12) , clear(s7) , clear(s4) , clear(s10) , clear(s13) , clear(s2) , clear(s6) , clear(s1) , truck_at(s8) , driver_at(s8) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s16) , on_floor(crate2 , s16) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 78. | on_floor(crate3 , s16) |
| | crane_at(s6) , clear(s12) , clear(s2) , clear(s8) , clear(s13) , clear(s6) , clear(s10) , clear(s7) , clear(s4) , clear(s16) , truck_at(s14) , driver_at(s14) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 79. | at(crate1 , s6) , crane_loaded(crate2 , s9) |
| | crane_at(s9) , clear(s6) , clear(s10) , clear(s14) , clear(s16) , clear(s1) , clear(s9) , clear(s2) , clear(s7) , clear(s8) , truck_at(s12) , driver_at(s12) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 80. | clear(s12) , crane_loaded(crate3 , s10) , on_floor(crate1 , s4) |
| | crane_at(s9) , clear(s4) , clear(s7) , clear(s3) , clear(s9) , clear(s16) , clear(s12) , clear(s2) , clear(s13) , clear(s10) , truck_at(s14) , driver_at(s14) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 81. | truck_at(s1) |
| | crane_at(s10) , clear(s6) , clear(s8) , clear(s2) , clear(s7) , clear(s9) , clear(s14) , clear(s13) , clear(s12) , clear(s4) , truck_at(s16) , driver_at(s16) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 82. | clear(s2) |
| | crane_at(s9) , clear(s14) , clear(s8) , clear(s6) , clear(s4) , clear(s12) , clear(s9) , clear(s10) , clear(s3) , clear(s1) , truck_at(s2) , driver_at(s2) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 83. | crane_loaded(crate1 , s14) , truck_at(s1) |
| | crane_at(s10) , clear(s7) , clear(s6) , clear(s3) , clear(s14) , clear(s13) , clear(s16) , clear(s2) , clear(s1) , clear(s8) , truck_at(s4) , driver_at(s4) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 84. | crane_loaded(crate2 , s9) |
| | crane_at(s9) , clear(s16) , clear(s8) , clear(s7) , clear(s2) , clear(s14) , clear(s9) , clear(s13) , clear(s4) , clear(s12) , truck_at(s3) , driver_at(s3) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 85. | at(crate1 , s6) , at(crate2 , s2) |
| | crane_at(s6) , clear(s10) , clear(s16) , clear(s2) , clear(s3) , clear(s12) , clear(s6) , clear(s13) , clear(s1) , clear(s9) , truck_at(s8) , driver_at(s8) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 86. | crane_loaded(crate2 , s9) |
| | crane_at(s6) , clear(s9) , clear(s2) , clear(s16) , clear(s4) , clear(s13) , clear(s6) , clear(s8) , clear(s1) , clear(s7) , truck_at(s14) , driver_at(s14) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 87. | truck_at(s1) |
| | crane_at(s10) , clear(s12) , clear(s14) , clear(s13) , clear(s2) , clear(s8) , clear(s1) , clear(s9) , clear(s7) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 88. | above_floor(crate2 , s9) , clear(s4) , clear(s12) |
| | crane_at(s14) , clear(s3) , clear(s7) , clear(s2) , clear(s4) , clear(s16) , clear(s12) , clear(s1) , clear(s10) , clear(s6) , truck_at(s9) , driver_at(s9) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 89. | at(crate1 , s6) , clear(s4) , clear(s12) |
| | crane_at(s10) , clear(s16) , clear(s4) , clear(s12) , clear(s9) , clear(s1) , clear(s7) , clear(s14) , clear(s2) , clear(s3) , truck_at(s8) , driver_at(s8) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 90. | on_floor(crate3 , s16) |
| | crane_at(s9) , clear(s14) , clear(s6) , clear(s4) , clear(s1) , clear(s12) , clear(s10) , clear(s16) , clear(s7) , clear(s8) , truck_at(s3) , driver_at(s3) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 91. | on_floor(crate1 , s1) |
| | crane_at(s6) , clear(s3) , clear(s14) , clear(s2) , clear(s9) , clear(s1) , clear(s8) , clear(s4) , clear(s7) , clear(s12) , truck_at(s10) , driver_at(s10) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 92. | crane_loaded(crate2 , s9) |
| | crane_at(s10) , clear(s3) , clear(s2) , clear(s4) , clear(s14) , clear(s7) , clear(s1) , clear(s9) , clear(s16) , clear(s12) , truck_at(s8) , driver_at(s8) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 93. | at(crate1 , s2) |
| | crane_at(s9) , clear(s16) , clear(s8) , clear(s2) , clear(s9) , clear(s6) , clear(s4) , clear(s10) , clear(s14) , clear(s1) , truck_at(s13) , driver_at(s13) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 94. | truck_at(s1) |
| | crane_at(s14) , clear(s8) , clear(s9) , clear(s16) , clear(s4) , clear(s14) , clear(s1) , clear(s12) , clear(s3) , clear(s7) , truck_at(s6) , driver_at(s6) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 95. | on_floor(crate3 , s6) |
| | crane_at(s9) , clear(s16) , clear(s4) , clear(s8) , clear(s7) , clear(s2) , clear(s13) , clear(s3) , clear(s9) , clear(s6) , truck_at(s12) , driver_at(s12) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 96. | above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s3) , clear(s14) , clear(s2) , clear(s7) , clear(s4) , clear(s16) , clear(s6) , clear(s12) , clear(s8) , truck_at(s10) , driver_at(s10) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 97. | on_floor(crate2 , s14) , at(crate1 , s4) , truck_at(s1) |
| | crane_at(s9) , clear(s13) , clear(s9) , clear(s3) , clear(s14) , clear(s6) , clear(s16) , clear(s12) , clear(s7) , clear(s10) , truck_at(s2) , driver_at(s2) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |
| 98. | on_floor(crate1 , s1) |
| | crane_at(s9) , clear(s3) , clear(s7) , clear(s13) , clear(s2) , clear(s1) , clear(s8) , clear(s10) , clear(s4) , clear(s9) , truck_at(s6) , driver_at(s6) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 99. | truck_at(s1) |
| | crane_at(s10) , clear(s2) , clear(s16) , clear(s1) , clear(s12) , clear(s14) , clear(s7) , clear(s10) , clear(s13) , clear(s9) , truck_at(s8) , driver_at(s8) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 100. | crane_loaded(crate3 , s10) , on_floor(crate1 , s4) |
| | crane_at(s14) , clear(s4) , clear(s8) , clear(s2) , clear(s10) , clear(s6) , clear(s16) , clear(s12) , clear(s1) , clear(s13) , truck_at(s3) , driver_at(s3) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |

### C.9.2 Uncontrolled Problem Sample

| | Task |
|---|---|
| 1. | crane_at(s6) , on_floor(crate2 , s13) , clear(s2) |
| | crane_at(s14) , clear(s9) , clear(s7) , clear(s13) , clear(s1) , clear(s6) , clear(s16) , clear(s3) , clear(s8) , clear(s4) , truck_at(s2) , driver_at(s2) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 2. | on_floor(crate1 , s1) |
| | crane_at(s9) , clear(s4) , clear(s7) , clear(s14) , clear(s9) , clear(s3) , clear(s6) , clear(s10) , clear(s2) , clear(s12) , truck_at(s13) , driver_at(s13) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s16) , on_floor(crate1 , s16) , in_truck , truck_unloaded , crane_unloaded |
| 3. | truck_at(s2) , on_floor(crate1 , s8) , on_floor(crate3 , s16) |
| | crane_at(s10) , clear(s9) , clear(s7) , clear(s4) , clear(s6) , clear(s14) , clear(s13) , clear(s12) , clear(s8) , clear(s16) , truck_at(s2) , driver_at(s2) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 4. | driver_at(s9) , stacked(crate2 , shelf6) |
| | crane_at(s6) , clear(s10) , clear(s13) , clear(s8) , clear(s9) , clear(s16) , clear(s1) , clear(s14) , clear(s12) , clear(s7) , truck_at(s6) , driver_at(s6) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 5. | above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s16) , clear(s12) , clear(s14) , clear(s13) , clear(s2) , clear(s7) , clear(s8) , clear(s9) , clear(s3) , truck_at(s1) , driver_at(s1) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 6. | clear(s12) , on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s14) , clear(s1) , clear(s7) , clear(s8) , clear(s12) , clear(s2) , clear(s10) , clear(s13) , clear(s4) , truck_at(s16) , driver_at(s16) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 7. | truck_at(s1) |
| | crane_at(s6) , clear(s7) , clear(s9) , clear(s1) , clear(s8) , clear(s4) , clear(s10) , clear(s6) , clear(s2) , clear(s13) , truck_at(s12) , driver_at(s12) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s16) , on_floor(crate1 , s16) , in_truck , truck_unloaded , crane_unloaded |
| 8. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s14) , clear(s8) , clear(s10) , clear(s7) , clear(s13) , clear(s16) , clear(s6) , clear(s4) , clear(s9) , truck_at(s3) , driver_at(s3) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 9. | at(crate3 , s6) |
| | crane_at(s9) , clear(s3) , clear(s16) , clear(s6) , clear(s10) , clear(s2) , clear(s1) , clear(s14) , clear(s7) , clear(s4) , truck_at(s12) , driver_at(s12) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 10. | at(crate2 , s2) |
| | crane_at(s14) , clear(s7) , clear(s10) , clear(s4) , clear(s16) , clear(s12) , clear(s3) , clear(s13) , clear(s6) , clear(s9) , truck_at(s2) , driver_at(s2) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 11. | truck_at(s1) , at(crate2 , s2) |
| | crane_at(s10) , clear(s1) , clear(s8) , clear(s13) , clear(s10) , clear(s6) , clear(s2) , clear(s9) , clear(s4) , clear(s12) , truck_at(s16) , driver_at(s16) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 12. | driver_at(s1) , at(crate2 , s2) |
| | crane_at(s9) , clear(s7) , clear(s13) , clear(s16) , clear(s14) , clear(s6) , clear(s12) , clear(s8) , clear(s2) , clear(s10) , truck_at(s4) , driver_at(s4) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 13. | at(crate2 , s16) , clear(s12) |
| | crane_at(s9) , clear(s4) , clear(s13) , clear(s6) , clear(s16) , clear(s8) , clear(s7) , clear(s3) , clear(s10) , clear(s2) , truck_at(s14) , driver_at(s14) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 14. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s16) , clear(s8) , clear(s10) , clear(s12) , clear(s6) , clear(s14) , clear(s1) , clear(s3) , clear(s2) , truck_at(s9) , driver_at(s9) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 15. | clear(s4) , at(crate1 , s2) |
| | crane_at(s9) , clear(s10) , clear(s1) , clear(s6) , clear(s12) , clear(s3) , clear(s13) , clear(s7) , clear(s9) , clear(s14) , truck_at(s16) , driver_at(s16) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |
| 16. | above_floor(crate3 , s10) |
| | crane_at(s9) , clear(s14) , clear(s8) , clear(s3) , clear(s12) , clear(s6) , clear(s9) , clear(s7) , clear(s16) , clear(s1) , truck_at(s13) , driver_at(s13) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 17. | crane_loaded(crate3 , s10) , truck_at(s1) |
| | crane_at(s10) , clear(s4) , clear(s12) , clear(s14) , clear(s13) , clear(s9) , clear(s2) , clear(s6) , clear(s8) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 18. | at(crate3 , s6) |
| | crane_at(s9) , clear(s10) , clear(s1) , clear(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s16) , clear(s2) , clear(s8) , truck_at(s9) , driver_at(s9) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 19. | driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s1) , clear(s3) , clear(s6) , clear(s2) , clear(s8) , clear(s13) , clear(s10) , clear(s7) , clear(s14) , truck_at(s4) , driver_at(s4) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 20. | driver_at(s9) , crane_loaded(crate3 , s10) , clear(s2) |
| | crane_at(s9) , clear(s3) , clear(s7) , clear(s14) , clear(s13) , clear(s10) , clear(s9) , clear(s2) , clear(s4) , clear(s12) , truck_at(s16) , driver_at(s16) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 21. | truck_at(s1) , above_floor(crate1 , s10) , at(crate2 , s6) |
