Proceedings of the

29th Workshop of the UK Special Interest Group on Planning and Scheduling

PlanSIG2011

University of Huddersfield, UK

8th - 9th December, 2011

Edited By:

Andrew Crampton Diane Kitchin Lee McCluskey

ISSN: 1368-5708

Preface

These proceedings collect the papers accepted for presentation at the 29th Workshop of the UK Planning & Scheduling Special Interest Group (PlanSIG-11).

The PlanSIG workshop is a yearly forum where academic staff, industrialists, and research students can meet and discuss current issues in an informal setting. We especially aim to bring together researchers attacking different aspects of planning and scheduling problems, and to introduce new researchers to the community.

PlanSIG has occasionally been hosted outside the UK: for example last year the workshop was held in Italy, and in 2007 it was held in the Czech Republic. This year it returns to the UK, being held at the University of Huddersfield, December 8-9th, 2011. The proceedings contain 11 long papers and 4 short papers with authors from at least 8 different countries. Each submitted paper was reviewed by either two or three reviewers chosen from an international program committee.

Many people were involved in the organization of the workshop, and we would like to thank, in particular, all members of the programme committee and the authors. Special thanks go to Jane Merrington for the website and front cover, and Gwen Wood for her work in stepping in where Easychair ends - putting the proceedings together ready for printing.

Andrew Crampton, Diane Kitchin, Leo McCluskey (Workshop Chairs)

Conference Organisation

Programme Chairpersons

Thomas Leo McCluskey	University of Huddersfield
Diane Kitchin	University of Huddersfield
Andrew Crampton	University of Huddersfield

International Programme Committee

Ruth Aylett
Roman Barták
Ken Brown
Amanda Coles
Andrew Crampton
Juan Fernández-Olivares
Maria Fox
Simone Fratini
Antonio Garrido
Diane Kitchin
John Levine
Derek Long
Thomas Leo McCluskey
Barry O'Sullivan
Julie Porteous
Rong Qu
Andrew Tuson
Tiago Stegun Vaquero
Gerhard Wickler

Table of Contents

When Planning Should Be Easy: On Solving Cumulative Planning Problems

Roman Barták*, Filip Dvořák*, Jakub Gemrot*, Cyril Brom*, Daniel Toropila*†

*Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Prague 1, Czech Republic

†Charles University in Prague, Computer Science Center
Ovocný trh 5, 116 36 Prague 1, Czech Republic

{roman.bartak, filip.dvorak, jakub.gemrot, cyril.brom, daniel.toropila}@mff.cuni.cz

Abstract

In recent years planning techniques achieved a huge improvement mainly due to International Planning Competition (IPC). However, exploitation of the same techniques in practice is still limited which raises the question whether the planning domains in IPC are practically relevant. In this paper we look at planning domains that appear in computer games, especially when modeling intelligent virtual agents. Some of these domains contain only actions with no negative effects and are thus treated as easy from the planning perspective. We propose two new techniques to solve the problems in these planning domains, a heuristic search algorithm ANA* and a constraint-based planner RelaxPlan, and we compare them with the state-of-the-art planners that were successful in IPC using planning domains motivated by computer games.

Introduction

Though certain planning problems are assumed by the planning community to be easy for solving, these problems still appear in many practical applications. Hence it is useful to have efficient solving approaches for such "easy" problems. In this paper we will look at planning problems that appear in some computer games. These problems can be characterized as cumulative planning problems where actions have no negative effects. Finding a plan solving such a problem can indeed be done in polynomial time (Blum and Furst 1997) but it is more complicated to find the optimal plan (with the smallest sum of action costs).

There has been a huge progress in developing automated planners in recent years so the hope was that the best planners should be able to solve the above mentioned "easy" problems. However, our initial experimental evaluation showed that this is not the case. Hence we decided to develop specific planners for solving the cumulative planning problems to demonstrate that the planning technology is ready to solve such problems. The first planner we have developed belongs to the category of heuristic search planners, which is one of the most successful approaches to automated planning in recent years, at least, if we measure the success via the results of International Planning Competition (IPC). The second planner exploits constraint satisfaction technology, which is not really very successful in IPC (compared to the related SAT technology). This planner was originally designed just for comparison with the heuristic search planner but as we will see later, it performs quite well.

The paper is organized as follows. We will first give the motivation for studying cumulative planning problems originated in computer games. Then we will describe two approaches for solving the planning problem till optimality, namely a heuristic search algorithm ANA* and a constraint-based planner RelaxPlan. Finally, we will experimentally compare these two solving approaches with three classical planners: LAMA 2011 (Richter et al. 2011), SelMax (Domshlak et al. 2011), and Fast Downward: Stone Soup 1 (FDSS1) (Helmert et al. 2011).

Motivation

It is known that planners can be useful for extending capabilities of intelligent virtual agents (IVAs). The classical example of successfully used planning techniques is the videogame F.E.A.R. (Orkin 2006). F.E.A.R. can be described as a 3D first-person-shooter game (FPS), where the player is supposed to fight with various IVAs along his way through the virtual environment as the storyline is pushing him forward. Behaviors for IVAs from FPS games are usually created using some reactive technique, such as hierarchical finite state machines (hFSM) or behavior trees (e.g., Lau 2008; Champandard 2008), but this is not the case in F.E.A.R. F.E.A.R. IVAs' reactive behavior is complemented with STRIPS-like planning system that allows IVAs to find short-term plans satisfying IVAs' predefined goals. This system is then able to solve situations such as "the player has barricaded the door" with the plan "jump on the box and jump through the window" if such an opportunity exists within the environment. The key difference between the reactive IVAs and F.E.A.R. IVAs lies in decoupling of IVAs' goals and actions. Once

done, it is no longer necessary to code the "jump on the box and jump through the window" plan explicitly but only to annotate the environment allowing the F.E.A.R. planning system to find such plans automatically. This approach simplifies the design of IVAs as well as makes IVAs more human-like.

Another motivation for using planners in games lies within the validation of game scenarios. Games played from the first or third person perspective, e.g., Hitman, Metro 2033, are usually split into multiple stand-alone scenarios that are played in a fixed sequence (Figure 1). Each scenario can usually be finished in several ways, in other words, there are multiple plans the player can adopt to fulfill the scenario. As game environments are getting increasingly more complex, designers may fail to realize that the scenario can be finished with simple plans, which is a thing to avoid in game design, as players may bypass intended storyline, interesting content or game events. Therefore, it is preferable to model every scenario as a problem domain that describes the environment as well as available player actions and then ask for "all" possible ways how to solve the scenario. The resulting plans can be reviewed and if an unintended solution is discovered, the designer can alter the environment to prevent adoption of such a plan. The described approach was used for instance in the context of Hitman (Pizzi et al. 2008).

Some games (as described above) are sequential in terms that the state of the virtual environment does not (usually) reoccur or the reoccurrence does not bring anything new for the player and therefore it can be ignored. This means that the vast majority of player actions can be treated as irreversible, which allows modeling possible negative effects (and negative preconditions) of actions as positive effects (and positive preconditions) by creating new atoms prefixed with "not_". This observation greatly simplifies planning problems as it brings the complexity of search for a satisficing solution from (possibly) EXPSPACE to PTIME, where planning algorithms are becoming practically feasible. Briefly speaking, we can see the problem as a planning problem where actions have only "add" effects and no "delete" effects.

Solving Cumulative Planning Problems

As described in the previous section, we need to solve cumulative planning problems where actions have only positive effects. We use the classical propositional representation of planning problems so it means that no atom is ever deleted by executing the plan. These problems are known to be solvable in polynomial time (Blum and Furst 1997). However, we attempt to find the shortest possible plan which is a problem that is NP-hard. We can quickly derive the NP-hardness by subsuming the well-known Set Cover Problem. Formally, given a universe U and a family S of subsets of U, a cover is a subfamily $C \subseteq S$ of sets whose union is U. The input is a pair (U,S) and the task is to find a minimal set covering.

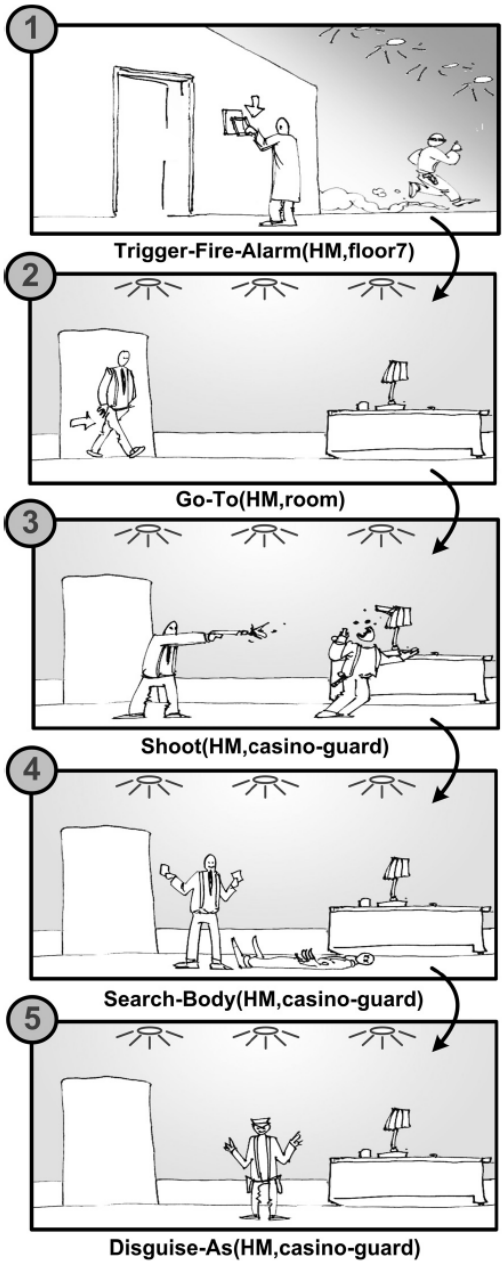


Figure 1. Example of plan in games (from (Pizzi et al. 2008)).

Proof sketch. We shall show that we can solve the set cover problem by translating it to the cumulative planning problem in polynomial time.

For a given input (U, S), we define the problem atoms to be the elements of the universe U. For each set X in S we create an action with no preconditions and with the positive effects to be the elements of X. We define the initial planning state to be empty and the goal state to contain all the atoms. Obviously, the translation can be done in linear time in the size of input.

Since the actions in the plan directly correspond to the sets chosen for the cover, finding the shortest plan now also solves the original set cover problem. ■

Obviously, when solving the cumulative planning problem, the decisions to be done are only whether or not a given action is part of the plan. Because there are no negative effects, there is no reason to include the same action more times in the plan. When the actions are known we still need to verify if they form a plan, that is, to find where in the plan (which layer) the action is located so its preconditions are satisfied.

Though the assumed planning problems should be easier to solve than most problems from IPC, the initial experiments with the winning general planners from IPC showed that these planners have some difficulty with solving this type of problems. Hence we decided to implement specific planners for solving the cumulative planning problems to demonstrate that planning techniques can indeed be used to solve even large-scale planning problems in computer games. Our initial attempt consists of two different approaches, one based on heuristic search with state-of-the-art planning heuristics and the other one based on relatively simple constraint model.

Heuristic Search – ANA*

In classical planning, heuristic search is a common and successful approach to planning. For example, the winners of the latest IPC 2011, Fast Downward Stone Soup-1 (Helmert et al. 2011) and LAMA 2011 (Richter et al. 2011), are both heuristic search planners. The key components of a heuristic planner are the search algorithm and the heuristic estimators.

Heuristics. In our approach for solving the cumulative planning problems we take an advantage of significant research effort in the area of developing heuristics for classical planning. In fact, we can directly incorporate admissible heuristics that operate upon the classical planning problem by first omitting the delete effects and then estimating the solution distance in the delete-relaxed planning problem. The family of such heuristics is known as the delete-relaxation heuristics (Helmert and Domshlak 2009). For our purpose we have chosen the current state-of-the-art heuristic among them, $h^{LM\text{-}cut}$ (Bonet and Helmert 2010). Due to its technical complexity, we do not describe the heuristic here, but for the purpose of this paper we assume that we have an admissible heuristic for the cumulative planning problem that can be computed in polynomial time.

Search algorithm. A* is a well-known search algorithm. It searches from the initial state by always expanding the most promising state. The most promising state is the one with the lowest expected solution cost, which we define as $f(s) = g(s) + h(s)$, where $g(s)$ is the partial cost that was already used (e.g. a path already travelled or the actions already inserted into the plan) and $h(s)$ is the estimated cost that needs to be spent before the solution is reached. We say that h is admissible, if it does not overestimate the cost. The key effect of using an admissible heuristic h is the optimality guarantee for the A* algorithm.

Since in practical problems the optimal solutions tend to be computationally expensive to find, we often relax the optimality requirement. The most common way is to add a *weight* of the heuristic. Such extension of A* is called weighted-A* and the solution cost is evaluated as $f(s) = g(s) + w*h(s)$. The usual approach is to start with a larger weight and once a solution is found reduce the weight by some amount. This approach gives worse solutions quickly (better any solution than none) and gradually improves them with time. However choosing the sequence of weights is generally hard and unpredictable. The issue has been recently addressed by the Anytime Non-parametric A* algorithm (van der Berg et al. 2011).

ANA* is an A*-based algorithm that subsumes both greedy search for the first solution and continual improvement of the solutions while retaining the optimality guarantee (given an admissible heuristic). The key difference is that instead of expanding the state with the minimal f(s), it expands the state with maximal $e(s) = (G - g(s))/h(s)$, where G is the cost of the current best known solution. Since the value of e(s) is equal to the maximal value of the weight that would be given in weighted-A*, such that $f(s) < G$, continually expanding the state with the maximal e(s) corresponds to the greediest possible search to improve the current best solution.

Because of the properties of the ANA* algorithm, we have adapted it for the cumulative planning.

```
solve^ana(s_0, Ac, Goal)
  if not goal_reachable(s_0, Ac, Goal) then return ∅
  landmark_actions ← find(s_0, Ac, Goal)
  plan ← ∅; G ← |Ac|
  op ← {s_0} // open queue
  cl ← ∅ // closed list
  while op ≠ ∅ & not arbitrary_break do
    s ← best^ana(op); op ← op \ {s}
    h(s) ← h^LM-cut(s)
    cl ← cl ∪ {s}
    G ← min(G, g(s) + |relevant_actions(s, Actions)|)
    foreach a ∈ applicable_relevant_actions(s) do
      next ← π(s,a,landmark_actions)
      h(next) ← h(s) - cost(all_applied_actions)
      if is_solution(next, Goal) then
        plan ← get_plan(next)
      else if not (next ∈ op & g(next) ≥ g(op.next)) and
              not (next ∈ cl & g(next) ≥ g(cl.next)) then
        op ← op ∪ {next}
      end if
    end for
  end while
  return plan
```

Solver. Since the cumulative problem has some unique properties that differ from the classical planning and that we can exploit, we provide our algorithm with the description of the important steps. The input of the algorithm is the initial state, the set of actions and the set of goal atoms. The output of the algorithm is the shortest plan found in the given time, or an empty set, if the algorithm did not find any plan. The algorithm is anytime; it can be terminated arbitrarily by breaking the main while-loop, in which case it returns the best complete plan it discovered.

The initial step is a goal reachability check that decides reachability of the goal atoms in polynomial time.

The next step of the algorithm is finding all landmark actions. Those are the actions that must occur in every valid plan, which is a feature that can be verified in polynomial time by checking the existence of a solution for the problem that contains all the actions except for the concerned action we are checking.

We initialize the *open* queue with the initial state, leave the closed list empty and set the value of the best known solution to be the total number of actions; since the goal is reachable, a plan that contains all the actions is a trivial solution. The main while-loop operates until all the states in the open queue have been explored or filtered out. Note that our queue automatically removes the states that cannot provably improve the best-known solution ($g(s) + h(s) \geq G$). In the first step of the iteration we find and remove the best state from the open queue according to the ANA* evaluation of states. At this point we find the heuristic value for the state according to $h^{LM\text{-}cut}$, add it to the closed list and improve the upper bound for the best solution. The *relevant action* is an action that can be further added into the plan; in other words, it is an action whose effects were not yet achieved and at least one of its effects is either an unachieved goal or a precondition of another relevant action. Consequently, we can see that we can construct a plan by adding all relevant actions into the current plan, which gives us potentially a new reduction of the upper bound for the cost of the best solution.

The inner foreach-cycle expands the current state by the application of all the actions that are both relevant and applicable (their preconditions are satisfied in the current state). The first step of the cycle applies the transition function π, which is slightly different from the usual transition function; we first apply the chosen action a, then we try to apply the largest number of landmark actions we can. This way we exploit the fact that since a landmark action must be in every plan exactly once, it does not matter when we apply it; hence we can apply it the first time it is possible. Since computing $h^{LM\text{-}cut}$ is expensive, in this step we calculate the heuristic value for the new state from the value of the parent state by reducing the heuristic value of the parent state by the costs of all the actions that were applied in the previous step. Lucky we are, this does not affect admissibility, since in cumulative planning the application of an action cannot increase the heuristic value as it can in classical planning (by the application of an action we only increase the number of covered atoms, but we do not lose any). The concept of using a heuristic value of the parent state is known as deferred evaluation (Richter and Helmert 2009). Finally, if we have reached the goal, we record the plan, otherwise we add the state into the open queue unless there exists either the same state in the open queue with the same or better cost or the same state is in the closed list with the same or better cost.

Since we use an admissible heuristic, the algorithm is complete; we never discard a state, unless we have either the same state with a less costly partial plan or a complete plan whose cost is lower than the admissible estimate of the state. Further, all the actions applicable for each state are systematically explored. Therefore, the algorithm eventually finds one of the optimal solutions.

The algorithm is sound, which again comes from the admissibility of the used heuristic and the systematical exploration of all states that are pruned only once their lower bound exceeds the global upper bound.

Constraint-based Planner – RelaxPlan

Constraint-based planners are using the idea of translating the planning problem to a constraint satisfaction problem. As the plan length is unknown in advance this translation is usually done in steps where in the *i*-th step the constraint model describes the problem of finding a plan of length *i* (Kautz and Selman 1992). This incremental approach is not necessary for cumulative planning problems where we are "only" selecting actions to add to the plan. Because we know all the actions and atoms in advance (due to the grounded representation of the problem), we can encode the whole cumulative planning problem as a single constraint satisfaction problem. Note that in some sense this is similar to the original constraint model in CPT planner (Vidal and Geffner 2004) that also assumed each action to appear at most once in the plan. Two components need to be specified: the constraint model and the search strategy.

Constraint Model. Recall that designing a constraint model means deciding about variables describing the problem, for each variable deciding its domain (a set of possible values), and finally deciding the constraints that restrict the values to be assigned to the variables.

In a planning problem we have two types of objects, atoms (predicates) and actions. We need to decide which actions will be in the plan and what the positions (levels) of actions will be in the plan. Also, by selecting the actions we are also deciding which atoms will become true and when (in which position in the plan). Obviously, an action can be in the plan only if all atoms from its precondition become true at the levels before the level of the action. Similarly, an atom will become true only if some action having this atom among its effects is in the plan. Then the atom becomes true at the level of the action. Hence the levels define the partial ordering of actions in the plan.

The above observations lead to the following constraint model. For each action a we introduce a variable $ActLevel_a$ describing the position of the action in the plan. Similarly, for each atom (predicate) p we introduce $PredLevel_p$. If we

have m actions and n atoms in the problem then we set the domains for the *Level* variables to $\{0,\ldots,\min(m,n)\}$. Obviously, there must be at least one action at each level and at least one atom must become true at each level (if no new atom becomes true at certain level l then the actions from the next level can actually be processed at level l too). Note that we allow parallel plans so it is possible to have more than one action at a given level. For atoms we also introduce variables specifying the action that makes the particular atom p true: $PredAction_p$. The domain of this variable contains identifications of actions that have atom p among their effects. This is similar to the model from (Do and Kambhampati 2000). For the atoms that are true in the initial state we set $PredLevel_p = 0$ and $PredAction_p = 0$. Basically, there are two groups of constraints connecting actions with atoms:

$$ActLevel_a > \max\{ PredLevel_p \mid p \in \text{precond}(a)\}$$

$$PredLevel_p = ActLevel_{PredAction_a}$$

The reason why we allow action a to be strictly after the level when the precondition holds is because we may decide not to include the action in the plan. In such a case, the action will be part of certain level after the goal level. Assume that G is a set of goal atoms. We use auxiliary variable $GoalLevel = \max\{ PredLevel_p \mid p \in G\}$. Then for action a we also use auxiliary Boolean variable B_a indicating whether a is in the plan or not:

$$B_a = 0 \Leftrightarrow GoalLevel < ActLevel_a.$$

These Boolean variables define the objective function to be minimized (it is easy to modify this objective function to include the cost of actions):

$$Obj = \Sigma_a B_a.$$

Search Strategy. We applied the backward planning approach to find the plan. It means that we start with the set G of goal atom(s) and for each goal atom in this set we try to find an action having this atom among its effects (this action determines the atom). When this action is decided, its preconditions are added to the set of goal atoms and the process is repeated until the set of goal atoms becomes empty. Let us describe this process in more details.

We have a set *Goals* of goal atoms. Initially, it contains the goal atoms G from the problem specification. First, we filter out the atoms for which the action was already decided in the search process[1]. If the set *Goals* becomes empty then we are done. Otherwise we select one goal atom using the first-fail principle. In particular, we prefer atoms with the smallest number of possible determining actions (the domain of variable *PredAction* is the smallest one). In case of tie, we prefer atoms that appear later in the plan (the minimal value in the domain of variable *PredLevel* is maximal among all the atoms). Formally, we select the goal atom:

$$\text{argmin}\{ (s,-k) \mid s = \text{size}(PredAction_i), k = \min(PredLevel_i), i \in Goals\},$$

where size(X) is the size of the domain of X and min(X) is the minimal value in the domain of X. Goal selection corresponds to variable selection in a classical CSP.

When the goal atom p is selected we need to decide the action determining this goal (value selection in a CSP). Basically, it means instantiating the variable $PredAction_p$. We prefer actions that are already decided to be in the plan over the actions that are not-yet decided. In case of tie, we prefer actions that may appear earlier in the plan, that is, actions with the smallest minimal value in the domain of variable $ActLevel_a$. Finally, in case of tie, we prefer actions with the smallest number of preconditions. When action a is selected it is assigned to the variable $PredAction_p$ and the atoms from the precondition of a are added to the set of goal atoms. As there may be more actions that can give atom p this step introduces a choice point. In case of backtracking and before assigning another action to $PredAction_p$ we post a constraint $PredLevel_p < ActLevel_a$ to ensure that action a will not give predicate p in the alternative search branch (the option where a gives p has already been explored).

The search procedure can be formally described as follows:

```
solve(Goals)
  Goals ← filterGoal(Goals)
  if Goals = ∅ then return true
  g ← selectGoal(Goals)
  Actions ← domain(PredAction_g)
  while Actions ≠ ∅ do
    a ← selectAction(Actions)
    PredAction_g ← a
    if solve(Goals \ {g} ∪ precond(a)) then return true
    un-assign PredAction_g
    Actions ← Actions \ {a}
    PredLevel_p < ActLevel_a                    // C1
  end while
return fail {remove constraints C1}
```

The reader may notice that we instantiate only the variables *PredAction*. This is because the filtering of inequality constraints "<" is strong enough to discover infeasibilities such as $A < B < C < A$, that is a cycle of actions. Hence if we decide which actions are giving which predicates we can guarantee that there exists some allocation of actions to levels that forms a parallel plan.

Note finally that optimization is realized using the standard branch-and-bound approach. In particular, when procedure Solve finishes we set $Obj \leftarrow \min(Obj)$ which moves (via constraint propagation) all non-used actions after *GoalLevel*, that is, outside the plan. The value of *Obj*

[1] Note that it is not enough if the corresponding variable *PredAction* is singleton (which may happen by filtering out other actions during constraint propagation). We need to select the action for the predicate explicitly during search to ensure that action preconditions become eventually part of the sub-goal and some actions are selected for them.

is then used as a bound when continuing search and looking for a solution with a smaller value of *Obj*.

Experimental Results

To compare the presented planning techniques we generated artificial planning problems modeling traditional planning problems from computer games as discussed in the introduction. The problems can be characterized by four parameters (n, m, i, j) that are used as follows. We generate a planning graph with n action layers where each layer contains 1 to m actions (equal random distribution) and each action has i positive preconditions – randomly selected atoms from the previous state layers – and j positive effects (randomly generated atoms including those from previous state layers). Only the actions in the first layer contain a single precondition – a special atom *start* that forms the initial state. The last action layer contains at least one action that has a single effect *goal*. The task is to select the smallest number of actions that form a plan achieving the *goal* atom. Figure 2 shows the structure of the used planning graph. Though this type of domains seems very restricted; it suffices to model many planning problems appearing in computer games.

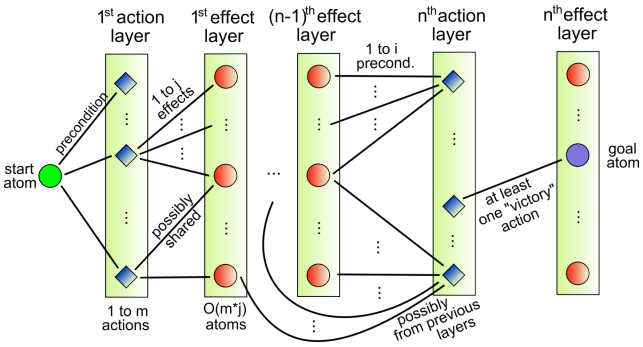


Figure 2. Planning graph structure used in the experiments.

Using the artificial domain described above we generated 432 testing instances of planning problems with the various level of complexity ranging from simple to very difficult to solve. Two parameters were used to modify the complexity of the problems. First, it was the number of action layers (n), ranging from 1 to 204. Second, we used the number of actions per single layer (m). In order to keep the number of testing problems low, for most configurations of parameter n we set the number of actions per layer to 10, with the exception of a few selected configurations for which we generated instances with the value of m ranging from 1 to 20. For all generated actions we used a random number of preconditions (equal random distribution) ranging from 1 to 3, and a random number of effects ranging from 2 to 3. All planning problems were generated using the standard PDDL syntax.

The experiments ran on Intel Xeon CPU E5335 2.0 GHz processor with 8GB RAM under Ubuntu Linux 8.04.2 (Hardy Heron). The ANA* planner was implemented in Java 1.6, the RelaxPlan planner was implemented using the clpfd library of SICStus Prolog 4.2.0. Because of the big number of tested problem instances, the time limit for solving a single planning problem was set to 5 minutes.

In order to evaluate the performance of the introduced new planners we decided to compare them to the three state-of-the-art domain-independent planners that were successful in IPC 2011. Namely, as we were interested in finding optimal (shortest) plans, the obvious choice was the winner of the sequential optimal track, Fast Downward: Stone Soup 1 (FDSS1) (Helmert et al. 2011), which uses a portfolio of selected successful planning techniques, such as BJOLP (Big Joint Optimal Landmark Planner) (Domshlak et al. 2011), LM-cut (A* with the landmark-cut heuristics) (Helmert and Domshlak 2009), or M&S-bisim1 and M&S-bisim2 (A* with two different merge-and-shrink heuristics) (Nissim et al. 2011). Each ingredient of the portfolio is then assigned a given amount of time limit based on its performance using the method described in (Helmert et al. 2011). Both the selected techniques and their assigned time are derived based on the experiments with the planning domains used for the IPC. However, since we were interested in solving the cumulative planning problems, we felt the urge to also compare the performance with a planner whose parameters were not derived based on the IPC domains. Therefore we chose the third best-performing planner from the IPC 2011 (the second place was taken by yet another variation of the FDSS planner), the SelMax planner, which combines two state-of-the-art admissible heuristics using an online learning approach (Domshlak et al. 2011). Finally, we also used LAMA 2011 planner (Richter et al. 2011), the well-know winner of the sequential satisficing track of IPC 2011.

Table 1 summarizes the results of our experimental evaluation. As it can be seen, out of the 432 tested instances the existing state-of-the-art optimal planners, FDSS1 and SelMax, managed to solve 233 and 235 planning problems, respectively, without providing any sub-optimal solutions for the unsolved problems. This fact was the reason to include also the LAMA 2011 planner to our experiments, which managed to provide a (sub-)optimal solution for all tested instances, however, only 64 of them were solved optimally. Nevertheless as the column Cost Sum in Table 1 shows, LAMA 2011 found many optimal plans, just the proof of optimality was missing. On the other hand, the first of the newly proposed planners, the ANA* planner, solved 241 planning problems, while providing a sub-optimal solutions for all but one testing instance. The clear winner of our experiments is the RelaxPlan planner, which solved optimally 313 problems and moreover it also provided sub-optimal solutions for all of the instances. Also, it required the least amount of time necessary to finish the computation.

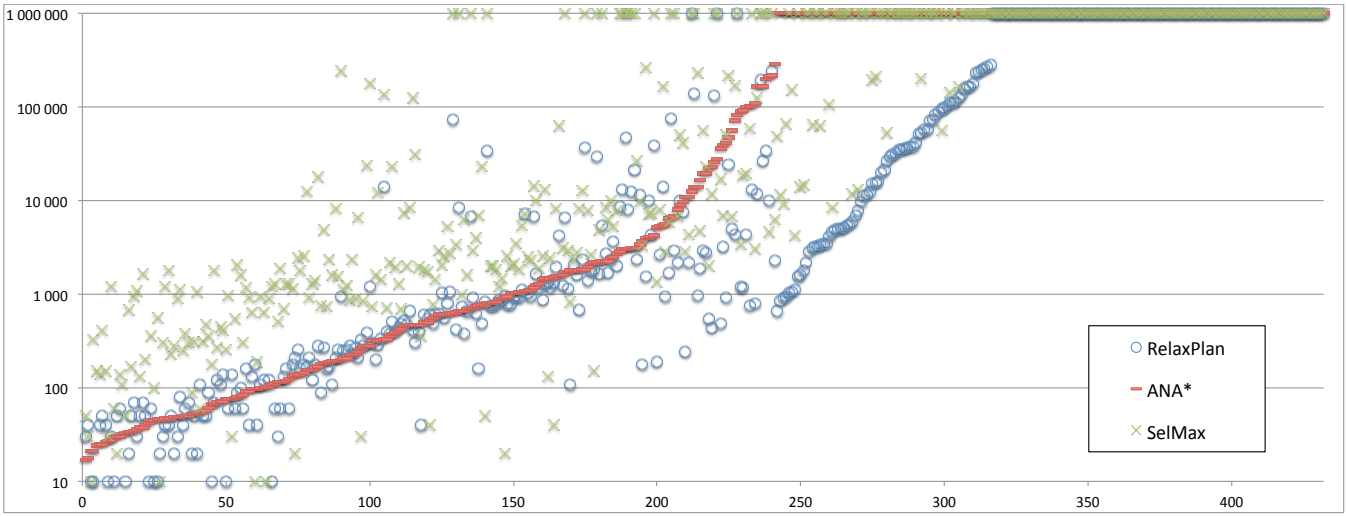


Figure 3. Comparison of runtimes (logarithmic scale) for generated cumulative planning problem. X-axis represents 432 planning problems sorted by ANA* solving time (RelaxPlan solving time is used for sorting when ANA* exceeded the time limit), Y-axis represents the runtime in milliseconds required to solve a given planning problem optimally. For the cases when time-out (set to 5 minutes) occurred we used the value of 1.000.000 to depict them, in order to visually separate such cases from the optimally solved instances more clearly.

Planner	***Solved***	***Optimally Solved***	***Cost Sum***	***Time Sum***
FDSS1	233	233	-	30.58h
SelMax	235	235	-	17.66h
LAMA 2011	432	64	17768	30.88h
ANA*	431	241	17681	16.70h
RelaxPlan	432	313	17756	11.51h

Table 1. Planner performance comparison when solving the 432 generated cumulative planning problems. For each planner the table shows the number of solved instances, the number of optimally solved instances, the total cost of found plans (only for planners that solved all or close-to-all instances), and the total time necessary to run experiments for given planner.

For the three planners that managed to provide sub-optimal solutions for all of the problems we include also the sum of the costs of the best plans they found (since ANA* failed to provide sub-optimal solution only in one case, for that single problem we counted as if it provided the same solution as LAMA 2011 and RelaxPlan). The interesting observation however is that even though the ANA* reached the best score, the differences between the three planners are only tiny, which is interesting especially for LAMA 2011 that provided the guarantee of optimality only in 64 cases, as opposed to RelaxPlan which solved optimally 313 instances.

Figure 3 depicts the difference between the times needed to optimally solve generated planning problems, using the logarithmic scale (note that when the planner exceeded the time limit of 5 minutes, we display its runtime as 1.000.000). For clarity we only include data for ANA*, RelaxPlan and SelMax planners (the SelMax planner exhibited the best performance out of the three existing planners).

As it can be seen, the new planners not only provide the combined advantages of modern satisficing and optimal planners, but also greatly outperform them when solving cumulative planning problems.

Conclusions

Computer games and digital entertainment provide many challenges for artificial intelligence and in particular for planning. Though some planning problems in these areas seem easy, for example the cumulative planning problems, the current state-of-the-art planners have some difficulties to solve them. Hence we proposed two new planners for solving the cumulative planning problems. These planners are not fine-tuned regarding the implementation but they are still beating the best planners from IPC when applied to specific planning problems appearing in certain computer games. So far we did a somehow restricted experimental study but the results naturally raise a more general question – whether the IPC domains examine the planners well in relation to the problems appearing in practice. We left this question un-answered, but the results from this paper suggest that there is indeed a gap between academic planning techniques and close to real-life problems.

Acknowledgement

Research is supported by the Czech Science Foundation under the contract no. P103/10/1287, and by the Grant Agency of Charles University as the project no. 306011 and 9710/2011.

References

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90, 281-300.

Bonet, B. and Helmert, M. 2010. Strengthening Landmark Heuristics via Hitting Sets. *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, IOS Press, 329-334.

Champandard, A. 2008. Getting Started with Decision Making and Control Systems. *AI Game Programming Wisdom IV*, 257-264.

Do, M.B. and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. *Proceedings of the Fifth International Conference on Artificial Planning and Scheduling* (AIPS-2000), AAAI Press, 82-91.

Domshlak, C.; Helmert, M.; Karpas, E.; Keyder, E.; Richter, S.; Röger, G.; Seipp, J. and Westphal, M. 2011. BJOLP: The Big Joint Optimal Landmarks Planner (planner abstract). *Seventh International Planning Competition (IPC 2011), Deterministic Part*, 91-95.

Domshlak, C.; Helmert, M.; Karpas, E. and Markovitch, S. 2011. The SelMax Planner: Online Learning for Speeding up Optimal Planning (planner abstract). *Seventh International Planning Competition (IPC 2011), Deterministic Part*, 108-112.

Helmert, M. and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, AAAI Press, 162-169.

Helmert, H.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S. and Westphal, M. 2011. Fast Downward Stone Soup (planner abstract). *Seventh International Planning Competition (IPC 2011), Deterministic Part*, 38-45.

Kautz, H. and Selman, B. 1992. Planning as satisfiability. *Proceedings of ECAI*, 359-363.

Lau, N. 2008. Knowledge-Based Behavior System-A Decision Tree/Finite State Machine Hybrid. *AI Game Programming Wisdom IV*, 265-274.

Nissim, R.; Hoffmann, J. and Helmert, M. 2011. The Merge-and-Shrink Planner: Bisimulation-based Abstraction for Optimal Planning (planner abstract). *Seventh International Planning Competition (IPC 2011), Deterministic Part, pp. 106-107.*

Orkin, J. 2006. Three states and a plan: the AI of FEAR. *Game Developers Conference*.

Pizzi, D.; Cavazza, M.; Whittaker, A. and Lugrin, J.-L. 2008. Automatic Generation of Game Level Solutions as Storyboards. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, AAAI Press, 96-101.

Richter, S. and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, AAAI Press, 273-280.

Richter, S.; Westphal, M. and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). *Seventh International Planning Competition (IPC 2011), Deterministic Part*, 50-54.

van den Berg, J.; Shah, R.; Huang, A. and Goldberg, K. 2011. ANA*: Anytime Nonparametric A*. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, AAAI Press, 105-111.

Vidal, V. and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, 570-577.

Two Semantics for Step-Parallel Planning: Which One to Choose?

Tomáš Balyo[⋆] and **Daniel Toropila**[⋆†] and **Roman Barták**[⋆]
{tomas.balyo,daniel.toropila,roman.bartak}@mff.cuni.cz

[⋆]Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, 118 00 Prague, Czech Republic

[†]Computer Science Center, Charles University
Ovocný trh 5, 116 36 Prague, Czech Republic

Abstract

Parallel planning is a paradigm that provides interesting efficiency improvements in the field of classical AI planning and it is one of the key components of successful SAT-based planers. Popularized by the Graphplan algorithm, it provides more structure information about the plan for a plan executor when compared to the traditional sequential plan, which is one of the crucial facts of its usability in real-life scenarios and applications. Our latest research shows that different semantics can be used for parallel planning, while deciding which semantic to use for given application can have significant influence on both practical planning domain modeling and also on the computational efficiency of searching for a plan.

Introduction

Classical AI planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state. We assume a world which is fully observable (we know precisely the state of the world), deterministic (the state after performing the action is known), and static (only the entity for which we plan changes the world), with a finite (though possibly large) number of states. We also assume actions to be instantaneous so we only deal with action sequencing. Actions are usually described by a set of preconditions – features that must hold in a state to make the action applicable to that state – and a set of effects – changes that the action makes to the state. Action sequencing is naturally restricted by causal relations between the actions, i.e., the effect of certain action gives a precondition of another action.

Traditional sequential planning algorithms explore directly the sequences of actions. One of the disadvantages of this approach is liability to exploring symmetrical plans where some actions can be swapped without changing the overall effect. Hence if some sequence of actions does not lead to a goal then the algorithm may explore a similar sequence of actions where certain actions are swapped though this sequence leads to exactly same non-goal state. This is called plan-permutation symmetry (Long and Fox 2003). It is possible to remove some of these symmetries by symmetry breaking constraints as suggested in (Grandcolas and Pain-Barre 2007) or (Barták and Toropila 2009). Another way to resolve this problem is partial-order planning where the plans are kept as partially ordered sets of actions (the partial order respects the causal relations). CPT planner (Vidal and Geffner 2006) is probably the most successful (in terms of International Planning Competition) constraint-based planner that does partial-order planning. A half way between partial-order and sequential planning is parallel planning, where the plan is represented as a sequence of sets of actions such that any ordering of actions in the sets gives a traditional sequential plan. This concept was popularised by the Graphplan algorithm (Blum and Furst 1997) that introduced a so called planning graph to efficiently represent causal relations between the actions. Planning graph became a popular representation of parallel plans for approaches that translate the planning problem to other formalisms such as Boolean satisfiability or constraint satisfaction (Do and Kambhampati 2001) (Lopez and Bacchus 2003).

Besides the elegant handling of the symmetries, another advantage of parallel planning is its strong usability for actual plan execution as the concept of parallel plan is much closer to the real-life scenarios. As an example we can imagine a transportation planning domain, such as logistics (Long et al. 2000), where the available actions represent mainly movements of vehicles and loading/unloading of these. Obviously, it is proficient for a plan executor to know that loading of one vehicle can be realized in parallel with sending another vehicle from one location to another. Other useful examples can be devised very easily.

Even though the concept of parallel planning is straightforward, our latest research shows that multiple useful semantics can be introduced for parallel planning by modifying the constraints specifying which actions can appear together in a single (parallel) step of a plan. In this paper we study the differences between the two main semantic concepts, proving the answers for two main questions: first we are investigating the influence of these differences on practical domain modeling, while, second, our main focus is exploring the impact on performance of searching for a plan.

The paper is organized as follows. First we provide necessary theoretical concepts together with the description of the two different parallel plan semantics we studied. After that, following sections discuss the impact of the differences between the two semantics on the practical usability for planning domain modeling and also on the computational ef-

ficiency. We then conclude by providing the experimental evaluation and final discussion of our results.

SAS$^+$ Planning

We use SAS$^+$ formalism to formalize the planning problem. This formalism is based on so called multi-valued *state variables*, as mentioned in (Bäckström and Nebel 1995) or (Helmert 2006). For each feature of the world, there is a variable describing this feature, for example the position of a robot. World state is then specified by values of all state variables at the given state. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies evolution of values of certain state variable. Actions are then the entities changing the values of state variables. Each action consists of preconditions specifying required values of certain state variables and effects of setting the values of certain state variables. We implicitly assume the frame axiom, that is, other state variables than those mentioned among the effects of the action are not changed by applying the action.

Formally, a planning task P in the SAS$^+$ formalism, alternatively referred to also as a SAS$^+$ planning problem, is defined as a tuple $P = \{X, A, s_I, s_G\}$, where

- $X = \{x_1, ..., x_n\}$ is a set of state variables, each with an associated finite domain $Dom(x_i)$;
- A is a set of actions and each action $a \in A$ is a tuple $(\textit{pre}(a), \textit{eff}(a))$, where $\textit{pre}(a)$ and $\textit{eff}(a)$ are sets of (partial) state variable assignments in the form of $x_i = v, v \in Dom(x_i)$;
- A state s is a set of assignments with all the state variables assigned. We denote s_I as the initial state, and s_G as a partial assignment that defines the goal (thus, there can be multiple states that satisfy the goal). We say state s is a goal state if $s_G \subseteq s$.

To express that a state variable x is assigned a value $v \in Dom(x)$ in a state s, we write $s(x) = v$. We say a is *applicable* in state s if all assignments in $\textit{pre}(a)$ match the evaluation of variables in s. We then use $\gamma(s, a)$ to denote the state after applying a to s (according to the assignments defined in $\textit{eff}(a)$).

Based on the formalism above we can also define the transitions that correspond to changes of variable assignments, as described in (Huang, Chen, and Zhang 2010). Given a state variable x, a *transition* is a change of the assignment of x from value f to g, denoted as $\delta^x_{f\to g}$, or from an unknown value to g, denoted as $\delta^x_{*\to g}$. Where unambiguous, the notation can be simplified to δ^x or δ.

For an action a, three types of transitions can be identified:

1. Transitions $\delta^x_{f\to g}$ such that $(x = f) \in \textit{pre}(a)$, and $(x = g) \in \textit{eff}(a)$. We call this type of transitions *active*. An active transition $\delta^x_{f\to g}$ is applicable to a state s, if and only if $s(x) = f$. We denote $\gamma(s, \delta)$ as the state after applying transition δ to state s, which results in a new state s' such that $s'(x) = g$.
2. Transitions $\delta^x_{f\to f}$ such that no assignment to x is in $\textit{eff}(a)$, and $(x = f) \in \textit{pre}(a)$. We call this type of transitions *prevailing*. A prevailing transition $\delta^x_{f\to f}$ is applicable to a state s, if and only if $s(x) = f$, resulting in the same state s.
3. Transitions $\delta^x_{*\to g}$ such that no assignment to x is in $\textit{pre}(a)$, and $(x = g) \in \textit{eff}(a)$. We call this type of transitions *mechanical*. A mechanical transition $\delta^x_{*\to g}$ can be applied to an arbitrary state s, with the result being a state s' where $s'(x) = g$.

For each action a, we denote its *transition set* as $M(a)$, which includes all three types of transitions above. Given a transition δ, we use $A(\delta)$ to denote the set of actions that include δ in their transition set. We call $A(\delta)$ the *supporting action set* of δ. Further, we use $R(x) = \{\delta^x_{f\to f} | \forall f, f \in Dom(x)\}$ to denote the *set of all prevailing transitions* related to x. Then we can introduce $T(x) = \{\delta^x | \exists a \in A, \delta^x \in M(a)\} \cup R(x)$, which represents the *set of all legal transitions* for the state variable x.

Two Semantics

When having a closer look at a sequential plan, one can very often realize that some of the actions included in the plan could be in fact swapped, or even executed together, without affecting the final result of the plan execution. This is probably not surprising observation, as many plans typically involve almost independent sub-plans, that have to be synchronized with each other only in a few specific time steps. As an example of such we can use a plan that involves activities for two trucks, each of which can have its own separate sub-plan of moves and loading/unloading some cargo, while the only needed synchronization between the two trucks is necessary in case one of them is waiting for the cargo that other truck is to deliver first, or in case both of them need to load/unload their cargo in a location where only one truck can operate at a time. Instead of a plain action sequence $\pi = \{a_1, ..., a_k\}$ we could express a solution for a given planning problem using a richer structure – by providing a sequence of sets of actions $\pi' = \{A_1, ..., A_m\}, \forall A_i \exists j : a_j \in A_i$, where for each set A_i all of the actions contained within could be executed in parallel. Of course, one could go further and provide a full graph of causal relations of individual actions contained in the plan, obtaining thus a partial-order plan (Vidal and Geffner 2006). However, constructing such a structure might not be too simple, and is out of the scope of this paper.

There are two motivations behind using parallel plans. First, it is a natural way how to remove some of the plan-permutation symmetries, so the planner does not have to explore the plans where a pair of actions within a set corresponding to single time step is swapped. This fact should, ultimately, help to improve the performance of a planner. Second motivation is even more practical – the knowledge provided by a parallel plan structure is, in fact, very useful for the plan execution phase, since the executor knows which actions can be performed in parallel (if possible), and thus the whole plan can be executed using less time steps.

There are, however, two different ways to define which actions can appear together in a single (parallel) time step, providing thus two different semantics for the step-parallel planning.

Strict Semantics

Probably the most natural requirement on the actions in a single set A_i is their strict pairwise independence, as defined in (Ghallab, Nau, and Traverso 2004), and also as originally described in the Graphplan planning system (Blum and Furst 1997), which was the first one to introduce step-parallel plans as a technique to rapidly improve the efficiency of solving planning problems. Needless to say, the preconditions of the actions within A_i must be non-conflicting.

We say that two actions a_1, a_2 are *independent* in case a_1 does not destroy the preconditions or effects of a_2, and vice versa. The consequence of the requirement of pairwise independence between actions from A_i together with their non-conflicting preconditions is that the actions from a single set A_i can be executed in an arbitrary sequential or parallel order. Therefore the only synchronization requirement on the plan execution is that prior to executing actions from a set A_i all action from A_{i-1} must be executed.

If two actions cannot appear within the same set A_i, we say they are mutually exclusive, i.e., they are in *action mutex*. The fact of which actions are allowed to appear within the same set A_i can be also expressed using the corresponding transitions. The actions a_1, a_2 are in action mutex if there exists a pair of transitions $\delta_1 \in M(a_1)$ and $\delta_2 \in M(a_2)$ such that these two transitions are in *transition mutex*. The two different definitions of the transition mutex will help us draw the difference between the two semantics discussed in this paper.

First, let us define the transition mutex for the *strict semantics*. A pair of transitions δ_1, δ_2 is in *strict transition mutex* if there exists $x \in X$ such that $\delta_1 \in T(x)$ and also $\delta_2 \in T(x)$, meaning they are both based on the same state variable. In other words, all transitions of a given state variable are in transition mutex.

The above relation is a corollary of the following observation. Given the actions a_1, a_2, if there is no pair of transitions $\delta_1^x \in M(a_1), \delta_2^y \in M(a_2)$ such that $x = y$, i.e., both transitions are based on the same state variable, then a_1, a_2 are independent and have non-conflicting preconditions. For the sake of evidence we include a sketch of the proof. If there is no common state variable for any pair of transitions δ_1^x, δ_2^y, the actions a_1, a_2 can neither affect each other nor have conflicting preconditions, since if they did so, there would be a state variable proving the conflict together with the associated transitions – which proves the required implication.

Interestingly, it turns out that the reverse implication does not hold. Hence, that means that the definition using the transition mutex is stricter than the one using the action independence (the fact of which also provided the name for this semantics). However, as proven above, it still guarantees the possibility of an arbitrary ordering during the execution of actions within A_i.

Synchronized Semantics

For the previous semantics we required the universal interchangeability and executability (whether sequential or parallel) for all actions within a single set A_i. In other words, all possible orderings of the actions led to the same state. It is however possible to introduce a less strict semantics, that will in turn pose some additional requirements on the action execution phase. Let us first define the new semantics using a different specification of transition mutex.

For *synchronized semantics*, two different transitions δ_1 and δ_2 are mutually exclusive, i.e., δ_1 and δ_2 are a pair of transition mutex, if there exists a state variable $x \in X$ such that $\delta_1, \delta_2 \in T(x)$, and either of the following holds:

1. Neither δ_1 nor δ_2 is a mechanical transition.
2. Both δ_1 and δ_2 are mechanical transitions.
3. Only one of δ_1 and δ_2 is a mechanical transition and they do not transit to the same variable assignment.

To better interpret the definition above, there are only two cases when $\delta_1, \delta_2 \in T(x), \delta_1 \neq \delta_2$ are not mutex. The first case is, without loss of generality, when $\delta_1 = \delta^x_{f \to f}$ and $\delta_2 = \delta^x_{* \to f}$ for any value $f \in Dom(x)$. The second case is, when $\delta_1 = \delta^x_{e \to f}$ and $\delta_2 = \delta^x_{* \to f}$ for any values $e, f \in Dom(x), e \neq f$.

In order to illustrate the semantical difference between the two definitions of transition mutex, consider the three actions from Table 1. According to the strict semantics, no two actions of these are allowed to appear within a set A_i, since a_1, a_2, a_3 are all pairwise mutex. On the other hand, according to the synchronized semantics, none of these actions are mutually exclusive, and therefore all of them can appear together within a single parallel step as long as their preconditions are met. However, there is a following complication: no valid sequential ordering exists for these three actions! The reason for this is that anytime an action is executed, it destroys a precondition of some other action. Still, it is possible to execute these three actions in a valid way within a single step – but only with the condition of their perfectly synchronized parallel execution.

As we depicted using the above example, the synchronized semantics does not guarantee the possibility of an arbitrary sequential or parallel order of action execution. In fact, there might be no sequential execution available at all for a given parallel step A_i, in case of which the synchronized parallel execution of some of the actions might be required in order not to break the plan validity.

Impact on Usability

Clearly, both semantics allow different actions within the steps of a parallel plan. Before we study the performance of these two semantics, let us have a closer look at their influence on the planning domain modeling.