| | crane_at(s6) , clear(s16) , clear(s14) , clear(s9) , clear(s8) , clear(s10) , clear(s3) , clear(s7) , clear(s12) , clear(s2) , truck_at(s1) , driver_at(s1) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 22. | on_floor(crate2 , s1) , at(crate3 , s13) , at(crate1 , s6) |
| | crane_at(s14) , clear(s16) , clear(s6) , clear(s13) , clear(s9) , clear(s14) , clear(s2) , clear(s10) , clear(s4) , clear(s3) , truck_at(s1) , driver_at(s1) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 23. | on_floor(crate2 , s16) , crane_loaded(crate1 , s10) , on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s6) , clear(s3) , clear(s7) , clear(s1) , clear(s16) , clear(s8) , clear(s13) , clear(s4) , clear(s10) , truck_at(s14) , driver_at(s14) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 24. | clear(s4) , at(crate3 , s2) , at(crate1 , s6) |
| | crane_at(s9) , clear(s7) , clear(s8) , clear(s3) , clear(s10) , clear(s16) , clear(s4) , clear(s9) , clear(s13) , clear(s1) , truck_at(s6) , driver_at(s6) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 25. | above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s14) , clear(s4) , clear(s7) , clear(s9) , clear(s8) , clear(s13) , clear(s3) , clear(s10) , clear(s16) , truck_at(s2) , driver_at(s2) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 26. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s6) , clear(s9) , clear(s4) , clear(s8) , clear(s10) , clear(s13) , clear(s2) , clear(s16) , clear(s12) , truck_at(s7) , driver_at(s7) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 27. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s8) , clear(s16) , clear(s13) , clear(s2) , clear(s14) , clear(s3) , clear(s12) , clear(s1) , clear(s10) , truck_at(s6) , driver_at(s6) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 28. | on_floor(crate1 , s4) , at(crate2 , s2) , on_floor(crate3 , s16) |
| | crane_at(s9) , clear(s16) , clear(s9) , clear(s4) , clear(s7) , clear(s6) , clear(s3) , clear(s8) , clear(s2) , clear(s12) , truck_at(s10) , driver_at(s10) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s1) , on_floor(crate2 , s1) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 29. | truck_at(s1) |
| | crane_at(s6) , clear(s7) , clear(s10) , clear(s16) , clear(s2) , clear(s12) , clear(s3) , clear(s6) , clear(s1) , clear(s4) , truck_at(s8) , driver_at(s8) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 30. | clear(s12) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s3) , clear(s14) , clear(s13) , clear(s6) , clear(s16) , clear(s10) , clear(s9) , clear(s1) , clear(s2) , truck_at(s4) , driver_at(s4) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 31. | on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s8) , clear(s16) , clear(s12) , clear(s14) , clear(s4) , clear(s6) , clear(s10) , clear(s3) , clear(s1) , truck_at(s7) , driver_at(s7) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 32. | on_floor(crate1 , s1) , stacked(crate2 , shelf6) |
| | crane_at(s14) , clear(s9) , clear(s16) , clear(s1) , clear(s14) , clear(s6) , clear(s2) , clear(s13) , clear(s10) , clear(s7) , truck_at(s12) , driver_at(s12) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 33. | truck_at(s8) , on_floor(crate1 , s8) , on_floor(crate3 , s16) |
| | crane_at(s6) , clear(s14) , clear(s8) , clear(s3) , clear(s16) , clear(s6) , clear(s2) , clear(s12) , clear(s4) , clear(s10) , truck_at(s9) , driver_at(s9) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 34. | driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s10) , clear(s3) , clear(s4) , clear(s2) , clear(s9) , clear(s16) , clear(s14) , clear(s7) , clear(s8) , truck_at(s6) , driver_at(s6) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 35. | driver_on_floor(s9) , at(crate2 , s2) |
| | crane_at(s14) , clear(s2) , clear(s10) , clear(s14) , clear(s4) , clear(s3) , clear(s8) , clear(s16) , clear(s13) , clear(s12) , truck_at(s1) , driver_at(s1) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 36. | driver_at(s2) , above_floor(crate1 , s10) , at(crate2 , s6) |
| | crane_at(s6) , clear(s6) , clear(s8) , clear(s14) , clear(s9) , clear(s1) , clear(s7) , clear(s3) , clear(s4) , clear(s2) , truck_at(s16) , driver_at(s16) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 37. | clear(s9) |
| | crane_at(s9) , clear(s9) , clear(s14) , clear(s8) , clear(s6) , clear(s12) , clear(s7) , clear(s10) , clear(s4) , clear(s16) , truck_at(s3) , driver_at(s3) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 38. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s14) , clear(s3) , clear(s10) , clear(s13) , clear(s6) , clear(s7) , clear(s16) , clear(s4) , clear(s2) , truck_at(s8) , driver_at(s8) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 39. | at(crate2 , s8) , driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s6) , clear(s16) , clear(s10) , clear(s1) , clear(s14) , clear(s9) , clear(s8) , clear(s7) , clear(s13) , truck_at(s4) , driver_at(s4) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 40. | driver_on_floor(s9) , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s9) , clear(s1) , clear(s4) , clear(s12) , clear(s2) , clear(s7) , clear(s16) , clear(s13) , clear(s3) , truck_at(s10) , driver_at(s10) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 41. | at(crate2 , s2) |