The main objection that can be raised against the practical use of the synchronized semantics is that the precise synchronization of the start of the actions is mostly infeasible in practice, so the planning domain designer does not want his model to be too fragile on the time constraints. Even though being a bit absurd, consider the following Bomb Terrorist planning domain. Let our modeled world consist of three terrorist t_1, t_2, t_3, each of them wearing a pack of dynamite. As usual in their community, only a terrorist that actually fires his explosives can reach the eternal fame, which is, obviously, everyone's ultimate goal. However once a bomb ex-

Table 1: Example of a set of actions that can be executed only in parallel.

Action	*Preconditions*	*Effects*	*Associated Transitions*
a_1	$x = a$	$x \leftarrow b, z \leftarrow f$	$\delta^x_{a\rightarrow b}, \delta^z_{*\rightarrow f}$
a_2	$y = c$	$y \leftarrow d, x \leftarrow b$	$\delta^y_{c\rightarrow d}, \delta^x_{*\rightarrow b}$
a_3	$z = e$	$z \leftarrow f, y \leftarrow d$	$\delta^z_{e\rightarrow f}, \delta^y_{*\rightarrow d}$

Table 2: The actions available for the silly Bomb Terrorist domain.

Action	*Preconditions*	*Effects*
$fire_1$	$alive_1 = true$	$alive_1 = false, alive_2 = false, alive_3 = false, fame_1 = true$
$fire_2$	$alive_2 = true$	$alive_1 = false, alive_2 = false, alive_3 = false, fame_2 = true$
$fire_3$	$alive_3 = true$	$alive_1 = false, alive_2 = false, alive_3 = false, fame_3 = true$

plodes, everything around is exterminated. Given the terrorist are located inside a room (together with a secret wooden box they are supposed to eliminate), the actions from Table 2 are available in the modeled world.

It is very unlikely that in such scenario it would be possible for all of the terrorists to fire their explosives at once. Still, it is absolutely correct to ask planner the question whether a plan of providing the fame for all three terrorists exists. For our model, the answer of the planner would depend on the underlying semantics it implements. A planner that complies with the strict semantics would answer that no such plan exists, while, on the other hand, a planner compliant with the synchronized semantics would return a plan consisting of a single parallel step that would include all available fire-actions.

Using the example above we illustrated that different semantics of the step-parallel planning used for the same domain can provide completely different results regarding the existence of a plan. Moreover, other examples can be constructed where the return plans will differ in the plan length or total number of used actions. Since we used very small and simple example, it is very easy to see the potential issues, however for the greater models with tens or hundreds of actions and state variables similar issues might be very difficult to trace.

The construction of the planning domain model is purely a task of a domain designer, however a special care must be taken in order to keep the required relevance to the reality, in case the planning technology to be used returns parallel plans. An example of a planner that uses synchronized semantics is SASE planning system, as described and published in (Huang, Chen, and Zhang 2010).

Impact on Performance

Regardless of the usability of the described parallel plan semantics, we were interested in their practical performance. In other words, we wanted to know the use of which semantics leads to finding plans faster. In order to find out we implemented the two versions of a SAT-based planner, which differ only in the definition of the transition mutex relation, as described in one of the earlier chapters. For both versions, the total number of the instances solved within a time limit together with the time required to solve them was measured on a large set of standard planning benchmark problems from the previous International Planning Competitions (IPC). Further details of our planner and the experiments will be described later in this paper.

Knowing which parallel plan semantics is practically faster would be useful when deciding which semantics to use at the phase of planning domain design.

Planner description

Our planner is a Java application which takes SAS$^+$ files as the input. We will refer to it as *SasPlan*. In order to translate PDDL files to SAS$^+$ formalism we used Helmert's Translator tool implemented in Python (Helmert 2006). For each planning problem within our benchmark set it took at most a few seconds for the Translator to generate the SAS$^+$ input file. Since our goal was to compare the efficiency of the two parallel plan semantics, for the final evaluation we only measured the runtime of our application, not including the translation time (which was also fast compared to the time necessary for solving).

SasPlan is basically a black-box planner which encodes SAS$^+$ problems into satisfiability (SAT) problems using the SASE transition-based encoding (Huang, Chen, and Zhang 2010). The operation of the planner can be roughly summarized as follows. First we generate a SAT formula that is satisfiable if and only if there is a plan with timespan equal to one. If the formula is unsatisfiable, we increase the timespan by one. Ultimately, we generate a satisfiable formula, and then we can extract a valid parallel plan from its satisfying assignment.

To solve the generated SAT problems we used SAT4J – a Java library for satisfiability by Daniel Le Berre (Berre and Parrain 2010). Although SAT4J is a few times slower than state of the art solvers written in C/C++, it supports incremental solving and is very easy to use within a Java application. Employing a state-of-the-art C++ solver would considerably increase the overall performance of the planner, since most of the runtime is spent on solving the formulas. MiniSat 2.2 (Eén and Sörensson 2003) would be an appropriate choice, as it is one of the fastest SAT solvers, supporting also

the incremental solving technique. Nevertheless, in order to examine the impact of the different semantics on the performance, the use of Java-based SAT solver was sufficient.

The way we use the support for incremental solving is very straightforward. When we need to generate and solve the formula for the next timespan, we first remove the clauses that require the goal conditions to hold at the end of the plan. Second, we add new variables for the next time step, together with all the clauses that ensure plan validity and the goal conditions. During the tests performed prior to the evaluation itself we observed that the SAT solver did gain some benefit from being used incrementally, compared to getting a new unknown formula at each time step. We did not however do thorough experiments on a large data set in order to compare the incremental and non-incremental approach in bigger detail, since in all tests we made the incremental version was always faster. Therefore we decided to use the incremental solving approach as a default configuration for both versions of our planner.

As stated earlier, we created two versions of SasPlan that correspond to the two transition mutex definitions. During the implementation phase we noticed that the strict semantics is much easier to encode. Since according to the strict mutex definition all transitions of a state variable are mutex, it is sufficient to add one constraint for every state variable ensuring that at most one of the corresponding transitions can be selected at given time step. In other words, a graph of transition mutex relations is a clique, which can be encoded into SAT clauses very efficiently, and which is also one of the reasons why our strict semantics is defined in a little bit stricter way compared to the definition using action independence. Actually, the interface of SAT4J provides a method for adding the *at-most-one* constraints and thus the transition mutex constraints can be added by just one method invocation for each variable. On the other hand, the mutex graph for the synchronized semantics is not a clique since not all transitions of a state variable are pairwise mutex. Therefore these relations cannot be encoded that efficiently.

Experiments

For the experiments we used a PC with 3.2GHz Intel Core i7 processor and 24GB of RAM. We limited the memory to 4GB and used a time limit of 30 minutes per instance. For the evaluation of the planner performance the following domains were selected from the available International Planning Competition (IPC) benchmark set: Airport, Depots, Driverlog, Elevator, FreeCell, Openstacks, Rovers, TPP and Zenotravel.

Before the actual experiments it was difficult to predict the results of our experiments, and therefore our expectations were not clear. The strict transition mutex definition constraints the problem more and thus the resulting SAT formula has less or equal solutions for a given makespan than for the other mutex definition. This could make us believe that the performance can be decreased. On the other hand, more constraints help unit propagation to prune the search space more efficiently. Because of these reasons it was very hard to tell in general which reasoning was stronger and how the performance of the SAT solver will be influenced.

Table 3: Total number of solved instances in the time limit of 30 minutes, per domain.

Domain	*Strict*	*Synchronized*	*Difference*
Airport	30	27	+3
Depots	13	12	+1
Driverlog	15	14	+1
Elevator	46	47	-1
Freecell	4	4	0
Openstack	5	5	0
Rovers	31	32	-1
TPP	27	27	0
Zenotravel	15	14	+1

In Table 3 we provide the total count of solved instances within the time limit, per domain. The differences are not big and for all domains except Elevator and Rovers the strict semantics solved greater or equal number of instances. Figure 1 presents the runtime increase ratio when solving the problems using the synchronized semantics compared to the strict semantics. We only considered the running times for those problems that were solved by both versions of SasPlan. Values above zero mean that the strict semantics is faster. For example, the y-value of 1 means that the runtime using the synchronized semantics was twice as long as the runtime using the strict semantics, i.e., $y = (runtime_{sync} - runtime_{strict})/runtime_{strict}$. From the box plots we can see that even though the strict semantics is not a clear winner in all domains, for some problems its runtime was over 3x faster compared to the synchronized semantics, while the experienced slowdown for other problems was less evident.

Overall, we can conclude that even though the difference between the efficiency of the presented semantics for the parallel planning is not dramatic, the strict semantics exhibited better performance compared to the synchronized one. Since the strict semantics is also more natural and practical for the real-world applications, we believe it should be preferred for majority of cases.

Conclusion

Step-parallel planning is a paradigm useful both for dealing with some of the plan-permutation symmetry problems and also for a practical execution of plans.

In this work we described the two available semantics for the parallel planning, for which also the implementations exist: strict semantics and synchronized semantics. After formally describing the theoretical concepts behind the two semantics, we demonstrated that the planning domain designer's awareness of the selected parallel planning technology and its underlying semantics is a crucial component of a successful application of automated planning in the real-world scenarios.

Finally, we provided the empirical evaluation of the performance of the two presented semantics by integrating them into the implemented SAT-based planning system. The ex-

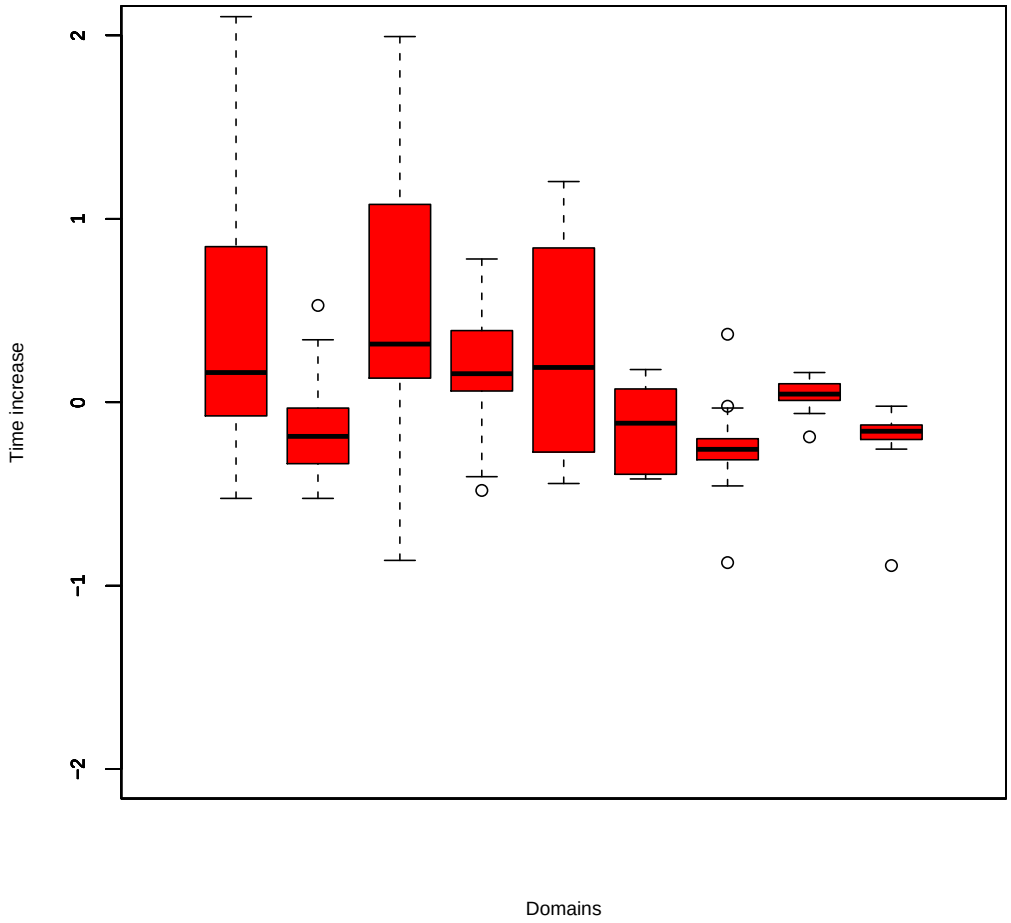


Figure 1: Box plot representation of the runtime increase ratio when solving the problems using the synchronized semantics compared to the strict semantics. Values above zero mean that the strict semantics is faster. For example, the y-value of 1 means that the runtime using the synchronized semantics was twice as long as the runtime using the strict semantics. The box plots represent the results for domains in following (alphabetical) order: Airport, Depots, Driverlog, Elevator, FreeCell, Openstacks, Rovers, TPP and Zenotravel

periments, for which we used some of the traditional IPC benchmark domains, showed that the strict semantics provides better performance in terms of runtime, while it is also more natural for the practical application.

Acknowledgment

The research is supported by the Czech Science Foundation under the projects no. P103/10/1287, 201/09/H057, SVV project number 263 314 and by the Charles University Grant Agency under contracts no. 9710/2011 and 266111.

References

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence* 11:625–656.

Barták, R., and Toropila, D. 2009. Revisiting Constraint Models for Planning Problems. In *ISMIS '09: Proceedings of the Eighteenth International Symposium on Foundations of Intelligent Systems*, 582–591. Berlin, Heidelberg: Springer-Verlag.

Berre, D. L., and Parrain, A. 2010. The Sat4j library, release 2.2. *JSAT* 7(2-3):59–6.

Blum, A., and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90(1-2):281–300.

Do, M. B., and Kambhampati, S. 2001. Planning as Constraint Satisfaction: Solving the Planning Graph by Compiling It into CSP. *Artificial Intelligence* 132(2):151–182.

Eén, N., and Sörensson, N. 2003. An Extensible SAT-solver. In Giunchiglia, E., and Tacchella, A., eds., *SAT*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

Grandcolas, S., and Pain-Barre, C. 2007. Filtering, Decomposition and Search Space Reduction for Optimal Sequential Planning. In *AAAI*, 993–998. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Huang, R.; Chen, Y.; and Zhang, W. 2010. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.

Long, D., and Fox, M. 2003. Plan Permutation Symmetries as a Source of Planner Inefficiency. In *Proceedings of UK Workshop on Planning and Scheduling*.

Long, D.; Kautz, H. A.; Selman, B.; Bonet, B.; Geffner, H.; Koehler, J.; Brenner, M.; Hoffmann, J.; Rittinger, F.; Anderson, C. R.; Weld, D. S.; Smith, D. E.; and Fox, M. 2000. The AIPS-98 Planning Competition. *AI Magazine* 21(2):13–33.

Lopez, A., and Bacchus, F. 2003. Generalizing GraphPlan by Formulating Planning as a CSP. In Gottlob, G., and Walsh, T., eds., *IJCAI '03: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 954–960. Acapulco, Mexico: Morgan Kaufmann.

Vidal, V., and Geffner, H. 2006. Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming. *Artificial Intelligence* 170(3):298–335.

Planning as Quantified Boolean Formula

Michael Cashmore and **Maria Fox**
University of Strathclyde
Glasgow, G1 1XH, UK
firstname.lastname@cis.strath.ac.uk

Enrico Giunchiglia
Universit di Genova
16145 Genova (GE), Italy
lastname@unige.it

Abstract

This paper introduces two techniques for translating bounded propositional reachability problems into Quantified Boolean Formulae (QBF). Both translations exploit the PSPACE complexity of the QBF problem to produce encodings logarithmic in the size of the instance and thus exponentially smaller than the corresponding SAT encoding with the same bound. The first encoding is based on the iterative squaring formulation of Rintanen. The second encoding is a tree encoding that is more efficient than the first one, requiring fewer alternations of quantifiers and fewer variables. We present experimental results showing that the approach is feasible, although not yet competitive with current state of the art SAT-based solvers.

1. Introduction

Planning as Satisfiability is one of the most well-known and effective techniques for classical planning: SATPLAN (Kautz and Selman 1992) was an award-winning system in the deterministic track for optimal planners in the first International Planning Competition (IPC) in 1998, the 4th IPC in 2004, and the 5th IPC in 2006. The basic idea is to encode the existence of a plan with $n + 1$ (or fewer) steps as a propositional (SAT) formula obtained by unfolding, n times, the symbolic transition relation of the automaton described by the planning problem. In recent work (Rintanen 2010), the basic SAT approach has been improved by equipping the solver with planning-specific variable and value-ordering heuristics that are similar to the helpful actions filter of FF (Hoffmann and Nebel 2001). This is very effective for solving planning problems in the SAT framework. In general, SAT-based planning, though quite successful, suffers from the problem that it is easy to come up with problems in which the number of steps required is large, making it impossible to even encode the original problem as a propositional formula. The same problem arises in bounded model checking (Biere et al. 1999). The use of compact encoding as Quantified Boolean Formulas (QBFs) combined with the use of QBF solvers has been proposed (Jussila and Biere 2007; Mangassarian, Veneris, and Benedetti 2010; Dershowitz, Hanna, and Katz 2005) as a way to overcome this problem. In particular, (Rintanen 2001; Jussila and Biere 2007) present an encoding that is logarithmic in n, resembling the proof of the PSPACE-hardness of solving QBFs (Savitch 1970; Stockmeyer and Meyer 1973).

Here we introduce two techniques for translating bounded propositional reachability problems into Quantified Boolean Formulae (QBF). Both of the translations that we present exploit the PSPACE complexity of the QBF problem to produce encodings that are logarithmic in n, and thus exponentially smaller than the corresponding SAT encodings with the same bound. The first encoding we present is a recursive formulation similar to that presented in (Rintanen 2001). We make no novelty claims for the first encoding - we are using it simply as an example of the standard recursive formulation which we contrast with our second encoding. The second encoding that we present reduces redundancy in the first formula and is *tree structured*. It is more efficient than the first one in that:

1. the QBF that encodes the existence of a plan of length n has one fewer universal quantifier, and
2. even when the two formulations have the same number of universal quantifiers, it necessitates far fewer variables.

We refer to the first encoding as the *Formula Encoding* and the second as the *Tree Encoding* throughout this paper. In order to determine the effectiveness of the two encodings we run a preliminary experimental analysis showing that the Tree Encoding tends to perform better on hard problems, solving (or proving unsolvable) many instances that cannot be solved (or proved unsolvable) by the Formula Encoding within the 30-minute time bound that we allowed. Furthermore, our results show that the Tree Encoding is much faster (often by at least one or two orders of magnitude) than the Formula Encoding on a large number of instances from classical planning benchmark domains. We show through these encodings that the QBF approach to classical planning in general is feasable. To the best of our knowledge, this is the first attempt in the literature to demonstrate this. We also stress that although we include a direct comparison with state-of-the-art planning as satisfiability techniques, we do not claim that the approaches discussed here are competitive. Indeed, such a comparison is not fair, given the maturity of the research in SAT and in planning as SAT, compared to the maturity of the much younger field of research in QBFs (and of course of classical planning as QBF, starting with this paper). Our goal is to demonstrate that QBF-based encodings are feasible for planning and to encourage further research into making them competitive.

After some preliminaries in Section 2. the two encodings will be introduced in Section 3. Section 4. will detail the experiments run on these encodings and discuss the results, beginning with memory consumption and moving onto time based comparisons. Finally we conclude in Section 5.

2. Preliminaries

Quantified Boolean Formula

Formally, the language of QBFs extends propositional logic by allowing for universal ($\forall$) and existential ($\exists$) quantification over variables (in our case, fluents and actions). Semantically, $\forall x.\varphi$ (resp. $\exists x.\varphi$) can be interpreted as $(\varphi_x \wedge \varphi_{\neg x})$ (resp. $(\varphi_x \vee \varphi_{\neg x})$), where φ_x (resp. $\varphi_{\neg x}$) is the formula obtained from φ by replacing x with $\top$ (resp. $\bot$).[1] The process of substituting $\forall x.\varphi$ (resp. $\exists x.\varphi$) with $(\varphi_x \wedge \varphi_{\neg x})$ (resp. $(\varphi_x \vee \varphi_{\neg x})$) is called *expansion*. By expanding all quantifiers, each QBF can be reduced to (possibly an exponentially larger) propositional formula. When every variable is quantified (in which case we say the QBF is *closed*), such an expansion reduces to a Boolean combination of $\top$ and $\bot$ and is thus equivalent to either $\top$ or $\bot$. The QBF formula expands recursively into a tree-structured representation where all of the propositional variables are at the leaves. Expansion therefore produces the conjuctive binary tree that accords to the semantics of the QBF.

The Planning Problem

Let $\mathcal{F}$ and $\mathcal{A}$ be two finite sets representing the sets of *fluents* and *actions* respectively. In the following, we use X to denote the whole set of variables, i.e., the set of fluents unioned with the set of actions.

A *planning problem* is a triple $\langle I, \tau, G\rangle$ where

- I is a Boolean formula over $\mathcal{F}$ and represents the set of *initial states*;
- τ is a Boolean formula over $X \cup X'$ where $X' = \{x' : x \in X\}$ is a copy of the set of variables and represents the *transition relation* of the automaton describing how (complex) actions affect states (we assume $X \cap X' = \emptyset$);
- G is a Boolean formula over $\mathcal{F}$ and represents the set of *goal states*.

The above definition of the planning problem differs from the traditional ones in which the description of the effects of actions on a state is described in an high-level action language such as STRIPS. We prefer this formulation because the techniques we are going to describe are independent of the action language used. The only assumption that we make is that the description is deterministic: there is only one state satisfying I and the execution of a (complex) action α in a state s can lead to at most one state s'. More formally, for each state s and complex action α there is at most one interpretation extending $s \cup \alpha$ and satisfying τ.

3. Encoding planning problems as QBFs

Consider a planning problem $\Pi = \langle I, \tau, G\rangle$. As standard in planning as satisfiability, the existence of a parallel plan with makespan n is proved by building a propositional formula with n copies of the set of variables. In the following,

- by X_α we denote one such copy of the set of variables;
- by $I(X_\alpha)$ (resp. $G(X_\alpha)$) we denote the formula obtained from I (resp. G) by substituting each $x \in X$ with the corresponding variable $x_\alpha \in X_\alpha$;
- by $\tau(X_\alpha, X_\beta)$ we denote the formula obtained from τ by substituting each variable $x \in X$ with the corresponding variable $x_\alpha \in X_\alpha$ and similarly each $x' \in X'$ with the corresponding $x_\beta \in X_\beta$.

For $n \geq 1$, the *planning problem* Π *with makespan* n is the Boolean formula Π_n defined as

$$I(X_1) \wedge \bigwedge_{i=1}^{n} \tau(X_i, X_{i+1}) \wedge G(X_{n+1}) \qquad (n \geq 0) \quad (1)$$

and a *plan for* Π_n is an interpretation satisfying (1).

However, since the plan existence problem (assuming, as we do, a deterministic transition relation and a single initial state) is a PSPACE-complete problem (Bylander 1994) the size of (1)) can be exponential in the number of fluents - making it impossible to even build (1). QBFs are a promising alternative representation language given that:

1. there exists an encoding of the planning problem with makespan n as QBFs which are polynomial in the number of fluents, and
2. there is a growing interest in developing efficient solvers for QBFs; see, for example, the report from the last QBF competition (Peschiera et al. 2010).

Formula Encoding

Consider the formula:

$$I(X_I) \wedge E_k(X_I, X_G) \wedge G(X_G) \quad (2)$$

where $k \geq 0$ and represents the folding parameter. $E_k(X_I, X_G)$ is defined as follows. In the following, given two finite sets X_α and X_β of variables, $\exists X_\alpha X_\beta$ denotes the result of existentially quantifying each variable in $X_\alpha \cup X_\beta$, and $(X_\alpha \leftrightarrow X_\beta)$ stands for $(\wedge_{x \in X} x_\alpha \leftrightarrow x_\beta)$.

$$\begin{aligned} E_k(X_I, X_G) := \exists X_{st} \forall y \exists X_s X_t (& \\ & (\neg y \Rightarrow ((X_{st} \leftrightarrow X_t) \wedge (X_I \leftrightarrow X_s))) \wedge \\ & (y \Rightarrow ((X_{st} \leftrightarrow X_s) \wedge (X_G \leftrightarrow X_t))) \wedge \\ & E_{k-1}(X_s, X_t) \qquad) \end{aligned}$$

if $k > 0$, and

$$\begin{aligned} E_0(X_I, X_G) := \exists X_1 \exists X_2 & \\ ((X_I \leftrightarrow X_1) \wedge \tau(X_1, X_2) \wedge (X_2 \leftrightarrow X_G)) & \end{aligned}$$

when $k = 0$. The correspondence between (1) and the Formula Encoding is clear when $k = 0$. When $k > 0$, by expanding the universal quantifiers we find the same correspondence. Intuitively, the branch formed by expanding y divides the formula into $E_{k(\neg y)}$ and $E_{k(y)}$, each representing one half of the total timesteps. As X_{st} remains above

[1] $\top$ and $\bot$ are the logical symbols we use for truth and falsity.

this branch it is used to link these two halves together with equality constraints.

The above formulation involves $(3k + 2)|X|$ existential variables and k universal variables. Further, the Formula Encoding can be converted into prenex conjunctive normal form (corresponding to the QDIMACS format used by most QBF solvers) with $4(2k + 1)|X| + |\tau|$ clauses, where $|\tau|$ is the number of clauses in the transition relation of the original planning problem.

Tree Encoding

We now describe a new encoding which removes redundancy and requires considerably fewer variables than the Formula Encoding.