| | crane_at(s14) , clear(s12) , clear(s16) , clear(s1) , clear(s8) , clear(s9) , clear(s4) , clear(s13) , clear(s7) , clear(s3) , truck_at(s2) , driver_at(s2) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 42. | driver_at(s8) , on_floor(crate1 , s8) , on_floor(crate3 , s16) |
| | crane_at(s6) , clear(s16) , clear(s9) , clear(s8) , clear(s13) , clear(s7) , clear(s10) , clear(s12) , clear(s3) , clear(s1) , truck_at(s4) , driver_at(s4) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 43. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s12) , clear(s4) , clear(s7) , clear(s2) , clear(s14) , clear(s6) , clear(s13) , clear(s1) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 44. | driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s2) , clear(s14) , clear(s12) , clear(s1) , clear(s4) , clear(s10) , clear(s7) , clear(s9) , clear(s8) , truck_at(s3) , driver_at(s3) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s16) , on_floor(crate2 , s16) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 45. | on_floor(crate1 , s4) , driver_on_floor(s9) , truck_at(s1) |
| | crane_at(s6) , clear(s9) , clear(s10) , clear(s12) , clear(s3) , clear(s8) , clear(s7) , clear(s1) , clear(s4) , clear(s16) , truck_at(s14) , driver_at(s14) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 46. | driver_on_floor(s9) , clear(s12) |
| | crane_at(s6) , clear(s10) , clear(s13) , clear(s9) , clear(s2) , clear(s6) , clear(s8) , clear(s1) , clear(s16) , clear(s3) , truck_at(s14) , driver_at(s14) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 47. | crane_loaded(crate3 , s10) , above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s16) , clear(s4) , clear(s12) , clear(s7) , clear(s14) , clear(s9) , clear(s3) , clear(s13) , clear(s8) , truck_at(s10) , driver_at(s10) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 48. | driver_on_floor(s9) , truck_at(s1) |
| | crane_at(s14) , clear(s7) , clear(s9) , clear(s13) , clear(s8) , clear(s12) , clear(s2) , clear(s1) , clear(s14) , clear(s4) , truck_at(s10) , driver_at(s10) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s16) , on_floor(crate1 , s16) , in_truck , truck_unloaded , crane_unloaded |
| 49. | on_floor(crate3 , s2) , on_floor(crate1 , s1) , crane_at(s9) |
| | crane_at(s9) , clear(s7) , clear(s14) , clear(s2) , clear(s16) , clear(s4) , clear(s13) , clear(s9) , clear(s1) , clear(s6) , truck_at(s12) , driver_at(s12) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 50. | driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s7) , clear(s12) , clear(s16) , clear(s13) , clear(s4) , clear(s8) , clear(s10) , clear(s2) , clear(s1) , truck_at(s3) , driver_at(s3) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 51. | clear(s12) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s1) , clear(s9) , clear(s2) , clear(s3) , clear(s16) , clear(s4) , clear(s13) , clear(s6) , clear(s8) , truck_at(s10) , driver_at(s10) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s12) , on_floor(crate1 , s12) , in_truck , truck_unloaded , crane_unloaded |
| 52. | crane_loaded(crate2 , s9) |
| | crane_at(s10) , clear(s14) , clear(s12) , clear(s1) , clear(s16) , clear(s8) , clear(s13) , clear(s7) , clear(s9) , clear(s6) , truck_at(s2) , driver_at(s2) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 53. | at(crate2 , s2) |
| | crane_at(s9) , clear(s6) , clear(s3) , clear(s9) , clear(s13) , clear(s8) , clear(s7) , clear(s2) , clear(s12) , clear(s4) , truck_at(s16) , driver_at(s16) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 54. | at(crate3 , s16) , at(crate1 , s6) |
| | crane_at(s9) , clear(s12) , clear(s16) , clear(s8) , clear(s13) , clear(s2) , clear(s14) , clear(s4) , clear(s6) , clear(s1) , truck_at(s10) , driver_at(s10) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 55. | at(crate2 , s2) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s10) , clear(s2) , clear(s14) , clear(s9) , clear(s7) , clear(s3) , clear(s16) , truck_at(s4) , driver_at(s4) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 56. | driver_on_floor(s9) , truck_at(s1) |
| | crane_at(s6) , clear(s4) , clear(s12) , clear(s2) , clear(s3) , clear(s16) , clear(s10) , clear(s9) , clear(s1) , clear(s7) , truck_at(s6) , driver_at(s6) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s8) , on_floor(crate1 , s8) , in_truck , truck_unloaded , crane_unloaded |
| 57. | at(crate3 , s9) , clear(s12) |
| | crane_at(s10) , clear(s12) , clear(s3) , clear(s6) , clear(s7) , clear(s13) , clear(s1) , clear(s8) , clear(s14) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 58. | at(crate3 , s16) , driver_at(s7) , above_floor(crate1 , s10) |
| | crane_at(s14) , clear(s12) , clear(s16) , clear(s7) , clear(s14) , clear(s2) , clear(s3) , clear(s8) , clear(s6) , clear(s4) , truck_at(s10) , driver_at(s10) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 59. | at(crate2 , s2) |
| | crane_at(s9) , clear(s6) , clear(s7) , clear(s3) , clear(s1) , clear(s8) , clear(s12) , clear(s14) , clear(s10) , clear(s2) , truck_at(s16) , driver_at(s16) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 60. | above_floor(crate1 , s10) |
| | crane_at(s9) , clear(s13) , clear(s4) , clear(s8) , clear(s9) , clear(s3) , clear(s10) , clear(s2) , clear(s16) , clear(s6) , truck_at(s14) , driver_at(s14) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 61. | driver_on_floor(s9) , above_floor(crate1 , s10) , clear(s12) |