Intuitively, the Formula Encoding describes a one-to-one correspondence between the states traversed and the *leaves* of the expansion corresponding to the QBF. The leaves of the expansion are composed of the innermost existentially quantified variables. By contrast, in our second encoding, which we call a *tree-structured* encoding, there is a one-to-one correspondence between the states traversed and the *nodes* of the tree corresponding to the QBF. Every existentially quantified layer in the QBF is used to represent at least one distinct state.

Given a QBF formula with k universal quantifiers, the new encoding is a tree of depth k which removes all redundancy from the formula, by only specifying equivalent edges once. The key novelty of this encoding is in the traversal of its tree structure, in which edges are encoded from each leaf node to one of the nodes in each of the preceding layers of the tree. This leads to a formula that encodes 2^{k+1} transitions in a tree with k layers. The formula is quadratic in k because every edge to and from level i requires $k - i - 1$ terms to express the context which is the assignment to the variables in the intervening layers. While the Formula Encoding does not require these contexts and therefore is linear in k, twice as many variables are required to enforce the equivalence of sets of existentially quantified variables on different branches as are required in the Tree Encoding to describe the traversal of the tree.

The encoding in question can be written as the formula:

$$I(X_I) \wedge Q_k(X_I, X_G) \wedge G(X_G) \tag{3}$$

where

$$\begin{array}{l} Q_k(X_I, X_G) := \exists X_k \forall y_k \ldots \exists X_1 \forall y_1 \exists X(\\ (\,(\bigwedge_{i=1}^{k} \neg y_i) \Rightarrow \tau(X_I, X)\,) \\ \wedge(\,(\bigwedge_{i=1}^{k} y_i) \Rightarrow \tau(X, X_G)\,) \\ \bigwedge_{i=1}^{k}(\;(\,(\neg y_i \bigwedge_{j=1}^{i-1} y_j) \Rightarrow \tau(X, X_i)\,) \\ \qquad \wedge(\,(y_i \bigwedge_{j=1}^{i-1} \neg y_j) \Rightarrow \tau(X_i, X)\,))\quad) \end{array}$$

This formula states that the goal state G_G is reachable in 2^{k+1} applications of the transition relation. Only $k+1$ states are quantified (X_k to X_1 and X).

The Tree Encoding (3) has k universal variables and $(k + 1)|X|$ existential variables. Further, with (3) we check the existence of plans having makespan equal to 2^{k+1}, i.e., twice the makespan allowed by the Formula Encoding. However, the conversion of (3) to prenex conjunctive normal form has $2(k + 1)|\tau|$ clauses.

The correspondence between the Tree Encoding and (1) is again found when expanding the universally quantified variables. Unlike the Formula Encoding however, when expanding y_k, X_k does not link the two halves using equality constraints, but instead with two transition relations - itself representing a distinct state in the solution to the planning problem.

Abstract Example

The following simple abstract example emphasises the difference between the Tree Encoding and the Formula Encoding.

If we build an expression containing two universal quantifiers using the Tree Encoding, we express the existence of a plan of makespan 8. Below is a simple example of such.

$$\begin{array}{l} \exists X_1 \forall y_1 \exists X_2 \forall y_2 \exists X_3 \cdot (\\ \qquad\qquad\qquad\qquad (\neg y_1 \wedge \neg y_2 \Rightarrow \tau(I, X_3)) \quad \wedge \\ \qquad\qquad\qquad\qquad (y_1 \wedge y_2 \Rightarrow \tau(X_3, G)) \quad \wedge \\ \qquad\qquad\qquad\qquad (\neg y_1 \wedge y_2 \Rightarrow \tau(X_3, X_1)) \quad \wedge \\ \qquad\qquad\qquad\qquad (y_1 \wedge \neg y_2 \Rightarrow \tau(X_1, X_3)) \quad \wedge \\ \qquad\qquad\qquad\qquad (\neg y_2 \Rightarrow \tau(X_3, X_2)) \quad \wedge \\ \qquad\qquad\qquad\qquad (y_2 \Rightarrow \tau(X_2, X_3))) \end{array}$$

It should be noted that the last two transitions are both invoked twice: once in the context where y_1 is true and once in the context where it is false. The other transitions are all invoked once, accounting for all 8 transitions.

By contrast, two universally quantified variables using the Formula Encoding produces:

$$\begin{array}{l} \exists X_1 \forall y_1 \exists X_2 \exists X_3 \cdot (\\ \quad (y_1 \Rightarrow I \leftrightarrow X_2 \wedge X_1 \leftrightarrow X_3) \wedge \\ \quad (\neg y_1 \Rightarrow X_1 \leftrightarrow X_2 \wedge X_3 \leftrightarrow G) \wedge \\ \qquad \exists X_4 \forall y_2 \exists X_5 \exists X_6 \cdot \\ \qquad\quad ((y_2 \Rightarrow X_2 \leftrightarrow X_5 \wedge X_4 \leftrightarrow X_6) \wedge \\ \qquad\quad (\neg y_2 \Rightarrow x_5 \leftrightarrow X_5 \wedge X_3 \leftrightarrow X_6) \wedge \\ \qquad\qquad \exists X_7 \exists x_8 \cdot \\ (x_5 \leftrightarrow X_7 \wedge x_8 \leftrightarrow x_6 \wedge \tau(x_7, x_8)))) \end{array}$$

The single transition is invoked in all of the contexts generated by assignments to y_1 and y_2. This is only 4 contexts, so the encoding expresses the existence of a plan of makespan 4.

It can be seen that the Formula Encoding uses twice as many variables as the Tree Encoding for the same number of universal quantifiers. Also, by expanding the universal quantifiers in both Formula and Tree Encoding, we get propositional formulae, in which the latter has twice as many transition relations. This is a consequence of the tree-structured encoding.

4. Results

We ran some experiments to form a number of comparisons, hoping to show that:

- the QBF approach uses less memory than SAT,
- the Tree Encoding uses less memory, and less time than the Formula encoding,

domain	solver					
	sat		qube		depqbf	
	average	average'	average	average'	average	average'
depots	1944	1427	790	539	**222**	**191**
driverlog	2090	2193	2042	1210	**288**	**299**
gripper	437	496	149	186	**70**	**91**
freecell	2087	2613	784	822	**506**	**513**
philosophers	902	763	**56**	**52**	108	72
pipesnotankage	2095	2194	1956	892	**410**	**321**
rovers	2061	1577	611	432	**536**	**322**

Table 1: Memory used solving problems using SAT and the Tree Encoding, sizes to the nearest MB.

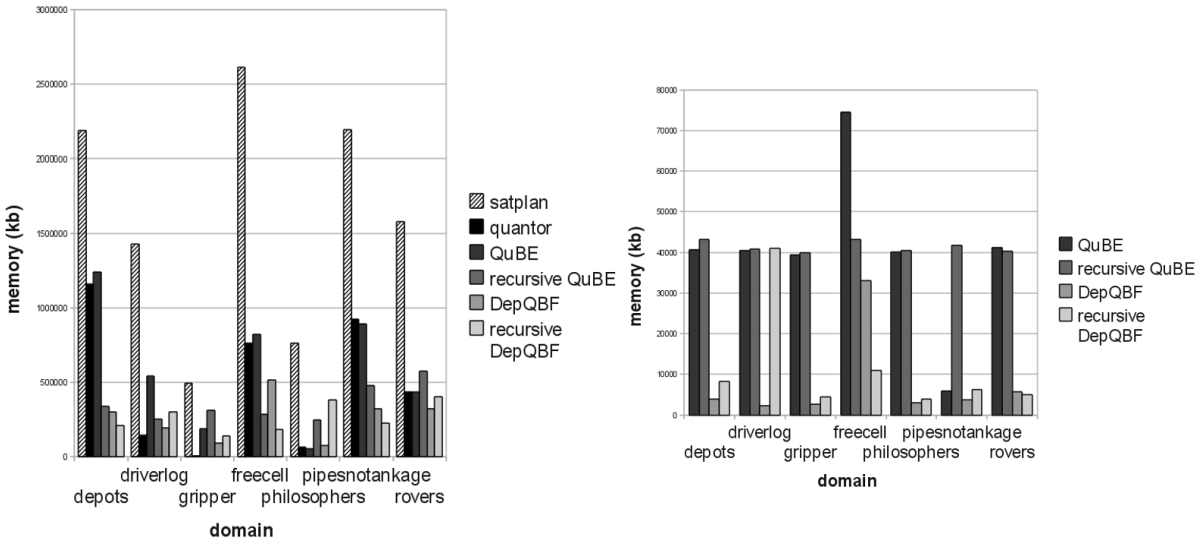


Figure 1: Averge memory usage over time per domain, solving with SAT and QBF encodings. Memory in (kb)

- there exists a problem that can be represented using QBF, but cannot be built with SAT.

We do not expect that the QBF approach is competitive with SAT in terms of time, but include a comparison to illustrate this gap in performance.

For these we used encodings of STRIPS-style planning problems from the IPC benchmarks; *depots, driverlog, freecell, gripper, opticaltelegraph, philosophers, pipesnotankage, rovers, tpp* and *zeno*. These were solved with a variety of solvers in a number of experiments.

The problems were solved using Quantified Boolean Formula as follows:

A plangraph was created until level-off and then encoded as a QBF, this was passed to the choice of solver. If the QBF proved to be unsatisfiable a larger encoding was created and the process repeated. Once a QBF was found to be satisfiable, or the time limit (30 minutes per encoding) was reached, the next problem was considered. This was repeated for each pairing of formula and solver.

Memory Comparison

The memory used for each approach was recorded. For each domain every problem was attempted, under the same time limits and on the same machines as previous experiments. Problems with greater than 5120 grounded fluent and action variables were not attempted. The memory use is displayed in Table 1 and Figure 1.

In Table 1 the *average* column denotes the mean of the maximum and minimum memory footprint of the solver. *average'* is the mean value for the memory used by the solver at small time intervals. As can be seen from the table, the QBF solvers use much less memory than the SAT approach.

The minimum amount of memory used is ideally the amount of space taken once the problem is grounded and translated into a boolean formula. As should be expected this is much smaller in the QBF form. The maximum amount of memory used behaves erratically in the results for QuBE (when compared to the SAT results). This is due to the way in which QuBE approaches the problem: the size of learned cubes can grow very large if the initial guesses to the solution are far from correct and a lot of backtracking is involved. This is often the case in the *depots* domain, in

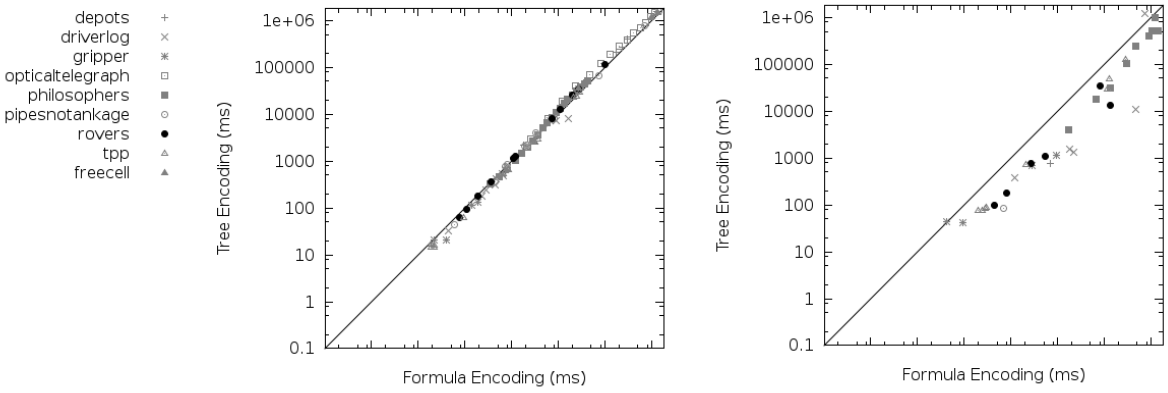


Figure 2: Times on problems solved with *quantor3.0* (left) and *QuBE7.0* (right) using the Tree Encoding and Formula Encoding, times in ms.

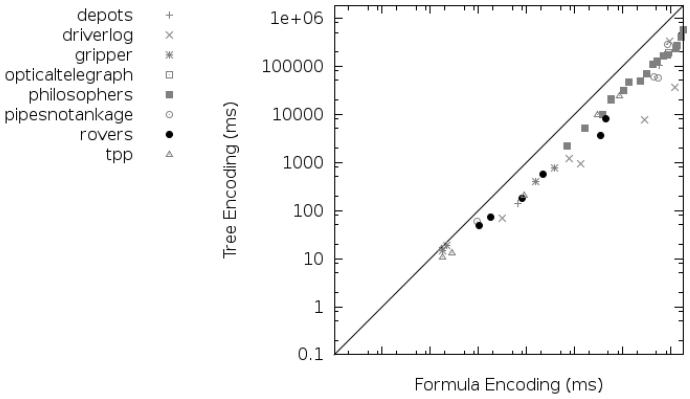


Figure 3: Times on problems solved with *DepQBF* using the Tree Encoding and Formula Encoding, times in ms.

which QuBE performed very poorly.

The amount of memory used over time gives some insight into how memory was used. This is close to the normal average in the QBF-based approaches, in most cases it is smaller. The amount of memory used is low for the majority of the execution time. The opposite is true for the SAT-based approach, in which the amount of memory used increases quickly from the minimum and levels out.

Figure 1 displays the results in the table (*average'*), with the inclusion of the Formula Encoding (solved using both *depQBF* and *QuBe7*). It also includes results for the Tree Encoding solved using *quantor-3.0*. All QBF-based approaches use less memory than SAT - even the resolution based solver *quantor*. The figure also displays the same results excluding both SAT and *quantor-3.0*, in order to better compare the differences between the other QBF-based approaches. In this figure it can be seen that *depQBF* uses far less memory in solving these encodings, and that, although the results are mixed, the smallest memory footprint is found when using the Tree Encoding and *depQBF*.

Comparing the Formula Encoding with the Tree Encoding

Figures 2 and 3 compare times (in ms) for solving with the Tree Encoding and Formula Encoding. These solvers were chosen to illustrate the differences between resolution and search-based solvers.

This comparison shows that the Tree Encoding outperforms the Formula Encoding in time, independent of the choice of solver. All problems from the benchmarks listed above were tested, with only problems solved by both techniques included in the figures. The number of problems solved by each formula is shown in table 3. This table also shows the number of problems solved using one formula that were not solved by the other (unique). Only one problem was solved using the Formula Encoding that couldn't be solved with the Tree Encoding.

The results show that the Tree Encoding performs far bet-

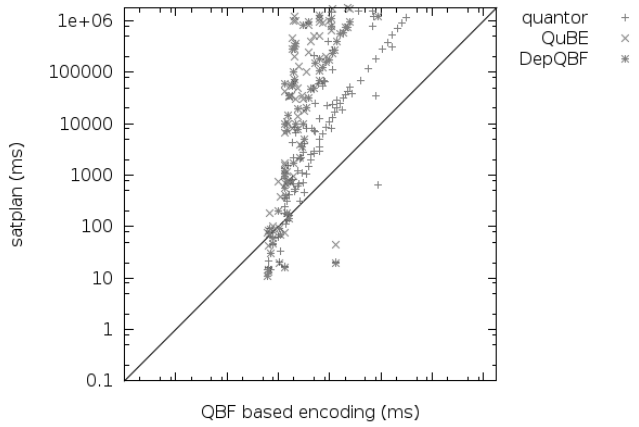


Figure 4: Times on problems solved using the Tree Encoding with various solvers against SAT, times in ms.

	SAT Encoding			Tree Encoding		
Problem Size	file size	vars	clauses	file size	vars	clauses
1	194	6	17	210	6	17
2	6.8K	84	694	5K	57	468
3	243K	840	20924	102K	362	8740
4	8.1M	7440	609976	2M	1987	157760
5	273M	62496	18250352	39M	10084	2879472
6	-	-	-	784M	48773	52755136

Table 2: Sizes of SAT and QBF encodings generated for rectangle packing instances.

ter than the Formula Encoding when using a search-based QBF solver (*QuBE7.0* and *DepQBF*).

quantor3.0, which uses a Q-resolution based approach (Kleine-Büning, Karpinski, and Flögel 1995), tackles variables *bottom-up* in the expanded quantifier tree. In the Formula Encoding this eliminates some of the problems caused by the redundancy, as all of it occurs in non-leaf nodes. As can be seen in Figure 2 on commonly solved problems, there is very little time difference. However, the additional universal quantifier still causes problems on harder problems, particularly on those with larger makespans, as can be seen in the disparity in table 3.

Comparison with SAT

The experiment carried out to compare the Tree Encoding with the Formula Encoding was repeated using a SAT based planner. *SATPLAN'06* was used for this as it uses the same translation of Plangraph and states to boolean formula as the encodings described here (Kautz and Selman 1996).

The results are shown in Figure 4.

Although on some smaller problems the QBF approach is faster, overall it scales much more poorly than the SAT-based planner.

Rectangle Packing Comparison

A trivial form of the Rectangle packing problems was used to demonstrate limitations in the ability of SAT to even build large problem instances. A number of different sized instances were constructed as planning problems, in order to create an example problem that is difficult for SAT to represent. The general Rectangle Packing Decision Problem is to place smaller rectangular shapes into a larger rectangular container such that:

1. None of the smaller rectangles overlap.
2. Every rectangle is placed in the final state.

The problem has been very well studied by Korf (2004). In our simple version, a number of instances were generated with incrementally larger containers. The area of each container's grid is 2^p, where p is referred to as the Problem Size. The rectangles to be placed in the container are all squares of unit size, with their combined area equal to the size of the container. This problem is converted into a planning problem by representing space in the container as a grid of cells, with the filled condition of each cell described by a fluent. Fluents also describe the placement of the rectangles; one fluent exists for each possible placement of each rectangle.

The problem is trivial as there are no incorrect actions to perform and every solution is symetrically identical. However, the problem was chosen only to examine the difficulty in representation - if the problem cannot be represented in the first place, its difficulty is immaterial. If it can be represented then it is worth going on to consider how to solve the interesting general case. It is plausible that a planning domain may contain an instance of a Rectangle Planning problem, and that current planning approaches could be required to deal with this problem.

	Formula Encoding		Tree Encoding	
solver	#solved	unique	#solved	unique
quantor3.0	105	1	107	3
QuBE7.0	39	0	61	22
DepQBF	58	0	68	10

Table 3: Number of solved problems using the Tree Encoding and Formula Encoding.

The problem instances were translated into boolean formulae, both SAT and QBF (using the Tree Encoding). In both cases the transition relation and state representations were identical - as described in Section 3. The translation was performed using 8GB of RAM and unlimited time. The process was killed once the size of the file containing the formula became larger than 4GB. The size of each formula is shown in table 2. The table shows the filesize, number of variables and number of clauses for each problem.

The instance of Problem Size 6, which is a container of grid size 64, was not translated into SAT. The filesize of the SAT instance grew larger than 4GB after an extended period of time and the process was killed.

5. Conclusions

In this paper we have presented methods for encoding reachability problems (specifically, propositional planning problems), as QBFs. We described two alternative encodings. The first encoding, the Formula Encoding, is an equivalence-based encoding based on (Rintanen 2001). The second encoding, the Tree Encoding, is a tree-based encoding that uses fewer universal and existential quantified variables than the first when encoding problems with a given makespan. In fact, for plans encoded with the same depth of universal quantification, the Tree Encoding describes a plan of double the makespan that that described by the Formula Encoding. We have experimented with sets of problem instances taken from the IPC benchmarks, and we have shown that the second encoding leads to faster solution (and proof of unsolvability) of the harder problems in these sets. We consider these results to be exciting because the tree-encoding of QBF formulae is more compact than the equivalence-based encodings characterised by the Formula Encoding, and demonstrates potential for marked improvement in the performance of planners based on QBF-solving. We think this work could lead to the development of satisfiability-based planners that are more scalable than propositional SAT-solvers. As final remark, it should be noticed that the QBF encodings of symbolic reachability problems presented in (Dershowitz, Hanna, and Katz 2005; Mangassarian, Veneris, and Benedetti 2010) have size polynomial in the makespan of the problem, and thus possibly exponential in the number of variables.

References

Biere, A.; Cimatti, A.; Clarke, E.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, 193–207.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artif. Intell.* 69(1-2):165–204.

Dershowitz, N.; Hanna, Z.; and Katz, J. 2005. Bounded model checking with QBF. In *SAT*, 408–414.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.* 14:253–302.

Jussila, T., and Biere, A. 2007. Compressing BMC encodings with QBF. *Electr. Notes Theor. Comput. Sci.* 174(3):45–56.

Kautz, H., and Selman, B. 1992. Planning as Satisfiability. In *Proc. ECAI*, 359–363.

Kautz, H., and Selman, B. 1996. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, 1194–1201.

Kleine-Büning, H.; Karpinski, M.; and Flögel, A. 1995. Resolution for quantified Boolean formulas. *Information and Computation* 117(1):12–18.

Korf, R. 2004. Optimal rectangle packing: New results. *The 2004 International Conference on Automated Planning and Scheduling (ICAPS-04)* 142–149.

Mangassarian, H.; Veneris, A. G.; and Benedetti, M. 2010. Robust QBF encodings for sequential circuits with applications to verification, debug, and test. *IEEE Trans. Computers* 59(7):981–994.

Peschiera, C.; Pulina, L.; Tacchella, A.; Bubeck, U.; Kullmann, O.; and Lynce, I. 2010. The seventh QBF solvers evaluation (QBFEVAL'10). In *Proc. SAT*.

Rintanen, J. 2001. Partial implicit unfolding in the Davis-Putnam procedure for Quantified Boolean Formulae. In *Proc. LPAR*, volume 2250 of *LNCS*, 362–376.

Rintanen, J. 2010. Heuristics for planning with SAT. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, 414–428. Berlin, Heidelberg: Springer-Verlag.

Savitch, W. J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.* 4(2):177–192.

Stockmeyer, L. J., and Meyer, A. R. 1973. Word problems requiring exponential time: Preliminary report. In *STOC*, 1–9.

Theoretical Aspects of Using Learning Techniques for Problem Reformulation in Classical Planning

Lukáš Chrpa
Knowledge Engineering and Intelligent Interfaces Research Group
Department of Informatics
School of Computing and Engineering
University of Huddersfield

Abstract

Using learning techniques in planning experienced significant growth during recent years. Learning can provide additional knowledge that accommodated with state-of-the-art planning systems can significantly improve the planning process. There exist several types of such learning techniques providing different types of knowledge (for instance, macro-actions or action elimination). In this paper, we address theoretical aspects of such learning techniques that reformulate planning problems (macro-actions, action elimination) to increase efficiency of (classical) planners. It is studied how these learning techniques influence the planning process with respect to its soundness and completeness. Contribution of this paper is in form of providing a formal framework addressing issues connected with reformulating planning problems. Because this paper presents ongoing work the technical part is followed by a thorough discussion about ideas for future research in this area.

Introduction

Artificial Intelligence (AI) planning is a traditional and hot research topic thanks to its theoretical interest as well as its wide range of real-life applications. Despite the significant improvement of planning systems in recent years many planning problems still remain hard and challenging.

In the last decade, a lot of planning systems were developed and many of them competed in the International Planning Competition (IPC)[1]. However, even the best planning systems often fail to find a solution in a reasonable time if the planning problem is more complex. It seems to be more than reasonable to use learning techniques to gather additional knowledge that should help planners to significantly reduce the depth of the search space as well as the branching factor. There exist several types of such learning techniques providing different types of knowledge. Such knowledge is usually learnt from given training plans that are obtained as solutions of simpler planning problems.

The idea of using learning techniques in planning is not very new. REFLECT (Dawson and Siklóssy 1977) was one of the first systems using learning techniques in planning. Well known type of state-of-the-art learning techniques consists of generating macro-actions (or macro-operators) (Botea et al. 2005; Chrpa 2009; 2010b; Newton et al. 2007; Coles and Smith 2007), where the macro-action (macro-operator) represents a sequence of (primitive) actions (operators). Other types of state-of-the-art learning techniques consist eliminating unnecessary actions (Chrpa and Barták 2009), problem decomposition (Bibai et al. 2008), selecting an efficient planning algorithm (Roberts et al. 2008) or heuristics learning (McCluskey 1987). Some learning techniques (macro-operators, action elimination) can be combined (Chrpa 2010a) to gain an additional improvement of the planning process.

In this paper, we address theoretical aspects of learning techniques (mostly offline based) for planning problem reformulation used in (classical) planning. Planning problems can be reformulated in various ways, the most common consist adding macro-operators removing potentially unnecessary actions or both. Contrary to related works that evaluate proposed learning techniques empirically we focus on study how particular learning techniques influence the planning process, mainly with respect to its soundness and completeness. It should give us an insight how the learning techniques can be used, in terms of potential solvability or insolvability of the original or reformulated (by using learning techniques) problems and come with questions that can influence future research in this area.

The paper is organized as follows. In the next section, we introduce notions known in the planning theory. Then, we briefly introduce main types of the learning techniques for planning. Section 4 is devoted to addressing the theoretical aspects of the learning techniques for planning. Then, we thoroughly discuss possible directions of future research and then, finally, we conclude.

Preliminaries

This section is devoted to a brief introduction of (classical) planning (Ghallab, Nau, and Traverso 2004) that is necessary to understand the paper.

Traditionally, AI planning deals with the problem of finding a sequence of actions transforming the environment from some initial state to a desired goal state. In this paper we will consider only classical planning, i.e. planning in de-

[1] http://ipc.icaps-conference.org

terministic, fully observable and static environment. The definitions below will formally introduce basic notions for classical planning (set-theoretic or STRIPS representation).

Definition 1. *A **planning problem** in the set-theoretic representation (STRIPS) is a 6-tuple* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *such that:*

- P *stands for a finite set of **atoms** (propositions)*
- $S \subseteq 2^P$ *stands for a set of **states***
- $I \in S$ *stands for **initial situation** (initial state)*
- $G \subseteq P$ *stands for **goal situation** (goal state is such a state that contains all the atoms from* G*)*
- A *is a set of **actions**, action* $a \in A$ *is specified via its precondition (*$pre(a) \subseteq P$*), negative effects (*$eff^-(a) \subseteq P$*) and positive effects (*$eff^+(a) \subseteq P$*)*
- *An action* $a \in A$ *is **applicable** in a state* $s \in S$ *iff* $pre(a) \subseteq s$
- $\gamma : S \times A \to S$ *is a **transition function**, where* $\gamma(s, a) = (s \setminus eff^-(a)) \cup eff^+(a)$ *if* a *is applicable in* s*, otherwise* $\gamma(s, a)$ *is undefined*

■

Definition 2. *A **plan** π is a sequence of actions* $\langle a_1, \ldots, a_k\rangle$*, where:*

- *A **length of plan** π is* $|\pi| = k$
- *A state produced by plan* π *(generalization of* γ*) can be recursively computed in the following way:*
  - $\gamma(s, \pi) = s$ *iff* $|\pi| = 0$
  - $\gamma(s, \pi) = \gamma(s, a_1)$ *iff* $|\pi| = 1$ *and* a_1 *is applicable in* s
  - $\gamma(s, \pi) = \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_k\rangle)$ *iff* $|\pi| \geq 2$ *and* a_1 *is applicable in* s
  - $\gamma(s, \pi)$ *is undefined otherwise*

■

Definition 3. *Let* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *be a planning problem and* π *be a plan.* π *is a **solution** of* Π *iff* $G \subseteq \gamma(I, \pi)$*.* π *is **minimal** (in classical planning also optimal) if for every solution* π' *for* Π $|\pi| \leq |\pi'|$*.* ■

Definition 4. *Let* $PROBS$ *be a set of planning problems. Let* $PLANS$ *be a set of plans. We say that a function* $M : PROBS \to PLANS \cup \{\perp\}$ *is a **planning method**. A planning method* M *is **sound** if for every* $\Pi \in PROBS$*,* $\pi \in PLANS$ *such that* $M(\Pi) = \pi$ *holds that* π *is a solution of* Π*. A planning method* M *is **complete** if* M *is sound,* M *is defined for every* $\Pi \in PROBS$ *and* $M(\Pi) = \perp$ *iff* Π *does not have any solution.* ■

Considering a state-space planning approach it is quite obvious that to solve a given planning problem we have to search the state space. State-space planning rather than plan-space planning is more suitable for the theoretical analysis we provide in this paper. The state space can be represented by a (finite) state transition system which can be represented by directed graph where nodes are states and edges are transitions. Obviously, there is a transition from a state s to a state s' if there is an action a such that $\gamma(s, a) = s'$. It is clear that the solvability of the planning problem depends on existence of a path (sequence of transitions) from the initial state to some of the goal states in the corresponding state transition system.

Learning for classical planning

As mentioned in the introduction learning techniques are used for gathering additional knowledge that can improve the planning process. There have been developed many learning techniques in the past. These learning techniques incorporate the following main categories (but not limited to).

1. **Generation of macro-actions (macro-operators)**
2. **Elimination of actions**
3. **Problem decomposition**
4. **Selection of an efficient planning method**
5. **Learning heuristics**

Macro-actions (macro-operators[2]) represent the most common and studied type of knowledge in (classical) planning. A macro-action can be formally defined in the same way as a primitive action, but behaves like a sequence of primitive actions. Informally said, the macro-actions can be understood as 'shortcuts' in the state space. The idea of learning macro-actions is not very new (Dawson and Siklóssy 1977), but it is still a hot topic, because many techniques have been recently developed (Botea et al. 2005; Chrpa 2009; 2010b; Newton et al. 2007). In addition, an approach (Chrpa 2009; 2010b) removes primitive actions that are replaced by learned macro-actions.

Elimination of actions that are unnecessary for a solution can help planners to reduce the branching factor during searching the state space. State-of-the-art planning systems eliminate unreachable actions (Helmert 2006; Hoffmann and Nebel 2001) (i.e., actions that cannot be applicable at any point of the planning process). On the other hand, approach (Chrpa and Barták 2009) learns relations (called entanglements) between planning operators and initial or goal atoms and then prunes all actions (not necessarily unreachable) that violate the learned relations (entanglements).

Problem decomposition reflect the divide and conquer paradigm. Since development of some general learning technique for plan decomposition seems to be very hard, several planning systems (Hsu and Wah 2008) and learning techniques (Bibai et al. 2008) focus on consecutive goal satisfaction.

It is well known that some planning method performs well on some classes of planning problems, but on some classes does not (Roberts et al. 2008). The purpose of this type of learning techniques is, clearly, to select as good planning method as possible for a given planning problem. Obviously, soundness or completeness of such an approach goes

[2] macro-actions are macro-operators' instances

hand in hand with soundness or completeness of particular planning methods.

Heuristics learning (McCluskey 1987) comes from the need of improving the planning process by reducing the number of unpromising states which the planner explores. Soundness of the planning process is obviously not affected. Completeness depends on the underlying method used for the search. For instance, if A*, which is complete, is used, then completeness of the planning process is not affected.

Theoretical aspects of learning techniques

This section addresses theoretical aspects of using learning techniques such as macro-actions or action elimination in (classical) planning. Knowledge gained by these learning techniques is used to reformulate planning problems. Therefore, we will handle this knowledge from a reformulation point of view (formally defined below). We will also focus on investigation how such a reformulation may affect the planning process with respect to its soundness and completeness. Moreover, we discuss complexity of checking whether reformulation affects completeness of particular planning problems.

Reformulation of planning problems

Certain sorts of additional knowledge such as macro-actions or elimination of actions must be somehow encoded in the planning problems. It means that the original planning problems are reformulated, then the planner is applied on the reformulated problems. Gathered solutions of the reformulated problems are reformulated back to be the solutions of the original problems. It is formally defined below.

Definition 5. *Let $PROBS$ be a set of planning problems. Let $PLANS = \{\pi \mid \pi \, is \, a \, solution \, of \, any \; \Pi \in PROBS\}$ be a set of plans. A **reformulation scheme for planning** is a pair of functions $(probref, planref)$ defined in the following way:*

- $probref : PROBS \rightarrow PROBS$ *is a* ***problem reformulation function***
- $planref : PLANS \rightarrow PLANS$ *is a* ***plan reformulation function***

*The reformulation scheme for planning $(probref, planref)$ is **sound** if the following condition holds.*

- *If $\pi' \in PLANS$ is a solution of $probref(\Pi) \in PROBS$, then $planref(\pi')$ is a solution of $\Pi \in PROBS$.*

*The reformulation scheme for planning $(probref, planref)$ is **complete** if it is sound and the following condition holds.*

- *If $\Pi \in PROBS$ is solvable (i.e., there exists a solution of Π), then $probref(\Pi) \in PROBS$ is also solvable.*

■

We assume that soundness and completeness of the reformulation scheme for planning, like in the previous cases, are also incorporated with a sound (resp. complete) planning method. It means that soundness or completeness of the planning method holds for every problem from $PROBS$ (see the above definition). The whole process how to use a learned reformulation scheme for planning $(probref, planref)$ in solving a given planning problem Π:

1. Run the planner for the reformulated problem $probref(\Pi)$
2. If π' is a solution of $probref(\Pi)$, then return $planref(\pi')$ as a solution of Π. Otherwise return no solution.

The soundness of the reformulation scheme for planning ensures (in connection with a sound planning method) that if we use the above process we cannot solve unsolvable problem or we cannot provide an invalid solution. The completeness of the reformulation scheme for planning ensures (in connection with a complete planning method) that if we use the above process we solve every solvable planning problem.

Now, it remains to discuss how soundness and completeness are affected, if we compose two (or more) reformulation schemes for planning.

Proposition 1. *Let $(probref, planref)$ and $(probref', planref')$ be sound (resp. complete) reformulation schemes for planning. Then, $(probref' \circ probref, planref \circ planref')$ be a sound (resp. complete) reformulation scheme for planning.*

Proof. It is obvious from the definition 5 that $(probref' \circ probref, planref \circ planref')$ is a reformulation scheme for planning. If $(probref, planref)$ is sound (from the assumption), then we know that for an arbitrary planning problem Π holds that if π' is a solution of $probref(\Pi)$, then $planref(\pi')$ is a solution of Π. It similarly holds also for $(probref', planref')$. Then, we can see that if π'' is a solution of $probref'(probref(\Pi))$, then $planref'(\pi'')$ is a solution of $probref(\Pi)$, and then $planref(planref'(\pi''))$ is a solution of Π. Now, we proved that $(probref' \circ probref, planref \circ planref')$ is sound. If $(probref, planref)$ and $(probref', planref')$ are complete (from the assumption), then if any planning problem Π is solvable (i.e., there exists a solution of Π), then $probref(\Pi)$ is also solvable, and then $probref'(probref(\Pi))$ is also solvable. Now, we have proved that $(probref' \circ probref, planref \circ planref')$ is complete.

□

Eliminating actions

Eliminating actions as mentioned before is one sort of knowledge for improving the planning process. This sort of knowledge can be represented by the reformulation scheme of planning (see definition 5).

Definition 6. *Let $(elim, id)$ be a reformulation scheme for planning. Let id be a plan reformulation function such that for every plan π $id(\pi) = \pi$ (identity function). Let $elim$ be a problem reformulation function such that for every planning problem $\Pi = \langle P, S, A, \gamma, I, G \rangle$ and*

$\Pi' = \langle P, S, A', \gamma', I, G\rangle$ *such that* $elim(\Pi) = \Pi'$, $A' \subseteq A$ *and* γ' *is defined according to the definition 1. Then, we say that* $(elim, id)$ *represents an* ***action eliminating scheme.*** ■

The above definition formally follow the straightforward meaning what action eliminating stands for. Soundness of the action eliminating scheme can be proved is the following way, of course, we assume that we use a sound planning method for solving the (reformulated) planning problems.

Proposition 2. *Let* $(elim, id)$ *be an action eliminating scheme. Then,* $(elim, id)$ *is a sound reformulation scheme for planning.*

Proof. From the definition 6 we know that $(elim, id)$ is a reformulation scheme for planning. Now, we need to prove that if we solve (by a sound planning method) the (reformulated) planning problem $elim(\Pi)$, then the solution of $elim(\Pi)$ is also a solution of Π (we know that $id(\pi) = \pi$, so if π is a solution of $elim(\Pi)$, then π is a solution of Π). From the definition 6 we know that every solution of $elim(\Pi)$ (for any planning problem Π) contains only actions defined in the (original) planning problem Π. Hence, it is obvious that any solution of $elim(\Pi)$ is also a solution of Π.

□

Regarding completeness of the action eliminating scheme we can imagine such a problem reformulation function that remove all the actions for the (original) planning problems. It is obvious that such a reformulation can provide unsolvable problem even though the original one is solvable. Clearly, it means that the action eliminating scheme is not complete in general.

Speaking in the language of the graph theory the problem reformulation by the action eliminating scheme is about removing transitions (edges) from the state transition system that represents the original planning problem. Straightforwardly, the completeness of the action eliminating scheme means that after removing transitions (edges) from the state transition system (representing the original problem) there must remain at least one path from the initial state to at least one of the goal states.

For further investigation of completeness of the action eliminating scheme we explore landmarks (Hoffmann, Porteous, and Sebastia 2004) - facts that must be true at some point of the planning process.

Definition 7. *Let* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *be a planning problem. If for every solution* $\pi = \langle a_1, \ldots, a_n\rangle$ *of* Π $p \in \gamma(I, \langle a_1, \ldots, a_i\rangle), 0 \leq i \leq n$, *then* p *is a* ***landmark*** *(in* Π*).* ■

Landmarks provide us with advice about what actions can be essential for a given planning problem. The following proposition (formally described below) deals with the situation when the action eliminating scheme eliminate actions (from some problem) which provide landmark(s). In that case the action eliminating scheme is not complete.

Proposition 3. *Let* $(elim, id)$ *be an action eliminating scheme. Let* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *and* $\Pi' = \langle P, S, A', \gamma', I, G\rangle$ *be planning problems such that* $elim(\Pi) = \Pi'$. *Without loss of generality we assume that there exists a solution of* Π. *If any* $p \in \bigcup_{a\in A\setminus A'} eff^+(a) \setminus \left(\bigcup_{a'\in A'} eff^+(a') \cup I\right)$ *is a landmark (in* Π*), then* $(elim, id)$ *is not complete.*

Proof. Let p be a landmark (in Π) and $p \in \bigcup_{a\in A\setminus A'} eff^+(a) \setminus \left(\bigcup_{a'\in A'} eff^+(a') \cup I\right)$ (i.e., p can be obtained only by performing some action removed from the original problem). From the definition 7 we know that p must be presented in some state during the execution of any solution of Π (i.e., every path from the initial state to one of the goal states in the corresponding state transition system must go through a state (or states) containing p). From the assumption we simply get that $p \notin \bigcup_{a'\in A'} eff^+(a') \cup I$. It results in the fact that a state containing p cannot be reached in Π' (i.e., there is no path from the initial state to one of the states containing p in the corresponding state transition system). It is clear that if Π' is solvable, then p is not a landmark (in Π'). With respect to id we know that a solution of Π' is also a solution Π. According to this we get that p is not a landmark (in Π) which is a contradiction with respect to the assumptions. It means that there does not exist any solution of Π', but according to the assumption there exists a solution of Π. It proves that $(elim, id)$ is not complete.

□

Corollary 1. *Let* $(elim, id)$ *be an action eliminating scheme and* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *be a planning problem having a solution. Let* $a \in A$ *be an action such that* $p \in eff^+(a)$, $p \notin \bigcup_{\overline{a}\in A, \overline{a}\neq a} eff^+(\overline{a}) \cup I$. *If* p *is a landmark in* Π *and* $a \notin A'$ *where* A' *is a set of actions defined in* $elim(\Pi)$, *then* $(elim, id)$ *is not complete.*

Proof. It follows directly from proposition 3. □

The last corollary suggests a hypothesis that deciding whether a single action must not be eliminated (such an action is called an action landmark) is in general as hard as deciding a landmark, which has been proved to be PSPACE-complete (Hoffmann, Porteous, and Sebastia 2004), thus hard as planning itself. However, the formal proof of the hypothesis is a subject of ongoing work.

State-of-the-art planning systems usually prune many actions that are unreachable from the initial state, i.e., such actions are not applicable in any point of the planning process. The following proposition formally shows us that removing unreachable actions from the planning problem does not affect its solvability, i.e., such an action eliminating scheme is complete.

Proposition 4. *Let* $(elim, id)$ *be an action eliminating scheme. Let* $\Pi = \langle P, S, A, \gamma, I, G\rangle$ *and* $\Pi' = \langle P, S, A', \gamma', I, G\rangle$ *be planning problems such that* $elim(\Pi) = \Pi'$. *If for every* $a \in A \setminus A'$ *holds that there does not exist any solution for a planning problem* $\Pi_a = \langle P, S, A, \gamma, I, pre(a)\rangle$, *then* $(elim, id)$ *is complete.*

Proof. We can show that any solution of Π is a solution of Π'. By a simple consideration we can see that no solution of

Π contains actions from $A \setminus A'$, because from the assumption we know that preconditions of the actions (from $A \setminus A'$) cannot be satisfied at any point of the planning process (i.e., there is no path in the corresponding state transition system from the initial state to any state containing all the precondition atoms of these actions). It results in the fact that every solution of Π must be also a solution of Π'. Then, we know that if Π has a solution, then Π' also has a solution (the same one). So, it means that $(elim, id)$ is complete.

□

Obviously, in general deciding reachability of an action is as hard as planning itself (PSPACE-complete) because we have to solve a planning problem such as Π_a (defined in proposition 4). Despite the high complexity many unreachable actions can be detected and pruned in a polynomial time, for instance when reaching a fixed point of Planning Graph (Blum and Furst 1997) all actions that are not presented in the last action layer are unreachable.

Macro-actions

As mentioned before in the text macro-actions are defined in the same way as 'normal' actions, but the result of application of a macro-action is the same as the result of consecutive application of a sequence of 'normal' actions.

Definition 8. *Let $\langle a_1, \ldots, a_k \rangle$ be a sequence of actions. Let $a_{1,\ldots,k}$ be an action. Assume that for every state s and transition function γ holds: $\gamma(s, a_{1,\ldots,k}) = \gamma(s, \langle a_1, \ldots, a_k \rangle)$ or both $\gamma(s, a_{1,\ldots,k})$ and $\gamma(s, \langle a_1, \ldots, a_k \rangle)$ are undefined. Then, we say that $a_{1,\ldots,k}$ is a **macro-action** over the sequence $\langle a_1, \ldots, a_k \rangle$.* ■

Macro-actions can be understood as 'shortcuts' in the state space (state transition system). In terms of reformulation scheme for planning, macro-actions are added into the planning problems and solutions of reformulated problems (with macro-actions) must be reformulated back (macro-actions are unfolded into the sequences of primitive actions).

Definition 9. *Let $(macro, unfold)$ be a reformulation scheme for planning. Let $macro$ be a problem reformulation function such that for every planning problem $\Pi = \langle P, S, A, \gamma, I, G \rangle$ and $\Pi' = \langle P, S, A \cup MA, \gamma', I, G \rangle$ such that $macro(\Pi) = \Pi'$, for every $a_{1,\ldots,k} \in MA$ such that $a_{1,\ldots,k}$ is a macro-action over the sequence of actions $\langle a_1, \ldots, a_k \rangle$ ($a_1, \ldots a_k \in A$) and $\gamma' \supseteq \gamma$ (defined acc. to def. 1). Let $unfold$ be a plan reformulation function such that for every π' and π such that $unfold(\pi') = \pi$ it holds that every macro-action (from MA) in π' is replaced by the corresponding sequence of primitive actions (from A) in π. Then, we say that $(macro, unfold)$ represents a **macro-action scheme**.* ■

Soundness and completeness of the macro-action scheme will be proved in the next proposition. Of course, we consider a sound (resp. complete) planning method.

Proposition 5. *Let $(macro, unfold)$ be a macro-action scheme. Then, $(macro, unfold)$ is sound and complete reformulation scheme for planning.*

Proof. Let π' be an arbitrary solution of the reformulated planning problem $macro(\Pi)$. It directly implies from the definitions 9 (definition of $unfold$ function) and 8 (properties of transition function) that $unfold(\pi')$ is a solution of Π. It means that $(macro, unfold)$ is sound. To prove completeness of $(macro, unfold)$ we have to prove that if there exists a solution of Π, then there also exists a solution of $macro(\Pi)$. We can see from the definition 9 that every solution of Π is also a solution of $macro(\Pi)$ (we do not remove or change any action defined in the original problem Π, we only add macro-actions). It proves that $(macro, unfold)$ is complete.

□

Soundness and completeness of the macro-action scheme show us a main advantage of using macro-actions. On the other hand adding macro-actions causes an increase of the branching factor in the state space (we only adding transition to corresponding state transition systems).

In addition, when macro-actions are added to the problem some primitive actions (included in the new macro-actions) might be removed from the problem (Chrpa 2009; 2010b). It can be understood, in our terminology, as a composition of the macro-action and action eliminating schemes. Soundness of such a composition can be proved in the following way.

Proposition 6. *Let $(macro, unfold)$ be a macro-action scheme. Let $(elim, id)$ be an action eliminating scheme. Then, $(elim \circ macro, unfold)$ is sound reformulation scheme for planning.*

Proof. Obviously, we know that $(macro, unfold)$ and $(elim, id)$ are sound reformulation schemes for planning (see the proposition 5 and 2. A composition of sound reformulation schemes for planning (see the proposition 1) is also a sound reformulation scheme for planning. It means that $(elim \circ macro, unfold \circ id)$ is sound. From the definition 6 we know that id is an identity function. Hence, we can see that $(elim \circ macro, unfold)$ is sound reformulation scheme for planning.

□

The question is how removing primitive actions affects the completeness of such a reformulation scheme for planning. Without loss of generality we assume in the following text that all macro-actions are assembled from two primitive actions (extension for more primitive actions is quite straightforward).

Macro-actions are 'shortcuts' in the state-space. Removing the primitive actions causes that the intermediate state might become unreachable or the goal might become unreachable from the intermediate state. If there exist inverse actions for the removed primitive ones (for illustration, see Figure 1), then the intermediate state remains reachable, thus the completeness is not affected. If the intermediate state is an initial state or a single goal state, then the completeness is broken. If there is another action applicable in the intermediate state, then to ensure completeness there must exist an alternative action which is applicable in one of the states connected by macro-action and its application results in the

same state as the application of the previously mentioned action in the intermediate state. Similarly we must find an alternative actions for actions which leads to the intermediate state (for illustration, see Figure 2). The following proposition formalizes the above idea.

Proposition 7. *Let $\Pi = \langle P, S, A, \gamma, I, G\rangle$ be a planning problem. Without loss of generality we assume that $a_1, a_2 \in A$ and $a_{1,2} \notin A$. Let $a_{1,2}$ be a macro-action over the sequence $\langle a_1, a_2\rangle$. Let $A^- = \{a_1, a_2\}$ be a set of actions. We assume that one the following conditions holds for every triple of states $s_0, s_1, s_2 \in S$ such that $\gamma(s_0, a_1) = s_1$ and $\gamma(s_1, a_2) = s_2$.*

1. *There exist actions $a'_1, a'_2 \in A$ such that $\{a'_1, a'_2\} \cap A^- = \emptyset$, $\gamma(s_2, a'_2) = s_1$ and $\gamma(s_1, a'_1) = s_0$.*
2. *$s_1 \notin I$, if $G \subseteq s_1$, then $G \subseteq s_0$ or $G \subseteq s_2$ and for every $a \in A \setminus A^-$ and $s \in S \setminus \{s_1, s_2\}$ such that $\gamma(s_1, a) = s$ (resp. $\gamma(s, a) = s_1$, $s \neq s_0$) it holds that $s = s_0$ or there exists an action $a' \in A \setminus A^-$ such that $\gamma(s_0, a') = s$ or $\gamma(s_2, a') = s$ (resp. $\gamma(s, a') = s_0$, $s \neq s_0$ or $\gamma(s, a') = s_2$).*

Let $\Pi' = \langle P, S, (A \cup \{a_{1,2}\}) \setminus A^-, \gamma', I, G\rangle$ be a planning problem (γ' is defined according to the definition 1). If Π has a solution, then Π' has also a solution.

Proof. From the definition 8 we know that $\gamma'(s_0, a_{1,2}) = s_2$.

If the condition (1) is satisfied, then we know that the states s_0, s_1 and s_2 are connected in Π' (i.e., there exists a path in the corresponding state transition system between the states). For a deeper insight, see figure 1. Now, it is clear that if Π has a solution, then Π' has also a solution.

If the condition (2) is satisfied, then we show that if the macro-action $a_{1,2}$ is added, then s_1 does not have to be reached at any point of searching for the solution (i.e., a path in the corresponding state transition system from the initial to one of the goal states that does not visit s_1). First, we know that s_1 is not the initial state and if s_1 is a goal state, then s_0 or s_2 is also the goal state. It means that we do not need to start or finish in s_1. Second, we have to prove that s_1 does not have to be visited (i.e., there exists a path in the corresponding state transition system that do not go through s_1). From the second part of the condition (2) we can see that if some state s is reachable by performing a single action from s_1, then s is also reachable from s_0 or s_2 (by performing a single action). If $s = s_0$, then the reachability of s_2 is not affected, because it is possible to go from s_1 through s_0 to s_2. For a deeper insight, see figure 2. Similarly, it holds also for situations where s_1 is reachable from s (by performing a single action). By a simple consideration, we can see that s_1 can be 'bypassed' (we can go through s_0 and/or s_2). Now, it is clear that if Π has a solution, then Π' has also a solution. □

The conditions (1,2) are quite strict. Generally, the proposition holds as well even if we replace (single) actions a'_1, a'_2 and a' by sequences of actions. However, considering sequences of actions rather than single actions will more likely raise the complexity (up to PSPACE-complete) of checking those conditions.