| | crane_at(s10) , clear(s7) , clear(s1) , clear(s4) , clear(s14) , clear(s16) , clear(s8) , clear(s9) , clear(s2) , clear(s6) , truck_at(s12) , driver_at(s12) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 62. | above_floor(crate1 , s10) , on_floor(crate3 , s16) |
| | crane_at(s6) , clear(s7) , clear(s3) , clear(s16) , clear(s10) , clear(s13) , clear(s8) , clear(s9) , clear(s1) , clear(s12) , truck_at(s4) , driver_at(s4) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 63. | at(crate3 , s2) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s7) , clear(s10) , clear(s2) , clear(s9) , clear(s16) , clear(s3) , clear(s8) , truck_at(s6) , driver_at(s6) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |
| 64. | at(crate1 , s6) |
| | crane_at(s6) , clear(s12) , clear(s2) , clear(s8) , clear(s10) , clear(s7) , clear(s1) , clear(s16) , clear(s9) , clear(s13) , truck_at(s6) , driver_at(s6) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 65. | at(crate3 , s3) , in_truck , truck_at(s1) |
| | crane_at(s9) , clear(s10) , clear(s13) , clear(s2) , clear(s16) , clear(s14) , clear(s7) , clear(s12) , clear(s9) , clear(s3) , truck_at(s8) , driver_at(s8) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 66. | driver_at(s12) , crane_loaded(crate2 , s9) |
| | crane_at(s9) , clear(s12) , clear(s14) , clear(s3) , clear(s6) , clear(s10) , clear(s1) , clear(s2) , clear(s7) , clear(s8) , truck_at(s16) , driver_at(s16) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 67. | above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s9) , clear(s10) , clear(s16) , clear(s4) , clear(s14) , clear(s8) , truck_at(s2) , driver_at(s2) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 68. | on_floor(crate3 , s16) , driver_on_floor(s9) , at(crate2 , s2) |
| | crane_at(s9) , clear(s2) , clear(s7) , clear(s13) , clear(s9) , clear(s1) , clear(s6) , clear(s3) , clear(s4) , clear(s16) , truck_at(s8) , driver_at(s8) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 69. | on_floor(crate2 , s3) , on_floor(crate1 , s1) |
| | crane_at(s6) , clear(s6) , clear(s7) , clear(s9) , clear(s8) , clear(s4) , clear(s2) , clear(s12) , clear(s1) , clear(s10) , truck_at(s13) , driver_at(s13) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s14) , on_floor(crate2 , s14) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 70. | crane_loaded(crate1 , s14) , clear(s12) |
| | crane_at(s6) , clear(s14) , clear(s1) , clear(s3) , clear(s13) , clear(s8) , clear(s7) , clear(s6) , clear(s12) , clear(s16) , truck_at(s10) , driver_at(s10) , at(crate3 , s9) , on_floor(crate3 , s9) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 71. | in_truck , above_floor(crate1 , s10) , clear(s12) |
| | crane_at(s9) , clear(s2) , clear(s10) , clear(s13) , clear(s16) , clear(s6) , clear(s9) , clear(s8) , clear(s1) , clear(s3) , truck_at(s12) , driver_at(s12) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 72. | at(crate3 , s6) , driver_on_floor(s2) , truck_at(s1) |
| | crane_at(s6) , clear(s14) , clear(s7) , clear(s12) , clear(s6) , clear(s10) , clear(s1) , clear(s9) , clear(s4) , clear(s16) , truck_at(s8) , driver_at(s8) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 73. | crane_loaded(crate2 , s9) |
| | crane_at(s14) , clear(s6) , clear(s14) , clear(s7) , clear(s12) , clear(s9) , clear(s10) , clear(s13) , clear(s8) , clear(s16) , truck_at(s1) , driver_at(s1) , at(crate3 , s3) , on_floor(crate3 , s3) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s4) , on_floor(crate1 , s4) , in_truck , truck_unloaded , crane_unloaded |
| 74. | truck_at(s1) , on_floor(crate2 , s3) |
| | crane_at(s9) , clear(s7) , clear(s10) , clear(s13) , clear(s16) , clear(s2) , clear(s8) , clear(s6) , clear(s14) , clear(s12) , truck_at(s4) , driver_at(s4) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 75. | at(crate2 , s2) |
| | crane_at(s6) , clear(s2) , clear(s1) , clear(s8) , clear(s6) , clear(s10) , clear(s12) , clear(s4) , clear(s7) , clear(s16) , truck_at(s13) , driver_at(s13) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 76. | truck_at(s14) , at(crate2 , s2) |
| | crane_at(s10) , clear(s12) , clear(s9) , clear(s6) , clear(s2) , clear(s16) , clear(s14) , clear(s13) , clear(s4) , clear(s1) , truck_at(s3) , driver_at(s3) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 77. | at(crate3 , s13) , on_floor(crate1 , s1) , at(crate2 , s2) |
| | crane_at(s9) , clear(s6) , clear(s13) , clear(s4) , clear(s12) , clear(s8) , clear(s1) , clear(s3) , clear(s9) , clear(s16) , truck_at(s7) , driver_at(s7) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 78. | at(crate3 , s9) , crane_loaded(crate2 , s9) |
| | crane_at(s6) , clear(s14) , clear(s16) , clear(s13) , clear(s7) , clear(s1) , clear(s12) , clear(s10) , clear(s8) , clear(s6) , truck_at(s9) , driver_at(s9) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 79. | above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s16) , clear(s13) , clear(s7) , clear(s8) , clear(s9) , clear(s6) , clear(s2) , clear(s3) , clear(s4) , truck_at(s14) , driver_at(s14) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 80. | at(crate2 , s10) , in_truck , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s4) , clear(s10) , clear(s14) , clear(s2) , clear(s7) , clear(s13) , clear(s1) , clear(s9) , clear(s12) , truck_at(s8) , driver_at(s8) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s6) , on_floor(crate2 , s6) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |

| | Task |
|---|---|
| 81. | crane_loaded(crate3 , s10) , above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s14) , clear(s1) , clear(s9) , clear(s7) , clear(s6) , clear(s16) , clear(s13) , clear(s3) , clear(s12) , truck_at(s10) , driver_at(s10) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 82. | driver_on_floor(s9) , above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s9) , clear(s14) , clear(s7) , clear(s6) , clear(s16) , clear(s1) , clear(s8) , clear(s13) , clear(s12) , truck_at(s4) , driver_at(s4) , at(crate3 , s10) , on_floor(crate3 , s10) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 83. | on_floor(crate1 , s4) , driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s9) , clear(s12) , clear(s3) , clear(s9) , clear(s8) , clear(s13) , clear(s14) , clear(s2) , clear(s1) , clear(s16) , truck_at(s4) , driver_at(s4) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s10) , on_floor(crate2 , s10) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 84. | on_floor(crate1 , s1) , at(crate2 , s2) |
| | crane_at(s10) , clear(s13) , clear(s12) , clear(s6) , clear(s7) , clear(s10) , clear(s3) , clear(s14) , clear(s9) , clear(s2) , truck_at(s16) , driver_at(s16) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 85. | crane_loaded(crate1 , s10) , on_floor(crate3 , s6) |
| | crane_at(s14) , clear(s13) , clear(s3) , clear(s10) , clear(s14) , clear(s9) , clear(s12) , clear(s4) , clear(s1) , clear(s16) , truck_at(s7) , driver_at(s7) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 86. | crane_at(s14) , in_truck , on_floor(crate3 , s6) |
| | crane_at(s9) , clear(s8) , clear(s2) , clear(s13) , clear(s10) , clear(s14) , clear(s6) , clear(s12) , clear(s1) , clear(s16) , truck_at(s9) , driver_at(s9) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 87. | stacked(crate2 , shelf6) |
| | crane_at(s6) , clear(s1) , clear(s2) , clear(s7) , clear(s12) , clear(s6) , clear(s14) , clear(s13) , clear(s8) , clear(s10) , truck_at(s3) , driver_at(s3) , at(crate3 , s16) , on_floor(crate3 , s16) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 88. | driver_on_floor(s9) , truck_at(s4) , above_floor(crate1 , s10) |
| | crane_at(s6) , clear(s14) , clear(s4) , clear(s10) , clear(s13) , clear(s2) , clear(s6) , clear(s8) , clear(s1) , clear(s16) , truck_at(s9) , driver_at(s9) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s3) , on_floor(crate2 , s3) , at(crate1 , s7) , on_floor(crate1 , s7) , in_truck , truck_unloaded , crane_unloaded |
| 89. | driver_at(s9) , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s8) , clear(s7) , clear(s13) , clear(s14) , clear(s3) , clear(s10) , clear(s16) , clear(s1) , clear(s6) , truck_at(s12) , driver_at(s12) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s9) , on_floor(crate1 , s9) , in_truck , truck_unloaded , crane_unloaded |
| 90. | at(crate1 , s6) , on_floor(crate3 , s6) |
| | crane_at(s10) , clear(s16) , clear(s1) , clear(s13) , clear(s14) , clear(s10) , clear(s7) , clear(s6) , clear(s4) , clear(s12) , truck_at(s3) , driver_at(s3) , at(crate3 , s8) , on_floor(crate3 , s8) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 91. | driver_on_floor(s1) |
| | crane_at(s10) , clear(s1) , clear(s12) , clear(s2) , clear(s3) , clear(s10) , clear(s6) , clear(s9) , clear(s16) , clear(s13) , truck_at(s7) , driver_at(s7) , at(crate3 , s4) , on_floor(crate3 , s4) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 92. | at(crate3 , s2) |
| | crane_at(s6) , clear(s7) , clear(s1) , clear(s13) , clear(s16) , clear(s9) , clear(s2) , clear(s8) , clear(s12) , clear(s3) , truck_at(s10) , driver_at(s10) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s14) , on_floor(crate1 , s14) , in_truck , truck_unloaded , crane_unloaded |
| 93. | at(crate1 , s6) , above_floor(crate1 , s10) |
| | crane_at(s10) , clear(s4) , clear(s9) , clear(s12) , clear(s7) , clear(s3) , clear(s2) , clear(s14) , clear(s13) , clear(s1) , truck_at(s16) , driver_at(s16) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s8) , on_floor(crate2 , s8) , at(crate1 , s10) , on_floor(crate1 , s10) , in_truck , truck_unloaded , crane_unloaded |
| 94. | in_truck , on_floor(crate3 , s6) |
| | crane_at(s6) , clear(s14) , clear(s8) , clear(s10) , clear(s16) , clear(s6) , clear(s13) , clear(s3) , clear(s12) , clear(s7) , truck_at(s9) , driver_at(s9) , at(crate3 , s2) , on_floor(crate3 , s2) , at(crate2 , s4) , on_floor(crate2 , s4) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 95. | above_floor(crate1 , s10) , clear(s4) , at(crate1 , s6) |
| | crane_at(s14) , clear(s10) , clear(s6) , clear(s8) , clear(s16) , clear(s14) , clear(s4) , clear(s7) , clear(s13) , clear(s3) , truck_at(s9) , driver_at(s9) , at(crate3 , s1) , on_floor(crate3 , s1) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s2) , on_floor(crate1 , s2) , in_truck , truck_unloaded , crane_unloaded |
| 96. | at(crate2 , s2) |
| | crane_at(s14) , clear(s14) , clear(s16) , clear(s13) , clear(s3) , clear(s2) , clear(s1) , clear(s10) , clear(s4) , clear(s7) , truck_at(s8) , driver_at(s8) , at(crate3 , s12) , on_floor(crate3 , s12) , at(crate2 , s9) , on_floor(crate2 , s9) , at(crate1 , s6) , on_floor(crate1 , s6) , in_truck , truck_unloaded , crane_unloaded |
| 97. | above_floor(crate1 , s10) |
| | crane_at(s14) , clear(s4) , clear(s9) , clear(s8) , clear(s13) , clear(s16) , clear(s14) , clear(s3) , clear(s6) , clear(s2) , truck_at(s10) , driver_at(s10) , at(crate3 , s7) , on_floor(crate3 , s7) , at(crate2 , s12) , on_floor(crate2 , s12) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 98. | on_floor(crate1 , s4) , driver_on_floor(s9) , at(crate2 , s2) |
| | crane_at(s9) , clear(s4) , clear(s10) , clear(s16) , clear(s12) , clear(s6) , clear(s3) , clear(s9) , clear(s2) , clear(s8) , truck_at(s14) , driver_at(s14) , at(crate3 , s13) , on_floor(crate3 , s13) , at(crate2 , s7) , on_floor(crate2 , s7) , at(crate1 , s1) , on_floor(crate1 , s1) , in_truck , truck_unloaded , crane_unloaded |
| 99. | truck_at(s4) , above_floor(crate2 , s6) |
| | crane_at(s9) , clear(s16) , clear(s4) , clear(s10) , clear(s9) , clear(s1) , clear(s6) , clear(s12) , clear(s8) , clear(s7) , truck_at(s2) , driver_at(s2) , at(crate3 , s14) , on_floor(crate3 , s14) , at(crate2 , s13) , on_floor(crate2 , s13) , at(crate1 , s3) , on_floor(crate1 , s3) , in_truck , truck_unloaded , crane_unloaded |
| 100. | at(crate2 , s2) |
| | crane_at(s14) , clear(s1) , clear(s3) , clear(s4) , clear(s10) , clear(s14) , clear(s9) , clear(s7) , clear(s16) , clear(s12) , truck_at(s8) , driver_at(s8) , at(crate3 , s6) , on_floor(crate3 , s6) , at(crate2 , s2) , on_floor(crate2 , s2) , at(crate1 , s13) , on_floor(crate1 , s13) , in_truck , truck_unloaded , crane_unloaded |