Discussion

We showed that reformulation of problems by eliminating actions and/or by adding macro-action does not affect soundness, which is an unsurprising result. It means that if we solve reformulated problems (added macro-actions or/and eliminated some actions), then we get (by a simple conversion) solutions of the original problems. It is advantageous in practice, because such solutions can be usually found in significantly less running time. Completeness on the other hand means that if we cannot solve reformulated problems, then we also cannot solve the original problems. Completeness has been proved without any restrictions only for adding macro-actions, which is also not very surprising. It is one of the reasons why macro-action learning (Botea et al. 2005; Chrpa 2010b; Newton et al. 2007) is the most studied type of the learning techniques for planning. Regarding the elimination of unreachable actions (performed during grounding phase (Helmert 2006; Hoffmann and Nebel 2001)) it is complete as well. Other methods (Chrpa and Barták 2009) that eliminate also reachable actions are not ensuring completeness in general. We raised a hypothesis that checking whether an action cannot be eliminated has the same complexity as checking whether an atom is a landmark (Hoffmann, Porteous, and Sebastia 2004), which is PSPACE-complete. Composition of macro-action and elimination of actions, in terms, that we eliminate only primitive actions that can be replaced by learned macro-actions is studied in (Chrpa 2010b). In this paper we showed some cases when the primitive actions can be removed without losing completeness. It could serve as an inspiration for developing techniques that can verify (in a polynomial time) whether a certain reformulation of the problem (including removing primitive operators replaced by macro-operators) may affect completeness or not.

From the practical point of view the learning techniques are evaluated in terms how they influence the planning process by comparing the running times and quality of solutions. Many of the learning techniques require training plans on which they can learn knowledge. Training plans are usually very simple and their number is quite low. It is because gathering training plans could be very expensive (remember the PSPACE-completeness of classical planning). Despite potential loss of completeness learning techniques gained very good results (Chrpa 2010b; Chrpa and Barták 2009; Chrpa 2010a) and only few problems became unsolvable after their reformulation, because many problems differ only by the number of objects and not by the types of initial or goal situations. Hopefully, the insolvability of the problem can be often detected by the state-of-the-art planning systems in a little time. However, if the problem reformulation breaks completeness and we do not get a solution of the reformulated problem, then it is necessary to try to solve also the original one.

Reformulation scheme for planning, which can be obtained by using and eventually combining various types of learning techniques, transforms original planning problems for the purpose of improving efficiency of the planning process. Obviously, success or failure of a given reformulation scheme depends also on a particular planning problem and

Figure 1: Replacing primitive actions (a) by a macro-action (b). Removed primitive actions are visualized by a dotted line in (b).

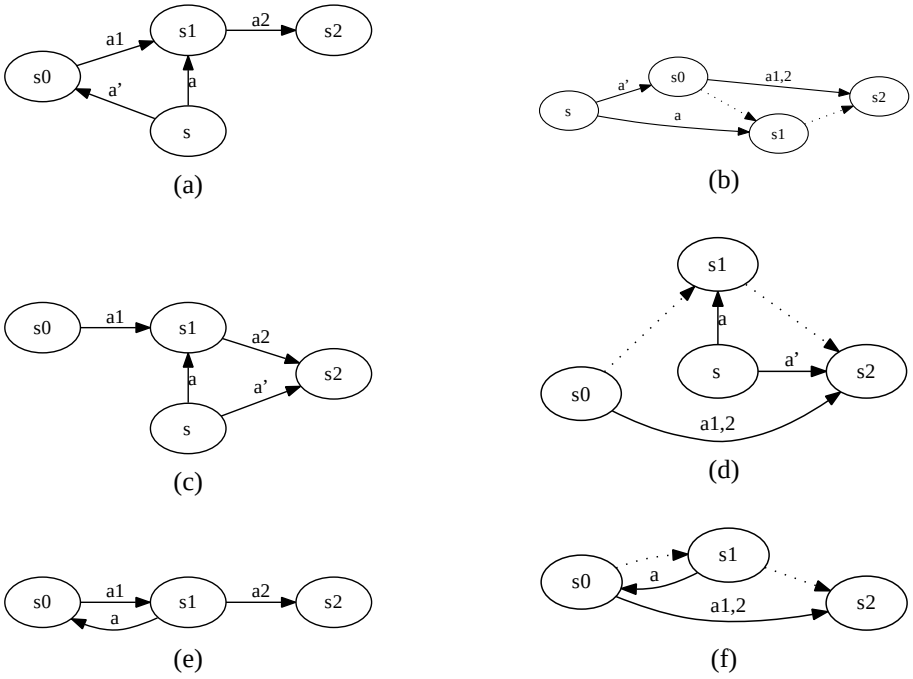


Figure 2: Replacing primitive actions a_1 and a_2 (a,c,e) by a macro-action $a_{1,2}$ (b,d,f) - it corresponds to situations according to the condition (2) - note that we obtain symmetrical situations to (a)-(d) if a and a' are in the reverse direction. Removed primitive actions are visualized by dotted lines in (b,d,f).

planning technique. Therefore, an evaluation how good reformulation scheme is supposed to be, which is an essential part of systems using learning techniques to produce reformulation schemes, is a very difficult task. For instance, in (Chrpa 2010b) after macro-operators are generated we check whether every macro-operator replaces at least one primitive operator. If it holds then we have found reformulation scheme, otherwise we modify parameters and run the macro-operator generator again. Such type of evaluation does not take into account heterogeneity of planning techniques which means that in some cases results can be much worse for reformulated problems. Wizard (Newton et al. 2007) evaluates reformulation schemes in such a way that it solves several reformulated problems (in a given domain) by a given planner. Even though this evaluation is more precise it is very time consuming. Work (Roberts et al. 2008) presents an approach based on machine learning which is able to estimate behavior of a given planner for a given problem. It seems to be a very promising option for evaluating reformulation schemes because this approach is fast and quite precise.

Assuming that we are able to exactly evaluate reformulation schemes in such a way that we can compare them. An interesting question arises targeting an existence the best reformulate scheme and ability to find it. There can be seen a parallel with compiler optimization. The best reformulation scheme is defined in such a way that every problem is reformulated such that the reformulated problem contains only one (macro-)action that solves the problem. It requires that a (learning) technique which provides such a reformulation scheme must in fact know a solution for every planning problem. It simply means that such a technique must solve the problem before reformulating it which is rather impractical. Reformulation scheme can be acquired i) directly from planning domain (or problem) analysis or ii) by exploring training plans, solution of simpler problems in a given planning domain. Therefore, reformulation scheme should be in general form (e.g., macro-operators are generalized macro-actions). Another common aspect of learning techniques, which produce reformulation schemes, is

domain-independence (even though they produce domain-dependent knowledge). We can ask how good can be reformulation scheme produced by learning techniques following the above rules (generalization, domain-independence) ? Where are the limits ?

It is also good to mention that using learning techniques for optimal planning is not very common. It is that even if we use an optimal planning method for solving reformulated problems, then we do not have to get optimal solutions. Obviously, eliminating only unreachable actions does not affect optimality. Elimination of any (reachable) action may cause breaking an optimal path in a corresponding state transition system. Regarding macro-actions we can ensure the optimality by extending the planning problem definition by action costs. Action cost assigns a (positive) value to every action. Optimality in planning with action costs means that we are looking for a solution (plan) with the lowest cost (a sum of action costs of all actions in the plan). Straightforwardly, action cost of a macro-action equals to a sum of action costs of the primitive actions (from which the macro-action is assembled).

Conclusions

In this paper we addressed the theoretical aspects of the most common learning techniques for (classical) planning used for reformulation of planning problems. We proved soundness of all presented reformulation schemes (macro-actions, action elimination). It means that if we solve reformulated problems, then we get solutions that after modification (by plan reformulation function) become solutions for the original problems. We also addressed completeness which remains untouched only in special cases.

Completeness is good for situations where the reformulated problem is unsolvable thus we do not need to try to solve the original problem. Otherwise, it must be checked whether completeness is broken or not. We raised a hypothesis (not formally proved yet) that deciding whether a single action must not be eliminated (to avoid breaking completeness) is PSPACE-complete which it is quite impractical. Regarding eliminating primitive actions replaced by macro-action we believe that according to proposition 7 we will be able to find a method for checking whether the removal of primitive actions does not affect completeness running in polynomial time.

In the future, we should continue studying theoretical aspect of learning techniques in planning. It can serve as an inspiration both for developing new (complete) learning techniques for planning and methods for verifying completeness of reformulated or otherwise modified problems.

References

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2008. Dae: Planning as artificial evolution (deterministic part). In *Booklet of the competing planners in the sixth International Planning Competition (IPC-6)*.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research (JAIR)* 24:581–621.

Chrpa, L., and Barták, R. 2009. Reformulating planning problems by eliminating unpromising actions. In *Proceedings of SARA 2009*, 50–57.

Chrpa, L. 2009. *Learning for Classical Planning*. Ph.D. Dissertation, Charles University in Prague.

Chrpa, L. 2010a. Combining learning techniques for classical planning: Macro-operators and entanglements. In *Proceedings of ICTAI*, volume 2, 79–86.

Chrpa, L. 2010b. Generation of macro-operators via investigation of action dependencies in plans. *Knowledge Engineering Review* 25(3):281–297.

Coles, A., and Smith, K. A. 2007. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research (JAIR)* 28:119–156.

Dawson, C., and Siklóssy, L. 1977. The role of preprocessing in problem solving systems. In *Proceedings of IJCAI 1977*, 465–471.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning, theory and practice*. Morgan Kaufmann Publishers.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research (JAIR)* 22:215–278.

Hsu, C.-W., and Wah, B. W. 2008. The sgplan planning system in ipc-6. In *The 6th International Planning Competition (IPC-6)*.

McCluskey, T. L. 1987. Combining weak learning heuristics in general problem solvers. In *Proceedings of IJCAI*, 331–333.

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *Proceedings of ICAPS 2007*, 256–263.

Roberts, M.; Howe, A. E.; Wilson, B.; and desJardins, M. 2008. What makes planners predictable? In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia*, 288–295.

Heuristically Guided Constraint Satisfaction for Planning

Mark Judge and **Derek Long**
Department of Computer and Information Sciences
University of Strathclyde, Glasgow

Abstract

Constraint satisfaction techniques have been used in Artificial Intelligence planning for many years. As with the original planning problem formulation, the CSP model's complexity can lead to a very large search space for all but the simplest of problems. Without additional guidance, the CSP solver will rely on standard CSP heuristics and search methods. Better use can be made of the CSP framework if the search is informed by the structure of the planning problem. This paper discusses a goal-centric, variable / value selection heuristic method of guiding the search for a solution to a constraint encoding of classical planning problems. Also, meta-CSP methods that provide further propagation are introduced. The prototype uses an extensional encoding of the problem, goal ordering, the variable / value ordering heuristic and the meta-CSP variables. Preliminary results on a number of test domains are presented. These show an improvement over the same encoding without heuristic guidance.

Introduction

Artificial Intelligence (AI) Planning (Russell et al. 2003) seeks to find a set of actions that transform a given initial state into a partially specified goal state. The Planning Domain Description Language (PDDL) (Ghallab et al. 1998) is the standard representation used by the research community to describe planning problems. An alternative approach is to reformulate the problem as a Constraint Satisfaction Problem (CSP) (Dechter 2003).

Using a CSP formulation allows planning problems to take advantage of Constraint Programming (CP) techniques, including enhanced *propagation* machinery and better *pruning* mechanisms. These are techniques not fully exploited by the planning community. Instead, many successful contemporary planners[1] rely on the use of heuristics based on a relaxed version of the original problem.

Noting that CSP based planners do not yet match the performance of their state of the art counterparts raises the question: Why not? Firstly, for most interesting problems, the size of the search space can become extremely large, growing exponentially with the problem size. Secondly, with a large problem comes an increased plan length (*horizon*). This large horizon greatly reduces the impact of the propagation resulting from the goal-state constraints. That is, with a small horizon, a CSP solver benefits from inferences made as a result of the constraints placed on the goal-state variables. These inferred values result in new constraints which, in turn, provide further pruning of the search space. In this manner, in an ideal situation, the power of the declarative CSP paradigm becomes clear; a solution can be inferred without the use of search. It has been shown (Vidal and Geffner 2006) that it is possible to solve simple planning problems using inference alone. The authors describe a new version of the partial-order CPT planner, eCPT, that makes use of landmarks (Porteous and Sebastia 2004) and additional distance constraints (van Beek and Chen 1999).

This paper presents a technique with a similar theme: Increased use of inference, with much less reliance on uninformed search. The aim is to use a given planning problem's structure to inform the variable and value selection method in the CSP encoding of that problem. This is a goal-centric approach which attempts to satisfy each goal variable's value as early in the plan as possible. By doing so, a series of intermediate horizons are introduced. These provide a bridge to the extra propagation required to find a faster solution.

Additionally, in order to maintain the value of newly achieved goals, this work introduces a meta-CSP approach to locking the relevant variable's value until the end of the plan. Here, too, extra propagation is the result.

Application of this algorithm to extensional CSP encodings of a number of planning problems has shown promising results for the production of sub-optimal plans.

The remainder of this paper is structured as follows. Firstly, a short introduction to planning as CSP is presented. The technical details of the goal-directed heuristic algorithm and the meta-CSP locking technique are then described, together with details of the prototype system used for testing. Preliminary results of the algorithm's implementation are shown, and finally conclusions are drawn, with the direction of future work indicated.

Planning as Constraint Satisfaction

Planning is a human decision making process which seeks to achieve a given outcome by using a set of predictable operations in sequence. AI planning attempts to automate this

[1] IPC-2011:http://www.plg.inf.uc3m.es/ipc2011-deterministic/Results?action=AttachFiledo= viewtarget=ipc2011-booklet.pdf

process. Informally, we can state that automated planning is a sequential decision making technique, the purpose of which is to provide a set of actions that transform a fully specified initial state into a given goal state.

In order to manage the complexity of the real world, AI planning restricts the information described and abstracts much of the unnecessary detail. Classical planning, also known as STRIPS planning (Fikes and Nilsson 1990), requires that the *state space* be finite and fully observable. It is assumed that only specified actions can change a state and that they do so instantaneously, with the resulting state being predictable.

Definition 1. *A STRIPS planning problem, P = (O,I,G), where O is a set of operators, I is a conjunction of fact literals describing the initial state, and G another conjuction of facts describing a partially specified goal state.*

Alternative solution techniques can be used if the planning problem is reformulated. One such approach sees the planning problem cast as a CSP.

Definition 2. *A constraint satisfaction problem, M, is a triple, (X,D,C), where X is a finite set of variables, $X = \{x_1, x_2, \ldots x_n\}$, D is a finite set of domains for those variables, $D = \{d_1, d_2, \ldots d_n\}$, and C is a set of constraints, $C = \{c_1, c_2, \ldots c_m\}$ where each constraint defines a predicate which is a relation over a particular subset of X.*

As with all software engineering, eliciting the requirements and formulating a full and correct problem specification are central to being able to find a solution to the original problem. When using CSPs for planning, we already have the problem specification, generally in PDDL. However, there are many ways in which to model the problem as a CSP, and it is well known that the modelling choices made are critical (Freuder 1999). It has been shown (Beacham et al. 2001) that the choices of model, search algorithm and heuristic interact and none of these decisions should be made independently.

In this work, the first step towards modelling the problem was to convert from PDDL into a representation based on SAS$^+$ (Backstrom 1992). By using intermediate output from the Fast Downward planner (Helmert 2006), it was possible to gain access to such a SAS$^+$ description. Basing the constraint model on this multi-valued representation is good modelling practice (Smith 2005) since it provides a smaller number of variables, each with larger domains.

Another important aspect of modelling is the way in which the constraints are expressed. For example, it is possible to use either a series of binary $\neq$ constraints or a global *allDifferent* constraint. Knowledge of the differing constraint propagation behaviours and costs is essential if best use is to be made of the CSP approach (Harvey, Kelsey, and Petrie 2003).

Based on the authors' empirical experience and on the results of recent research (Bartak and Toropila 2008b), it is clear that an *extensional* representation of constraints leads to a more efficient model of a given planning problem. For that reason, extensional, or *table* constraints were used in this work.

Finding Solutions to CSPs

It is possible to categorise CSP solution methods as being either *constructive* or *local* in nature. Local search methods (Clark et al. 1996) generally employ a hill-climbing strategy and are guided by a cost function. Moving over the space of completely instantiated variables by changing, say, one variable at a time (local change), the search stops either when the problem is solved (cost equals zero) or when no further improvement in cost is available (local minimum). This paper describes a system that makes use of a constructive approach (Tsang 1993) and, for that reason, the following discussion focuses on this area.

Constructive search attempts to iteratively extend a consistent partial assignment of the variables in the CSP. This continues until a consistent assignment of all variables is made. If the partial assignment proves inconsistent, the algorithm fails and backtracking takes place. The important parts of this procedure are the order in which the variables are selected, the order in which the values are tried on those variables, the method by which variable assignments are *propagated*, and the means by which the search procedure *backtracks*.

Looking first at the variable selection strategy, the ordering can be static or dynamic. The former specifies the order before search begins while the latter makes a decision based on the current state of the search. An example of a computationally inexpensive variable ordering heuristic is the *Minimum Remaining Values* (MRV) (Gent et al. 1996), or *first fail*, method. In this approach, the variable with the lowest number of remaining values is chosen. This results in the *branching factor* being minimised for the longest possible time. MRV has been shown to work well on a large number of CSPs (Dechter and Meiri 1994).

Having chosen a variable to instantiate, it is now necessary to assign a value to that variable. A good choice of value will reduce the amount of backtracking required. It should be noted that, if the correct choice of value is made at each point of the search, it is possible to reach a solution with no backtracks. In contrast to variable ordering, the strategy for value selection is to choose the value that is most likely to succeed, since failure would cause the search to backtrack. An example of a value ordering heuristic is the *min-conflict* heuristic (Dechter 2003). In this method, the values are ordered based on the number of conflicts that they are involved in with the unassigned variables.

With a variable selected and a value assigned, *inference* can be used to construct new constraints based on this latest assignment. There are many propagation methods available, including *forward checking* and *k-consistency* strategies (Freuder 1978). One commonly used technique is *arc consistency* (Mackworth 1977). Here, the algorithm guarantees that any allowable value in a variable's domain is consistent with a permitted value in the domain of any other single variable.

If the propagation step finds an inconsistency, it is necessary to backtrack. A simple backtracking technique steps back over the last made assignment and tries an alternative value from that variable's domain. More advanced methods can pinpoint the variable responsible for causing the failure

and will backtrack accordingly. These include *backmarking, backjumping and conflict-directed backjumping* (CBJ) (Prosser 1993).

CSP approaches to planning

CSP techniques have been used in planning for many years, although early systems made use of constraint methods only to solve part of the problem (Stefik 1981), (Joslin and Pollack 1995) and (Goldman et al. 2000). That is, subproblems were posted and solved as CSPs, with the result returned for use by traditional planning machinery.

Systems that completely encode the planning problem as a CSP include *CPlan* (van Beek and Chen 1999), *GP−CSP* (Do and Kambhampati 2001) and *Csp−Plan* (Lopez and Bacchus 2003). The first of these makes use of various types of constraint, including *distance constraints* and *capacity constraints*, although these are manually specified by the user. GP−CSP encodes the planning graph as a *dynamic CSP*, whereas Csp−Plan avoids the planning graph and instead exploits the CSP encoding by adding new constraints and removing single valued variables.

Another system (Gregory, Long, and Fox 2007), based on a *meta-csp* formulation of the planning problem performed well when compared to state of the art SAT (Kautz and Selman 1992) planners.

Reformulation of a number of earlier CSP-based encodings using table constraints has been shown (Bartak and Toropila 2008b) to be effective. Also, the inclusion of *symmetry breaking*, *singleton consistency* and *lifting* (Bartak and Toropila 2008a) improves CSP planner performance.

Recent CSP planning systems include those based on *timelines* (Verfaillie and Pralet 2008), (Cesta and Fratini 2008), (Bartak 2011) and one that makes use of *dominance constraints* (Gregory, Fox, and Long 2010).

Technical Details

Whilst recasting a planning problem as a CSP does allow CP specific tools to be used, generally the solution process does not make use of the structure of the original planning problem. That is, the planning problem is converted into a CSP and the CSP is solved using CP methods. In order to make better use of the information implicit in the original problem, it is helpful to note some sources of *leverage* in planning. For example, a time bound will force out useless actions from the plan, thereby aiding progress towards a solution. Similarly, the consumption of a resource may prevent goal achievement, making it necessary to undo a previous assignment.

With this in mind, is it possible to employ similar forms of leverage in the CSP paradigm? The following sections describe two techniques that attempt to do so. The first, a goal centric variable and value heuristic, introduces a series of time bounds, allowing for more propagation and the removal of unhelpful actions. The second, a meta-CSP technique, locks goal values once these are achieved.

Preliminaries

The test configuration used in this work makes use of a heuristic estimate to gain an initial seed plan length. Empirical testing has shown that this generous CSP seed plan length works on all of the test domains. However, further work in this area may improve the accuracy of the estimation and consequently could lead to improved efficiency.

Due to the goal centric nature of the following procedure, it is first necessary to order the goals. Such a goal ordering is achieved by breaking all cycles in the original planning problem's *causal graph* (Helmert 2004) and sorting topologically.

With a seed plan length known, the CSP can be constructed. Referring to Figure 1, the partial *solution matrix* shows some of the CSP (state) variables (columns 1 to 3) and their associated domains for a small logistics problem example. The final column shows the action variables, also with their domains. At this stage, only the variables and their domains are represented. The next step is to define and apply the constraints. The constraints represent the grounded actions derived from the planning problem. That is, the means by which the variables change value. Hence, the action in action slot one operates on the variables in the initial state to produce row two in the matrix, and so on.

3	5	2	. . .	Ac1:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac2:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac3:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac4:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac5:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac6:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac7:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac8:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac9:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac10:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac11:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac12:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac13:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac14:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac15:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac16:	_{1..109}
.	.	.	.	.	.
.	.	.	.	.	.
_{0..8}	_{0..8}	_{0..5}	. . .	Ac38:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac39:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac40:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac41:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac42:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac43:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac44:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac45:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac46:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac47:	_{1..109}
4	4	0	. . .		

Figure 1: Partial solution matrix before constraints applied.

Figure 2 shows the solution matrix following application of the constraints. From this it can be seen that the constraint solver has used inference and propagation to reduce the domains of the three state variables at layers 1 and 2, and the first two variables at layers 46 and 47. Also, the domains of action slots 1 through 5 and 43 through 47 are reduced. Note should also be made of the reduction by one in the size of all variables' domains. The solver recognises that the SAS$^+$ "none of those" value is never used and simply removes it from the range of available values.

Although there has been some constraint propagation, with associated domain reduction, there is not nearly enough to solve the problem; the horizon is too large. The following heuristic attempts to address this.

Algorithm 1: solveMatrix(uplim,lowlim,smax)

Input: uplim (The upper limit), lowlim (The lower limit), smax (The solution matrix)
Output: smax (The solution matrix)

```
1  goalset ⟵ smax[uplim,(1..numofvars)];
2  for s ← uplim to lowlim do
3      q ⟵ (s − 1);
4      foreach goalvar in goalset do
5          if action in layer q then
6              if goalvar supported by action in layer q then
7                  if goalvar is a g-state goal then
8                      lockGoalvar(s,g-state)
9              else
10                 if goalvar supported by any row above then
11                     if goalvar is a g-state goal then
12                         lockGoalvar(s,g-state)
13                 else
14                     alloc ⟵ 0;
15                     acset ⟵ getSugActs(goalvar);
16                     foreach act ∈ acset do
17                         findSlotForAction(act,alloc,level);
18                         if alloc == 1 then
19                             lolev ⟵ (level − 1);
20                             solveMatrix(level,lolev,smax);
21                             break;
22         else
23             if goalvar supported by any row above then
24                 if goalvar is a g-state goal then
25                     lockGoalvar(s,g-state)
26             else
27                 alloc ⟵ 0;
28                 acset ⟵ getSugActs(goalvar);
29                 foreach act ∈ acset do
30                     findSlotForAction(act,alloc,level);
31                     if alloc == 1 then
32                         lolev ⟵ (level − 1);
33                         solveMatrix(level,lolev,smax);
34                         break;
```

A Goal-directed CSP Heuristic

Using the appropriately ordered goals as a starting point, the first goal is selected and the algorithm (Algorithm 1) regresses up through the solution matrix, attempting to find a layer in which the current goal is not satisfied. If such a layer is found, one of a set of *suggested goal achiever actions* is selected and the algorithm moves back down through the matrix until a suitable slot is found for that action. Slot availability is determined by the domain of permitted actions at a given layer. This, in turn, is determined by propagation of

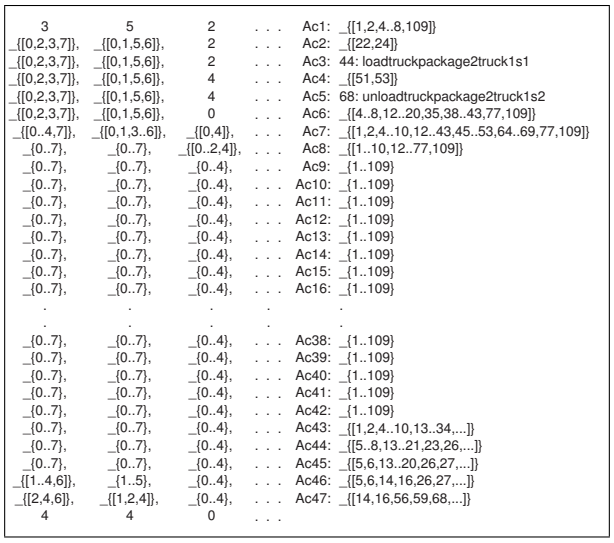
3	5	2	. . .	Ac1: _{[1..8,109]}
_{[0,2,3,7]}	_{[0,1,5,6]}	_{[2,3]}	. . .	Ac2: _{[1..24,109]}
_{[0..4,7]}	_{[0,1,3..6]}	_{[2,3]}	. . .	Ac3: _{[1..53,109]}
_{0..7}	_{0..7}	_{2..4}	. . .	Ac4: _{[1..72,109]}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac5: _{[1..105,109]}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac6: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac7: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac8: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac9: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac10: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac11: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac12: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac13: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac14: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac15: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac16: _{1..109}
.	.	.	.	.
.	.	.	.	.
_{0..7}	_{0..7}	_{0..4}	. . .	Ac38: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac39: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac40: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac41: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac42: _{1..109}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac43: _{[1,2,4..10,13..34,...]}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac44: _{[5..8,13..21,23,26,...]}
_{0..7}	_{0..7}	_{0..4}	. . .	Ac45: _{[5,6,13..20,26,27,...]}
_{[1..4,6]}	_{1..5}	_{0..4}	. . .	Ac46: _{[5,6,14,16,26,27,...]}
_{[2,4,6]}	_{[1,2,4]}	_{0..4}	. . .	Ac47: _{[14,16,56,59,68,...]}
4	4	0	. . .	

Figure 2: Partial solution matrix after constraints applied.

the constraints resulting from the variable assignments made in previous layers. This first part of the solution procedure is inspired by Jussi Rintanen's work (Rintanen 2010) in the area of planning with satisfiability (SAT).

With an action chosen, and available slot found, the action is placed. This action's preconditions then become new subgoals to be achieved. There now exists a skeleton subplan, bounded at one end by the previous goal's subplan (or initial state) and at the other by the current goal's achieving action. It is now only necessary to find supporting actions for that goal achiever; this is a small, self-contained CSP with a very limited horizon. The solver is able to make better use of inference here due to the propagation of constraints from both ends of the subproblem. Proceeding recursively in this way, blocks of actions are formed into subplans. Upon completion of a subplan, control returns to the highest level and the next goal-state goal is processed.

Tracing this procedure through on the matrix shown in Figure 2 helps illustrate the operation of the algorithm. The first ordered goal-state goal, variable number 3 (bottom row, column 3), is chosen. The set of suggested actions that achieve the goal value is found, in this case actions 68 and 96. Moving up through the matrix, the first point where the variable is instantiated and not equal to its current value (0) is in the initial state, where it has the value of 2. Proceeding down the matrix now, it is clear that the domains of the first three action slots do not allow either action 68 or action 96. The first available slot is action slot 4, where action 68 is tried. Propagation of this choice leads to a backtrack, showing that, although there had been some propagation of the initial constraints, there wasn't enough inference to reduce the domain of action slot 4. Continuing down to action slot 5, action 68 is placed and propagation occurs. This leads to the situation shown in Figure 3. With action 68 in place, the solver has inferred that action 44 must be placed in action slot 3 and has propagated the new constraints arising from these assignments.



3	5	2	. . .	Ac1: _{[1,2,4..8,109]}
_{[0,2,3,7]},	_{[0,1,5,6]},	2	. . .	Ac2: _{[22,24]}
_{[0,2,3,7]},	_{[0,1,5,6]},	2	. . .	Ac3: 44: loadtruckpackage2truck1s1
_{[0,2,3,7]},	_{[0,1,5,6]},	4	. . .	Ac4: _{[51,53]}
_{[0,2,3,7]},	_{[0,1,5,6]},	4	. . .	Ac5: 68: unloadtruckpackage2truck1s2
_{[0,2,3,7]},	_{[0,1,5,6]},	0	. . .	Ac6: _{[4..8,12..20,35,38..43,77,109]}
_{[0..4,7]},	_{[0,1,3..6]},	_{[0,4]},	. . .	Ac7: _{[1,2,4..10,12..43,45..53,64..69,77,109]}
_{0..7},	_{0..7},	_{[0..2,4]},	. . .	Ac8: _{[1..10,12..77,109]}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac9: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac10: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac11: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac12: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac13: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac14: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac15: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac16: _{1..109}
.	.	.	.	.
.	.	.	.	.
_{0..7},	_{0..7},	_{0..4},	. . .	Ac38: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac39: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac40: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac41: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac42: _{1..109}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac43: _{[1,2,4..10,13..34,...]}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac44: _{[5..8,13..21,23,26,...]}
_{0..7},	_{0..7},	_{0..4},	. . .	Ac45: _{[5,6,13..20,26,27,...]}
_{[1..4,6]},	_{1..5},	_{0..4},	. . .	Ac46: _{[5,6,14,16,26,27,...]}
_{[2,4,6]},	_{[1,2,4]},	_{0..4},	. . .	Ac47: _{[14,16,56,59,68,...]}
4	4	0	. . .	

Figure 3: Partial solution matrix with action placement.

Continuing the process recursively, the next subgoal, taken from the preconditions of action 68, is chosen. There is now a much smaller search space due to the reduced domains of variables 1 and 2 in the first 5 rows of the matrix. Also, the range of possible values for action slots 1, 2 and 4 is greatly reduced, having been pruned, again by propagation of constraints resulting from the previous action allocations.

It is also clear to see that further pruning has occurred in the domains of the first three variables **after** the assignment of the goal-state goal achieving action, action 68, in slot 5. This, together with the reduction of the domains of action slots 6 to 8, aids the choice of actions in the next top-level iteration which constructs the next subplan in order to satisfy the next of the ordered goal-state goals. The completed subplan for the achievement of the first goal-state goal can be seen in Figure 4.

The efficacy of the goal-directed algorithm is detailed in the results section of this paper. However, there exist problem instances for which this approach is ineffective due to the introduction of conflicts between sub-plans. In such cases the system backtracks over the (goal-centric) suggested supporting action choice and defaults to a standard CSP labelling technique (Dechter 2003) in order to circumvent the conflict and solve the current goal. This backup technique negatively impacts on the system's performance, but this is mitigated by the fact that the search is localised and is aided by the propagation coming from both ends of the sub-plan.

Meta Variables

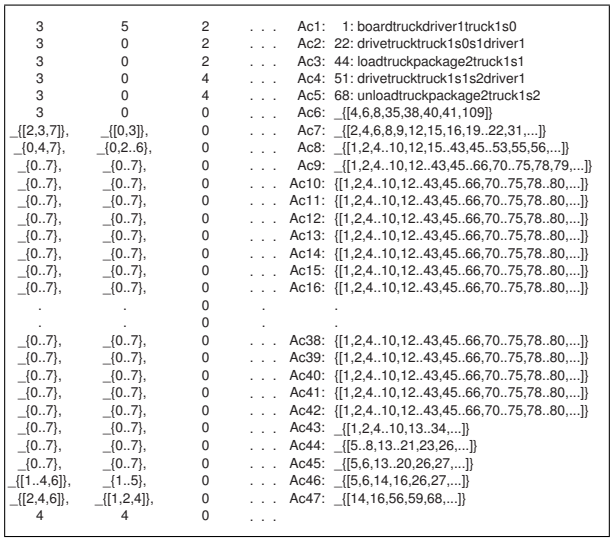
It is possible to eliminate interchangeable values in constraint satisfaction problems (Freuder 1991). In this way, by clustering subsets of the original CSP variables, a set of *meta-variables* can be created.

Definition 3. *A* meta-*CSP of a ground CSP,* P*, consists of variables that correspond to subsets of the variables in* P*. The values of the meta-variables are the solutions of the problems induced by the subsets of variables. The constraints between the meta-variables are satisfied when all of the constraints from the original CSP are satisfied.*

Further, applying this concept to planning (Gregory, Long, and Fox 2007) allows a meta-CSP to be constructed, the variables of which represent the goals of the original planning problem.

Definition 4. *For a planning problem, P = (O,I,G), the meta-CSP is a CSP where* $|X| = |G|$ *and* $D_i = \{\ a \mid g_i \in add(a)\}$.

Each variable represents a goal, $g_i \in G$, with the associated domain containing only the actions that achieve g_i. Such actions have g_i in the add list of their effects. With a variable assigned a value in this representation, $\langle x_i, a \rangle$, the meaning is that a is the *final achiever* of g_i. This means that it is not possible for any action appearing later in the plan than a to delete g_i.



3	5	2	. . .	Ac1: 1: boardtruckdriver1truck1s0
3	0	2	. . .	Ac2: 22: drivetrucktruck1s0s1driver1
3	0	2	. . .	Ac3: 44: loadtruckpackage2truck1s1
3	0	4	. . .	Ac4: 51: drivetrucktruck1s1s2driver1
3	0	4	. . .	Ac5: 68: unloadtruckpackage2truck1s2
3	0	0	. . .	Ac6: _{[4,6,8,35,38,40,41,109]}
_{[2,3,7]},	_{[0,3]},	0	. . .	Ac7: _{[2,4,6,8,9,12,15,16,19..22,31,...]}
_{0,4,7},	_{0,2..6},	0	. . .	Ac8: _{[1,2,4..10,12,15..43,45..53,55,56,...]}
_{0..7},	_{0..7},	0	. . .	Ac9: _{[1,2,4..10,12..43,45..66,70..75,78,79,...]}
_{0..7},	_{0..7},	0	. . .	Ac10: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac11: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac12: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac13: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac14: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac15: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac16: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
.	.	0	.	.
.	.	0	.	.
_{0..7},	_{0..7},	0	. . .	Ac38: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac39: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac40: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac41: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac42: {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
_{0..7},	_{0..7},	0	. . .	Ac43: _{[1,2,4..10,13..34,...]}
_{0..7},	_{0..7},	0	. . .	Ac44: _{[5..8,13..21,23,26,...]}
_{0..7},	_{0..7},	0	. . .	Ac45: _{[5,6,13..20,26,27,...]}
_{[1..4,6]},	_{1..5},	0	. . .	Ac46: _{[5,6,14,16,26,27,...]}
_{[2,4,6]},	_{[1,2,4]},	0	. . .	Ac47: _{[14,16,56,59,68,...]}
4	4	0	. . .	

Figure 4: Partial solution matrix with 1st subplan.

The result of using this additional constraint machinery is that, once achieved, a given goal variable's value will be maintained until the end of the plan. Figure 4 shows an example of this. Variable 3's goal value, 0, is achieved by action 68 in action slot 5. Thereafter, it is *locked* until the end of the plan. This approach facilitates an increase in the amount of inference, as evidenced by the reduction in the domains of action variables 6 through 42 in the example of Figure 4. As the solution develops and further goals are achieved, the cumulative effect of the locking meta-variables further aids solution of the problem.

Taken together with the goal-oriented heuristic approach discussed above, the meta-variable goal locking technique provides additional leverage in the solution of planning problems cast as CSPs.

Results

In order to test the methods discussed in this paper, the constraint encoding, heuristic algorithm and meta-CSP were implemented in *ECLiPSe* (Schimpf and Shen 2010) running on a Kubuntu Hardy Linux system with a Pentium IV (3.4GHz) processor and 1GB of RAM.

The table constraint encoding, although expected to become memory intensive on the larger problem instances, proved acceptable over the test set used in this study.

The prototype system was tested on a number of domains taken from previous International Planning Competitions (IPC)[2].

Figure 5 shows, for a selection of IPC problem instances, the difference in runtime between the standard extensional CSP encoding and that same encoding with added heuristic guidance and meta-CSP variables. In three instances the standard encoding is faster. In those cases the problem is so simple, and plan so short, that the base inference is solving the problem. However, in all other cases, the enhanced (heuristic) system is faster, sometimes significantly so. For example, problem instance 3 in the TPP domain is solved in 404 seconds by the base system, whereas a solution is reached in just 3 seconds with the prototype. Likewise, Elevator instance 3 required 7,991 seconds with no guidance, but only 1.9 seconds with the heuristic.

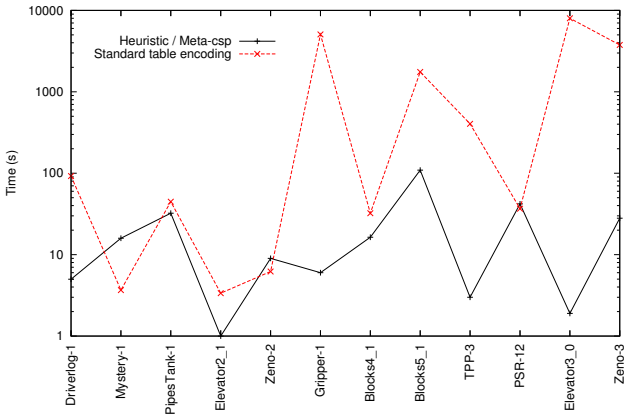


Figure 5: Standard encoding vs. Heuristic.

The standard system is only able to solve problem 1 in the Driverlog domain whereas the prototype solves instances 1 to 13 (Figure 6). This domain provides an intuitive understanding of the algorithm; with the goals ordered, the procedure starts servicing the package goals. The suggested actions are all of the *unload package* type, the choice here being between trucks from which to unload. With one such action selected, there is a requirement for the given truck to be at the pick up point to load the package and, further, for a driver to be at the truck's initial location to board the truck. Although the heuristic technique is clearly working, it is obvious from Figure 6 that the larger instances, those with increased numbers of variables and larger search spaces, could benefit from further heuristic guidance through the remaining resource search space. A suggested approach is provided in the conclusions section of this paper.

[2]http://ipc.icaps-conference.org/

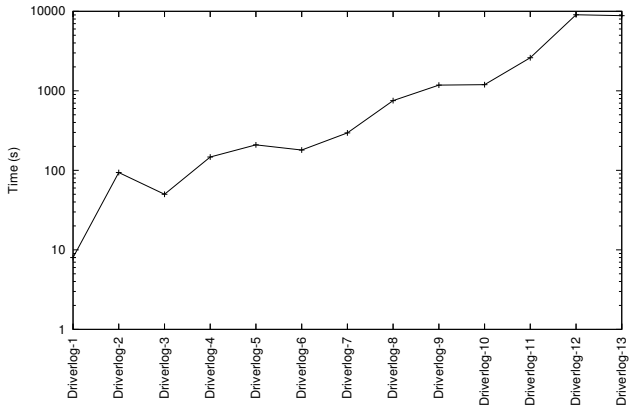


Figure 6: Driverlog instances 1 to 13.

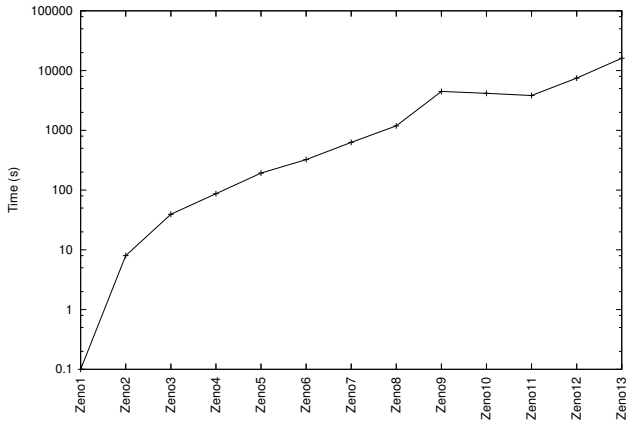


Figure 7: Zeno instances 1 to 13.

Figures 7 and 8 again highlight the efficacy of the techniques described in this paper. There were only 3 instances of the Zeno domain solved by the standard system, but 13 with the prototype. A similar increase was found in the Elevators domain; from 3 to 18.

The final graph, Figure 9, contains results for additional selected instances in the TPP, Gripper, BlocksWorld and Pipes-Tankage domains, shown in ascending runtime order.

It is interesting to note that, in the Gripper problems, despite being limited to carrying one ball at a time due to the goal-centric nature of the algorithm, the prototype system can return faster solutions. Compared to the non-heuristic approach, in which the search space comprises all combinations of ball and gripper, it is clear to see that by selecting just one goal (ball) at a time the prototype system removes

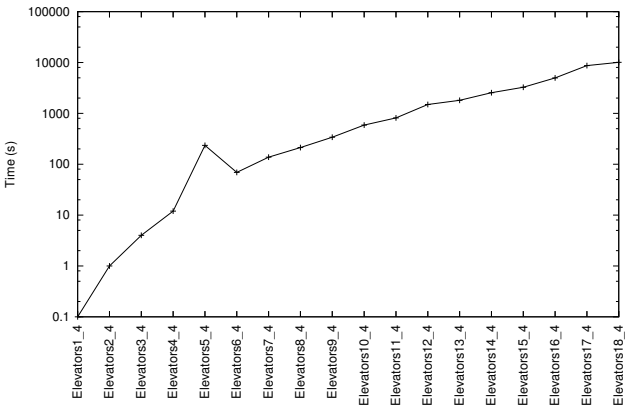


Figure 8: Elevator instances 1 to 18.

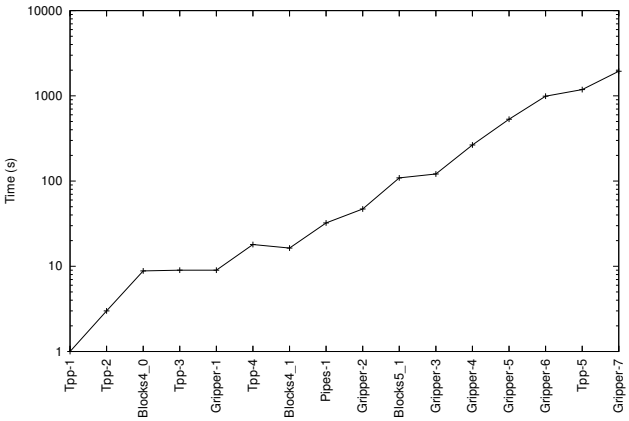


Figure 9: Selected instances solved within time limit.

most of the symmetry from the problem, thereby significantly pruning the search space.

In summary, the results show the significance of the heuristic / meta CSP approach: It is indeed possible to better use the structure of the original planning problem to guide the search for a solution to the CSP reformulation of that problem. This is evidenced by the fact that, where the base system could solve a given problem, the prototype could solve it significantly faster. Also, in the test cases where the base system could not find a solution within the time limit, the prototype provided solutions.

Conclusions

This paper has described two methods for gaining extra leverage in the solution of CSP encoded planning problems. The intention was to show that better use could be made of inference, meaning that a solution could be reached with less reliance on blind search.

The first approach, a variable and value ordering heuristic, was shown to increase the amount of propagation by introducing intermediate horizons. These act as bounds from which the effects of variable / value choices are propagated in both forward and backward directions. This tightening of constraints reduces the search space and forces out useless action choices.

The introduction of meta-variables to act as locks on the final achievers of goal-state goals allowed further inferences to be made. This additional pruning of the search space has a cumulative effect.

Used in tandem, these techniques have shown that it is indeed possible to make better use of the inherent constraint propagation machinery. However, the authors believe that further gains can be made.

One direction for future work is the use of a heuristic estimator for the allocation of resources during the solution of the problem. It is proposed that by measuring which is the most cost effective resource to allocate, it should be possible to reduce the number of steps in a given plan, thereby producing a more efficient solution.

References

Backstrom, C. 1992. Equivalence and Tractability Results for SAS+ Planning. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, 126–137. Morgan Kaufmann.

Bartak, R., and Toropila, D. 2008a. Enhancing Constraint Models for Planning Problems. In *Proceedings of the Twenty Seventh Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2008)*.

Bartak, R., and Toropila, D. 2008b. Reformulating Constraint Models for Classical Planning. In *Proceedings of the Twenty First International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 525–530. AAAI Press.

Bartak, R. 2011. A Novel Constraint Model for Parallel Planning. In Murray, R. C., and McCarthy, P. M., eds., *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference, May 18-20, 2011, Palm Beach, Florida, USA*. AAAI Press.

Beacham, A.; Chen, X.; Sillito, J.; and van Beek, P. 2001. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the Fourteenth Canadian Conference on Artificial Intelligence*, 78–87.

Cesta, A., and Fratini, S. 2008. The timeline representation framework as a planning and scheduling software development environment. In *Proceedings of the Twenty Seventh Workshop of the UK Planning and Scheduling Special Interest Group, Edinburgh, UK, December 11-12, 2008*.

Clark, D. A.; Frank, J.; Gent, I. P.; MacIntyre, E.; Tomov, N.; and Walsh, T. 1996. Local search and the number of solutions. In Freuder (1996), 119–133.

Dechter, R., and Meiri, I. 1994. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence* 68(2):211–241.

Dechter, R. 2003. *Constraint processing*. Elsevier Morgan Kaufmann.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132(2):151–182.

Fikes, R. E., and Nilsson, N. J. 1990. Strips: A new approach to the application of theorem proving to problem solving. In Allen, J.; Hendler, J.; and Tate, A., eds., *Readings in Planning*. San Mateo, CA: Kaufmann. 88–97.

Freuder, E. C. 1978. Synthesizing constraint expressions. *Communications of the ACM* 21(11):958–966.

Freuder, E. C. 1991. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 1*, AAAI'91, 227–233. AAAI Press.

Freuder, E. C., ed. 1996. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*. Springer.

Freuder, E. C. 1999. Modeling: The final frontier. In *The First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP 99)*.

Gent, I. P.; MacIntyre, E.; Prosser, P.; Smith, B. M.; and Walsh, T. 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In Freuder (1996), 179–193.

Ghallab, M.; Knoblock, C.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D. E.; Sun, Y.; and Weld, D. 1998. *PDDL: The Planning Domain Definition Language*. Yale University.

Goldman, R. P.; Haigh, K. Z.; Musliner, D. J.; and Pelican, M. 2000. MACBeth: A Multi-Agent Constraint-Based Planner. In *Working Notes of the AAAI Workshop on Constraints and AI Planning*.

Gregory, P.; Fox, M.; and Long, D. 2010. Constraint Based Planning with Composable Substate Graphs. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence*.

Gregory, P.; Long, D.; and Fox, M. 2007. A Meta-CSP Model for Optimal Planning. In *Proceedings of Abstraction, Reformulation and Approximation, Seventh International Symposium*, 200–214.

Harvey, W.; Kelsey, T.; and Petrie, K. E. 2003. Symmetry Group Expression for CSPs. In Smith, B. M., ed., *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03) - CP 2003*, 86–96.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS*, 161–170.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191 – 246.

Joslin, D., and Pollack, M. E. 1995. Passive and Active Decision Postponement in Plan Generation. In *Proceedings of the Third European Conference on Planning*, 1–15.

Kautz, H. A., and Selman, B. 1992. Planning as Satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 359–363.

Lopez, A., and Bacchus, F. 2003. Generalizing GraphPlan by Formulating Planning as a CSP. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI 03)*, 954–960.

Mackworth, A. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8:77–98.

Porteous, J., and Sebastia, L. 2004. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research* 22:215–278.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9:268–299.

Rintanen, J. 2010. Heuristics for planning with SAT. In *Principles and Practice of Constraint Programming - CP 2010 - Sixteenth International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*. Springer.

Russell, S. J.; Norvig, P.; Candy, J. F.; Malik, J. M.; and Edwards, D. D. 2003. *Artificial Intelligence - A Modern Approach*. Prentice-Hall, Inc., 3rd edition.

Schimpf, J., and Shen, K. 2010. Eclipse - from lp to clp. *CoRR* abs/1012.4240.

Smith, B. M. 2005. Modelling for Constraint Programming. First International Summer School on Constraint Programming.

Stefik, M. 1981. Planning with Constraints. *Artificial Intelligence (An International Journal)* 16:111–140.

Tsang, E. P. K. 1993. *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press.

van Beek, P., and Chen, X. 1999. CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 99)*, volume 100, 585–590. AAAI Press.

Verfaillie, G., and Pralet, C. 2008. How to Model Planning and Scheduling Problems using Timelines. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS'08)*.

Vidal, V., and Geffner, H. 2006. Branching and pruning: an optimal temporal pocl planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

Planning Modulo Theories: Extending the Planning Paradigm

Derek Long and **Maria Fox**
Department of Informatics
King's College London, UK
firstname.lastname@kcl.ac.uk

Peter Gregory
School of Computing and Engineering
University of Huddersfield, UK
p.gregory@hud.ac.uk

Chris Beck
Mechanical and Industrial Engineering
University of Toronto, Canada
jcb@mie.utoronto.ca

Abstract

The classical planning problem is defined in a purely propositional language. Considerable effort has been devoted, particularly in the past decade, to extending the scope of planning to include richer domains, including time and numbers. Each of these extensions has been designed as a separate and specific semantic enrichment of the underlying planning model, with its own representational syntax.

A part of the SAT community has recently started exploring an extension of SAT to include first order theories, developing solvers for SAT Modulo Theories (SMT), in which the theories are essentially parameters of the solvers. In this paper, we argue that it is fruitful to apply an analogous idea to planning and define Planning Modulo Theories (PMT).

We indicate how several existing extensions to classical planning can be seen as instances of PMT. We provide a modular extension to PDDL with PMT as its theoretical underpinning, and discuss the differences between modelling in PDDL and our framework. We also demonstrate that a heuristic planner based on PMT can provide competitive performance to current state-of-the-art planners as well as extending the range of interesting expressive power in modelling planning domains.