# Bibliography


[1] S. Amarel. On Representations of Problems of Reasoning about Actions. In D. Mitchie, editor, *Machine Intelligence*, volume 3, pages 131 – 171. Edinburgh University Press, 1968.

[2] F. Bacchus and Q. Yang. The Downward Refinement Property. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[3] Bäckström and Klein. Parallel Non-Binary Planning in Polynomial Time. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[4] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Representation and Compilation Laboratory. Technical Report RR-93-08, The German Research Centre for Artificial Intelligence, 1993.

[5] T. Bylander. A Simple Model of Knowledge Compilation. *IEEE Expert*, pages 73 – 74, 1991.

[6] T. Bylander. Complexity Results for Planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[7] T. Bylander. Complexity Results for Extended Planning. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, 1992.

[8] G. Carbonnel, C. A. Knoblock, and S. Minton. PRODIGY: An integrated architecture for planning and Learning. In K. Van Lehn, editor, *Architectures for Intelligence*. Morgan Kaufman, 1991.

[9] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32, 1987.

[10] S. V. Chenoweth. On the NP-Hardness of Blocks World. In *Ninth National Conference on Artificial Intelligence*, 1991.

[11] J. Christensen and A. Grove. A Formal Model for Classical Planning. In *Ninth National Conference on Artificial Intelligence*, 1991.

[12] W. Cohen. Generalizing Number and learning from multiple explanations in Explanation Based Learning. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[13] K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49 – 86, 1991.

[14] C. Dawson and L. Siklóssy. The Role of Preprocessing in Problem Solving Systems. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

[15] T. Dean and M. Boddy. Reasoning about Partially Ordered Events. *Artificial Intelligence*, 36(3):375 – 399, 1987.

[16] T. Dean, R. J. Firby, and D. Miller. Hierarchical planning involving deadlines, travel time and resources. *Computational Intelligence*, 4(4):381 – 398, 1988.

[17] G. F. DeJong. Some thoughts on the present and future of explanation-based learning. In *ECAI-88*, 1988.

[18] R. Doyle. Constructing and Refining Causal Explanations from an Inconsistent Domain Theory. In *Fifth National Conference on Artificial Intelligence*, 1986.

[19] M. Drummond and K. Currie. Goal Ordering in Partially Ordered Plans. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[20] O. Etzioni. Why PRODIGY/EBL Works. In *Eighth National Conference on Artificial Intelligence*, 1990.

[21] O. Etzioni. STATIC: A Problem – Space Compiler for PRODIGY. In *Ninth National Conference on Artificial Intelligence*, 1991.

[22] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2, 1971.

[23] E. Fink and Q. Yang. Automatically Abstracting the Effects of Operators. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, 1992.

[24] D. Fisher, D. Subramanian, and P. Tadepalli. An Overview of Current Research on Knowledge Compilation and Speedup Learning. In *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*, 1992.

[25] N. S. Flann. Knowledge Compilation in Counter-Planning Using Abstract Reverse Enumeration. In *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*, 1992.

[26] Brassard G. and Bratley P. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[27] M. Ginsberg. Universal Planning: An (Almost) Universally Bad Idea. *AI Magazine*, 3(4), 1989.

[28] N. Gupta and D. S. Nau. Complexity Results for Blocks-World Planning. In *Ninth National Conference on Artificial Intelligence*, 1991.

[29] P. J. Hayes. The Frame Problem and related problems in artificial intelligence. In A. Elithorn and D. Jones, editors, *Artificial and Human Thinking*, pages 45 – 59. San Francisco: Jossey – Basse, 1973.

[30] J. Hobbs. Granularity. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[31] G. Iba. Learning by discovering macros in puzzle solving. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[32] K. B. Irani and J. Cheng. Subgoal Ordering and Goal Augmentation for Heuristic Problem Solving. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[33] K. B. Irani and J. Cheng. Ordering Problem Subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[34] D. Joslin and J. Roach. A Theoretical Analysis of Conjunctive – Goal Problems. *Artificial Intelligence*, 41:97 – 106, 1989.

[35] R. M. Keller. Applying Knowledge Compilation Techniques to Model – Based Reasoning. *IEEE Expert*, pages 82 – 87, 1991.

[36] C. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[37] C. A. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *Eighth National Conference on Artificial Intelligence*, 1990.

[38] C. A. Knoblock. An Analysis of ABSTRIPS. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, 1992.

[39] R. E. Korf. Toward a Model of Representation Change. *Artificial Intelligence*, 14:41 – 78, 1980.

[40] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1), 1985.

[41] R. E. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33, 1987.

[42] V. Lifschitz. On the Semantics of STRIPS. In *Proc. 1986 workshop: Reasoning about actions and plans*, 1986.

[43] M. R. Lowry. Algorithm Synthesis through Problem Reformulation. In *Sixth National Conference on Artificial Intelligence*, 1987.

[44] M. R. Lowry. The Abstraction/Implementation Model of Problem Reformulation. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[45] T. L. McCluskey. Deriving a Correct Logic Program from the Formal Specification of a Non-Linear Planner. In Z. W. Ras and L. Saitta, editors, *Methodologies for Intelligent Systems, 3.* Elsevier Scientific, 1988.

[46] T. L. McCluskey. *Experience Driven Heuristic Acquisition in General Problem Solvers.* PhD thesis, Department of Computer Science, The City University, 1988.

[47] T. L. McCluskey. The FM User Guide. Technical Report TCU/CS/88/30, Department of Computer Science, The City University, 1988.

[48] T. L. McCluskey and J. M. Porteous. Model Adaption: an approach to tackling the efficiency proble m in general planning systems. In *Proceedings of the 11th Workshop of the UK Planning Special Interest Group*, 1992.

[49] T. L. McCluskey and J. M. Porteous. Research into the Possibility of Automated Planning for En Route Air Traffic Control. *IEE Colloquium on: Software in Air Traffic Control Systems – The Future*, 1992.

[50] T. L. McCluskey and J. M. Porteous. Some Empirical Results in Planning Speedup Using Combinations of On-line Adaption Techniques. In *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*, 1992.

[51] S. Minton. Selectively Generalizing Plans for Problem Solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 596 – 599, 1985.

[52] S. Minton. Quantitative Results concerning the utility of Explanation-Based Learning. In *Seventh National Conference on Artificial Intelligence*, pages 564 – 569, 1988.

[53] S. Minton. Quantitative Results concerning the Utility of Explanation Based Learning. *Artificial Intelligence*, 42:363 – 392, 1990.

[54] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gill. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40:63 – 118, 1989.

[55] J. Mostow and A. E. Prieditis. Discovering Admissible Heuristics by Abstracting and Optimizing: A Transformational Approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[56] D. M. Neves and J. R. Anderson. Knowledge Compilation: Mechanisms for the Automization of Cognitive Skills. In J. R. Anderson, editor, *Cognitive Skills and their Acquisition.* Erlbaum, Hillsdale, N.J., 1981.

[57] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87 – 127, 1982.

[58] A. Newell and H. A. Simon. GPS: a Program that Simulates Human Thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought.* McGraw-Hill, New York, 1963.

[59] D. A. Pearce. The Induction of Fault Diagnosis Systems from Qualitative Models. In *Seventh National Conference on Artificial Intelligence*, 1988.

[60] R. Holte and R. Zimmer and A. MacDonald. A Study of the Representation – Dependency of Abstraction Techniques. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, 1993.

[61] D. Ruby and D. Kibler. Learning Subgoal Sequences for Planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[62] E. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5, 1974.

[63] E. D. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975.

[64] A. Segre, C. Elkan, and A. Russell. A Critical Look at Evaluations of EBL. *Artificial Intelligence*, 6:183 – 195, 1991.

[65] J. W. Shavlik. Acquiring Recursive and Iterative Concepts with Explanation Based Learning. *Machine Learning*, pages 39 – 70, 1990.

[66] J. W. Shavlik and G. F. DeJong. An Explanation-Based Approach to Generalising Number. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[67] P. Shell and J. G. Carbonnel. Towards a General Framework for Composing Disjunctive and Iterative Macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[68] M. J. Stefik. Planning with Constraints. *Artificial Intelligence*, 16:111 – 140, 1981.

[69] D. Subramanian and R. Feldman. The Utility of EBL in Recursive Domain Theories. In *Eighth National Conference on Artificial Intelligence*, 1990.

[70] G. A. Sussman. A Computational Model of Skill Acquisition. Technical Report A.I. Lab. Memo no. AI – TR – 297, M. I. T., 1973.

[71] P. Tadepalli and R. Isukapalli. Learning Plan Knowledge from Simulators. In *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*, 1992.

[72] A. Tate. Project Planning Using a Hierarchical Non-linear Planner. Technical Report Department of Artificial Intelligence memo 75, University of Edinburgh, 1977.

[73] A. Tate, J. Hendler, and M. Drummond. A Review of AI Planning Techniques. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 26 – 49. Morgan Kaufman, 1990.

[74] C. Tong. The Nature and Significance of Knowledge Compilation. *IEEE Expert*, pages 82 – 87, April 1991.

[75] S. Vere. Temporal Scope of Assertions and Window Cutoff. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[76] D. Wilkins. *Practical Planning: Extending the Classical A.I. Paradigm.* Addison-Wesley, 1988.

[77] Q. Yang, D. S. Nau, and J. Hendler. Merging Separately Generated Plans with Restricted Interactions. *Computational Intelligence*, 8:648 – 676, 1990.