1 Introduction

Classical planning is the problem of finding a path through a state space. States are valuations over propositional variables and the transitions between states are actions. Each action is described by a precondition formula, which must be satisfied in a state in order for the action to be applied, and a collection of effects which assign new values to a subset of the state variables. The remaining variables are assumed to remain unchanged (the STRIPS assumption). Preconditions are arbitrary logical propositional formulae over the propositional variables. This basic problem is usually lifted so that variables are indexed by a finite set of parameters and action schemas parameterised with values from this set.

In its simplicity, classical planning shares a great deal with SAT, which is also concerned with valuations of propositional variables. An interesting development in the SAT community is recent interest in the extension of the propositional language of SAT via SAT Modulo Theories (SMT) (Nieuwenhuis, Oliveras, & Tinelli 2006). SMT is the following family of problems:

Instance: Given a formula, F, of first order logic (including constant, predicate and function symbols), and a theory, T, defining the meanings of the constant, predicate and function symbols.

Question: Is F satisfiable, subject to the interpretations of the symbols imposed by T?

A language, SMTlib (Barrett, Stump, & Tinelli 2010), has been developed for describing these problems and more than 18 solvers have been constructed that can solve problems for specific instantiations of T, including Difference Logic, Linear Arithmetic, Arrays, Lists and Non-Linear Real-valued Functions (de Moura & Bjørner 2008; Nieuwenhuis, Oliveras, & Tinelli 2006; Dutertre & de Moura 2006).

In this work we present a new planning formalism called Planning Modulo Theories (PMT), which allows classical planning to be arbitrarily extended in a modular way, following a similar path to SMT. New types can be added (for example: sets, vectors, etc.) and functions that operate over those types can be incorporated into actions. Our vision is that PMT offers a way to unify a wide variety of work on planning that uses different (application-oriented) extensions of the propositional core. We briefly consider how existing planners can be seen as solvers for PMT with specific theories. We present an implementation of a planner that operates on this class of problems, and we demonstrate the advantage of this formalism over modelling in PDDL. We go even further and demonstrate that there are problem classes that cannot be adequately represented or solved in PDDL, for which PMT-based planners can both naturally represent and scale better than in PDDL.

In the remainder of this paper we introduce the PMT problem formally in Section 2 and discuss in Section 3 the modular extension to PDDL that we use to represent PMT problems. We then compare a PDDL representation and a PMT representation of the same problem in Section 4. In Section 5 we show how a forward-chaining heuristic search planner can be created to solve PMT problems. Finally, in Section 6 we provide some evidence that solving problems in this way can be more effective than translating into PDDL, even with an optimised model.

2 Planning Modulo Theories

We now introduce the formal definition of Planning Modulo Theories, which is analogous to the classical planning problem extended in the same way that SMT extends SAT. Recall that a (first order) theory is a set of first order sentences, usu-

ally constructed as the deductive closure of a set of axioms, that defines the behaviour of a (possibly infinite) collection of symbols: constants, functions and predicates. Examples are the theories of arithmetic and of set theory.

Definition 2.1 — State A *state* is a valuation over a finite set of variables, V, where each variable, $v \in V$, has a corresponding domain of possible values, D_v. The *state space* for V is the set of all valuations over V.

Classically, all state variables have boolean domains. In SAS+ encodings, variables have finite domains, while, in metric planning, domains can be integers or real numbers. In PMT we allow arbitrary domains, such as sets, lists, multi-dimensional vectors over reals and so on.

Definition 2.2 — First order sentence over a state space S modulo T A *first order sentence over a state space S modulo T* is a first order sentence over the variables of the state space, constant symbols, function symbols and predicate symbols, where T is a theory defining the domains of the state space variables and interpretations for the constants, functions and predicates. A *term* over S modulo T is, similarly, an expression constructed using the symbols defined by S and T.

The key to extending classical planning into PMT is to support first order sentences modulo theories in the preconditions of actions.

Definition 2.3 — Action An *action*, A, for a state space S modulo T, is a state transition function, comprising:

- A first order sentence over S modulo T, Pre_A, called the *precondition* of A.
- A list, Eff_A, of assignments to a subset of the state variables, each setting a variable to a value defined by a term over S modulo T.

One of the most important early developments in early planning was the introduction of the STRIPS assumption: that variables not explicitly affected by an action are assumed unchanged. This dramatically simplifies the description of actions compared with complete axiomatisations. We adopt the same assumption in our definition of action application:

Definition 2.4 — Action application An action, A, for state space S modulo T, is applicable in state $s \in S$ if $T \cup \{s\} \models Pre_A$. The state resulting from application of A is the result of applying the assignments in Eff_A to the subset of variables in s that they affect and leaving all other variables unchanged.

We now complete the definition of the PMT problem.

Definition 2.5 — Planning Modulo Theory A *Planning Modulo T* problem is the tuple $\langle S, A, I, G\rangle$ where:

- S is a state space in which all of the variable domains are defined in T;
- A is a collection of actions for S modulo T;
- I is an initial valuation (the *initial state*) and
- G is a first order sentence over S modulo T (the *goal*).

A *plan* is a sequence of actions, $\langle a_1, ..., a_n\rangle$ such that a_i is applicable in state s_{i-1} and s_i is the result of applying a_i to s_{i-1}, where $s_0 = I$ and $T \cup \{s_n\} \models G$.

PMT can also be lifted to allow variables and actions to be parameterised. Variables must be parameterised with finite types to ensure that the set of variables remains finite. It is also straightforward to generalise the above definitions to include parallel plans and temporal planning problems and their plans (Fox & Long 2003; 2006).

The idea of extending classical planning to include interpreted predicates and functions is not entirely new, although it has not been generally discussed in those terms. A significant exception is the proposal by Geffner (2000) for Functional STRIPS, which is very similar in intention to our proposal. However, while Functional STRIPS is powerful enough to express computations on complex data structures and relationships between these data structures and the propositional part of a planning problem, the idea has not been pursued. Furthermore, the Function STRIPS proposal does not make theories an explicit parameter of the encodings. In this paper we propose a different approach which shares some of the same ideas, but goes further in terms of the way theories are integrated as parameters to planning problems and we go on to show how these ideas can be supported in implementation.

In table 1 we illustrate some of the theories that have been used in existing planning systems. The table shows several planners that have been designed to solve planning problems modulo specific theories. In each case, the interpretation of the theory is managed via a specialised subsolver that can evaluate terms and predicates over terms within the theory, supporting the associated planner in solving problems that use the theory. This support varies from a simple consistency check for sentences in the theory (eg the STN in CRIKEY3) through to production of no-goods for the associated planner (eg LPSAT).

Other authors have also observed the opportunities that SMT offers for planning with specific theories. In particular, TM-LPSAT (Shin & Davis 2005) uses a compilation approach to convert PDDL+ problems into SAT modulo LP problems, while Wolfman and Weld's LP-SAT (1999) uses a similar compilation to achieve solutions to planning with metric resources. TLP-GP (Maris & Régnier 2008) also compiles to SMT to handle temporal problems, using SAT modulo temporal problems (solved with STNs). In all of these examples the researchers have considered the planning problems as classical planning supplemented with specific fixed theories and have perceived the SMT compilation as a way to access the power of SMT to solve SAT modulo the specific theories involved. This is quite distinct from the idea of theories as *parameters* of the planning problem.

Research on SMT has focussed on theories that have application in the kinds of program and hardware verification tasks that are common application targets of the SAT technology, such as theories of arithmetic and of arrays. Similarly, planning can benefit from theories of spatial and kinematic reasoning to support integrated task and motion planning (Cambon, Alami, & Gravot 2009), data-operations for planning pipeline operations such as image processing (Golden 1998) or web-services (Hoffmann 2008), or of data-structure variables (arrays, sets, lists) to support, for ex-

Planning problems	Theory	Planner	Subsolver
Simple durative actions (PDDL2.1)	Difference Logic	Sapa (Do & Kambhampati 2003) CRIKEY3 (Coles *et al.* 2008b)	STN solver
Continuous effects (PDDL+)	Linear Programs	COLIN (Coles *et al.* 2009)	LP solver
Metric resources	Presburger Arithmetic	Metric-FF (Hoffmann 2003) LP-RPG (Coles *et al.* 2008a) Filuta (Dvorak & Barták 2010) LPSAT (Wolfman & Weld 1999)	Interval bounded search LP solver Specialised solver SAT modulo LP
Axioms	Datalog	FF variant (Thiébaux, Hoffmann, & Nebel 2005)	Specialised
3d manufacturing problems	3d geometry	IMACS (Gupta, Nau, & Regli 1998)	CAD-based
Physical system models	Non-linear functions	ASPEN (Chien *et al.* 2000) SHOP (Nau *et al.* 1999) Europa (Reddy *et al.* 2008)	External code attachments
Data products	Ontological theories	WSC planner (Hoffmann *et al.* 2008)	Inference over ontologies

Table 1: Some of the existing examples of planners that use sub-solvers to work with theories.

ample, bioinformatics applications such as DNA analysis. There are also many examples of systems that integrate programmed functions into planning models to support efficient modelling of complex subsystems, such as batteries, thermal behaviour and so on (Chien *et al.* 2000). PMT subsumes all of these examples and incorporates them into a single uniformly defined problem structure.

3 Describing PMT Problems

To support convenient description of PMT domains, we propose a modular extension of PDDL. The idea of a modular language mirrors SMTlib (Barrett, Stump, & Tinelli 2010), allowing easy extension by the addition of new theories providing new types, with new interpreted constants and predicates and functions over these types. Although we call the new language PDDL, we denote module files MDDL and core files CDDL files, in order to provide discrimination. In this section, we describe the CDDL and MDDL formats and then discuss some finer details of the language.

3.1 MDDL and CDDL

In CDDL and MDDL all constants and functions are typed. There are three types defined in the core language: `boolean`, `pddlobject` and `unit`. The type `pddlobject` defines object fluents in the language. The `unit` type defines a class of special functions over a type, used to update variables during action application. Unit functions return values of the same type as their first argument, which is called the left-hand-side (LHS) of the function. Such functions can be used to update variables of the type of their first argument: we interpret effects written as unit functions to mean that the value of the LHS of the function is updated to take the return value of the function in the next state.

The core language provides two functions: a polymorphic structural equality and a polymorphic assignment function. Trivially, the return type of equality is boolean. Assign is a unit function, `(assign ?A x)` updates the value of variable `?A` to `x` in the next state (and can be seen as returning `x`). In theory, the assignment function is the only unit function needed in order to model effects. However, it is often more concise and readable for the modeller to define additional unit functions. For example `(double ?X)` is much more convenient to write than `(assign ?X (* ?X 2))`.

```
(define (module set)
  (:type set of a')
  (:functions
  (construct-set ?x+ - a')                    - set of a'
  (empty-set)                                 - set of a'
  (cardinality ?s - set of a')                - integer
  (member      ?s - set of a' ?x - a')        - boolean
  (subset      ?x - set of a' ?y - set of a') - boolean
  (union       ?x - set of a' ?y - set of a') - set of a'
  (intersect   ?x - set of a' ?y - set of a') - set of a'
  (difference  ?x - set of a' ?y - set of a') - set of a'
  (add-element ?s - set of a' ?x - a')        - set of a'
  (rem-element ?s - set of a' ?x - a')        - set of a'))
```

Figure 1: The definition of a set module in MDDL.

Each module can define a new type and functions over that type. Modules are supported by implementations of function evaluators that are supplied through interfaces to the planner. With this modular structure it is relatively straightforward to add a collection of new functions into the language. Once added, these functions are then available to use in any future domains. As examples, we have encoded integer, set and multiset modules, as well as a more speculative path-planning module for robotic control. The set module is shown in Figure 1. Functions in these modules can return any types; for example, the cardinality function in the set module returns an integer.

The set module defines a polymorphic set type. Its functions can be used in operators, the initial state and goals. Sets can be constructed in the initial state either by the empty-set function or the construct-set function. The parameters of the construct-set function are `?x+ - a'`, meaning that the function takes one or more constants of type `a'`. Since it will often be useful to have variable numbers of arguments in function calls, we have allowed the C-like syntax permitting a variable number of constants of the final parameter in the function prototype. Strictly, this means that such definitions define a related family of functions, with overloaded names and differentiated by their arities.

A CDDL planning operator is defined in two parts: a precondition list and an effect list. Each precondition is a boolean function over the state variables and the list is interpreted as a conjunction. Each effect is a unit function over the state variables in which the LHS is a `pddlobject`. An example CDDL operator is shown in Figure 2. Notice that this action involves the interaction of integers, object fluents

```
(:action load-truck
  :parameters (?p - package ?t - truck)
  :precondition
   ((member (at (loc-of ?t))       ?p)
    (<      (cardinality (in ?t)) 2))
  :effect
   ((rem-element (at (loc-of ?t)) ?p)
    (add-element (in ?t)          ?p)))
```

Figure 2: A sample operator that uses a combination of the set module, the integer module and object fluents.

```
(define (domain setdomain)
  (:types
    location locatable - object
    truck obj          - locatable)
  (:modules integer set)
  (:functions
    (at ?loc - location)        - set of package
    (loc-of-truck ?tru - truck) - location
    (in ?tru - truck)           - set of package
    (linked-to ?x - location)   - set of package)
```

Figure 3: Header of a CDDL domain file.

and sets. In comparison to a similar PDDL operator, this operator has fewer parameters, as the location of the truck is determined functionally by an object fluent expression. Also, in order to enforce a capacity constraint on each truck, it is only necessary to use the cardinality function. In PDDL it would have been necessary to maintain a numeric fluent to count the items in a truck. From an engineering perspective, the PMT version is more robust, as there are fewer opportunities to introduce errors in the model.

Variables can be assigned initial values or left undefined. The *undefined* value is considered to be a member of every type except `boolean`, is excluded from use in preconditions, but is free to be assigned as an effect. The reason that `boolean` has no undefined value is that we use a closed-world assumption (as is usual in planning) and treat anything that is not true as false.

3.2 Types of Action Parameters

In current variants of PDDL, parameters are restricted to finite types, primarily to simplify grounding of actions. It is possible to relax this constraint while maintaining a straightforward semantics: provided the types used to define the variables in a state are finite, then actions may have parameters drawn from infinite types. The `?duration` parameter of durative actions already represents a special case of such a parameter. The key to managing such parameters is that plans are always finite. Each action in a plan may have its own (finite) set of control parameters, so that the trajectory of a plan will require finitely many action control parameters to be instantiated, along with finitely many state variables. We propose that PDDL constraints be relaxed to allow parameters of any types, but variables defined in the `:functions` field of the domain definition continue to be restricted to parameters from finite types (see Figure 3).

3.3 Standard Modules and Beyond

In order for a planner to make use of extensions, it is, of course, necessary that it support the appropriate functions, predicates and types of the theories. As has been the case in SMT, we anticipate that a library of standard extensions will be constructed so that planners can depend on the same symbols being used for standard predicates and functions. However, our planner is equipped with a standard interface allowing external sub-solvers to provide the semantic attachments to support new functions or predicates and through these the planner can be made capable of handling newly specified theories as they are introduced.

4 PMT Versus PDDL

In this section, we consider a small example which it is natural to model using sets, but which is more problematic to implement in PDDL, even with quantification and numerics. Consider a logistics-style domain where trucks are required to deliver packages between different locations. Unlike typical benchmark domains, packages are loaded into trucks one at a time, whilst all packages in a truck must be unloaded at the same time.

In the PMT framework, this unload action (along with relevant state functions) can be modelled as follows:

```
(:functions
  (loc-of-truck ?tru - truck) - location
  (at ?loc - location)        - set of package
  (in ?tru - truck)           - set of package)

(:action UNLOAD-TRUCK
  :parameters  (?truck - truck)
  :precondition  (true)
  :effect
   ((union (at (loc-of-truck ?truck)) (in ?truck))
    (assign (in ?truck) (empty-set))))
```

The function `loc-of-truck` defines an object fluent denoting the location of the truck. The `at` and `in` functions return sets representing the packages which are at locations and in trucks respectively. The unload action uses this representation to model the effects correctly. The effects can be understood straightforwardly: the set of packages at the location of the truck is assigned to its union with the packages in the truck and there are then no packages left in the truck. Note the use of the unit function to update the set of packages at the location of the unloaded truck: the LHS evaluates to a variable (the packages at a location) and the interpretation of the effect is that this variable is updated to take the value computed by the function.

In order to model this problem in standard PDDL, we propose the following alternatives:

1. The ADL fragment of the language could be used, quantifying over the packages in the effects of actions;
2. The propositional fragment of PDDL could be used and the powerset of the universal package set could then be enumerated as distinct named PDDL objects.

The ADL approach seems preferable to the propositional one, since the latter requires an exponential construction of sets of packages. Many planners ground the powerset when confronted with the ADL version, effectively producing the same model as the propositional version.

Within the PMT framework, it is possible to specify goals based on functional evaluation. For example, the goal:

```
(= (cardinality (at location2))
   (cardinality (at location1)))
```

specifies that the number of packages at `location1` must equal the number of packages at `location2`. It is possible to encode this type of goal in PDDL only by changing the original model, adding numeric variables into the original domain to count the numbers of items in the trucks and at locations. This would require the counts to be updated in a consistent way in load and unload actions (note that this

is only possible because we know implicitly that all package sets are disjoint). Then a goal could be specified in the following way:

```
(= (cardinality-at location2)
     (cardinality-at location1))
```

where `cardinality-at` is a numeric fluent. Using ADL and numerics in combination severely limits the planners which can solve a problem.

It is possible to model this type of goal using the ADL subset of PDDL in combination with the idea of explicitly naming the subsets of packages that can occur. This is achieved by adding a predicate that models whether two (named) sets have the same cardinality. When a load or unload occurs, the effect will determine which (named) subset is now present in the truck and the location. The cardinality of the sets at two locations can be then be checked using the cardinality testing predicate. Adding just one form of goal expression greatly complicates the PDDL model.

In the PMT framework it is possible to specify a very wide range of goals. For example, with the set module available, it is possible to reason about the combination of any sets, using any of the available functions (or new user-defined ones). Every new set reasoned about in the PDDL model must be added explicitly. Consider the following PMT goal:

```
(= (cardinality (union (at location1)
                       (at location2))) 4)
```

which specifies that the union of the two sets `(at location1)` and `(at location2)` must have four elements. In this specific domain, since all sets are disjoint, the sum of the two cardinalities will be equal to the cardinality of the union. However, this is not true in general. To encode this goal in PDDL for the more general case, it is necessary to create a new predicate to maintain the value of the union of the sets of packages at `location1` and `location2`. Actions which change either `at-location1` or `at-location2` need to also change this new `(union-of-at-location1-and-location2)` predicate. In general, the result of combining each pair of subsets of all packages must be encoded propositionally in anticipation that it might be needed in the goal.

In general, it is not possible to create a definitive PDDL representation of a PMT model because there is always a more expressive goal specification than can be captured in the PDDL domain model. From an engineering perspective this is a significant barrier to use, since the model has to be redeveloped as each new goal expression is encountered.

5 Implementation

One possible approach to PMT is to compile domain models into SMT. This approach uses the propositional-planning-as-SAT paradigm and then exploits the similarity of the the roles of theories in PMT and SMT. We have tried this approach, but it suffers from the same weaknesses as propositional-planning-as-SAT: without search guidance that reflects the underlying structure of the planning problem, it scales very poorly.

Our second approach is to construct a PMT planner that operates with PMT models directly. We have implemented a forward state-space planner based on the PMT framework. There are several complications involved when writing a planner to handle this new language. Since types and functions are user-defined through modules, the planner must handle arbitrary (and possibly infinite) types. This has very significant implications for the heuristic computation, as well as complicating other aspects of the typical planning process, such as grounding. At an implementation level, it is a design goal to make it as easy as possible for a user to implement a new module for use in the planner. We first present the implementation of our core solver and indicate what a module developer must implement in order to integrate a new type or set of functions over existing types. We then show how PMT state progression can be implemented before considering search heuristics.

5.1 State Progression

In order to implement a breadth-first search, we simply need to be able to perform state progression, i.e. precondition checking and effect execution.

The planner is split in to two components: the Core and the Modules. The Core is the part of the planner that drives the search and accesses each of the Modules. The Modules are custom components which provide new types, new functions or both. The Core module provides the following types: `boolean`, `pddlobject` (first-class object fluents) and `unit`. It also provides the boolean constants and the null constant. It provides the following minimal interface of functions:

```
assign a' -> a' :: unit
=      a' -> a' :: boolean
```

Each of the Core and Module components satisfies the following interface:

```
evaluate  :: function -> state -> constant
satisfies :: function -> state -> boolean
```

The `evaluate` function returns the value of a given function with respect to a given state. The `satisfies` function determines whether or not a boolean function is satisfied in a particular state (it is the realisation of the first of the components of Definition 2.4). It might appear that `satisfies` is redundant since it can be implemented in terms of `evaluate`. However, we will demonstrate its role when we discuss heuristics. Each module will implement its own `evaluate` and `satisfies` for the functions which it provides. For the set module, `evaluate` will implement `union` function by calculating the actual union of two sets provided as argument, using its own internal representation of the sets involved. The Core `evaluate` function is implemented as shown in Algorithm 1. Notice that the state is used to evaluate those expressions that depend on the current valuation recorded in the state, including the recursive call to specialised evaluation functions for other modules. In the case where the expression is a unit function, the first argument is evaluated to a reference for use as the LHS to be updated. The corresponding `satisfies` function is shown in Algorithm 2, which shows the close relationship between the functions.

The task of the module designer is to implement the functions that each module provides. A precondition expression can then be checked by a simple call to `Core.satisfies`, and an effect can be applied by calling `Core.evaluate` on it and then setting the changed variable(the first argument of the effect) in the next state to take the corresponding value.

Algorithm 1: Core Module: evaluate(function,state)

```
if function is a Constant then
    return function;
else
    let name(parameters) match function;
    i = 0;
    for e ∈ parameters do
        parameter[i] = evaluate(e,state);
        increment i;
    if name returns a pddlobject then
        return value of state[function];
    if name is "assign" then
        return parameter[1];
    if name is "=" then
        return (parameter[0]==parameter[1]);
    if name is a function in Module M then
        return M.evaluate(function,state);
    if name is a boolean function then
        return state[function];
    return UNDEFINED_VALUE;
```

Algorithm 2: Core Module: satisfies(function,state)

```
return evaluate(function,state);
```

This is all that is needed to implement state progression, and with this a forward-state-space breadth-first search is now easily implemented.

5.2 Projections and Interpretations

A breadth-first search is not very practical for interesting planning problems. One of the most powerful techniques in modern planning is the use of relaxed problems as a basis for heuristic guidance, both as a heuristic distance measure for A* or similar searches and for immediate guidance in action selection through helpful actions. One way to view relaxed problems is as searches for plans in projections of the true state space, so that variables are associated with sets of possible values across multiple reachable states. A projection is a transformation of a state into a new state, along with a corresponding set of functions which reinterpret the original functions with respect to this new state. A relaxed state, for example, can be seen as a projection where each value in a state is mapped to a list of values in the relaxed state. The functions then need to be reinterpreted accordingly.

The relaxation projection can be defined recursively. We define a relaxed value as a set of regular values. A regular value can be projected on to a relaxed value, simply by creating a singleton set with that value in it. Then, a new interpretation is given to each function, corresponding to what we mean by relaxed evaluation. For example, instead of the assignment operation returning a single value, it returns a set of values corresponding to the *potential* next values. As an example, the relaxed numeric function `(* {1,2} {3})` evaluates to `{3,6}`.

We say that a projection is *foldable* if two values of its value type can be combined in to a single value. For example, the relaxed projection is foldable because two relaxed values can be combined by simply taking their union. This notion of folding is important when creating the analogue data structure to the relaxed planning graph.

Projections in our solver must conform to this interface:

```
evaluate :: function -> state -> constant
satisfied:: function -> state -> boolean
project  :: constant -> constant
fold     :: constant -> constant -> constant
```

Instead of evaluating functions within the modules, we evaluate them within projections. In order to do this, modules are parameterised by interpretations of functions. Examples of projections, other than the standard relaxation into sets of reachable values, are finite projections, where the values of infinite types are ignored (this is useful to demonstrate unsatisfiability, amongst other things) and bounds projections, where only the bounds of each variable is considered, rather than the entire domain. We also define the identity projection as the base value type, and the original interpretation of the functions. It is worth observing that the bounds projection corresponds to the way that MetricFF handles the numeric type (Hoffmann 2003).

It is because of the way that projections are handled that we separate `satisfies` and `evaluate`, since their implementations can differ in handling particular projections.

5.3 Heuristic Search Algorithms

In order to search for plans more efficiently we need to implement heuristics. In this section, we discuss what is necessary in order to implement a forward heuristic search in PMT similar to the many algorithms in classical planning which use this approach. We define a forward heuristic state-space search algorithm as parameterised by two projections: one as the base projection and another as the relaxation projection (used to compute the heuristic). Typically, the base projection, which defines the space in which a solution is sought, is the identity projection, although we will discuss cases in which it is not.

In order to compute a heuristic value, a relaxed planning graph is constructed in the following way. Given a search algorithm parameterised by base projection b and relaxation projection r, the algorithm to construct an RPG is shown in Algorithm 3. This definition of the RPG is useful because it allows *any* foldable projection to be used as the relaxation. Thus, it is possible to use the full set of reachable values as a relaxed set or an interval relaxation as MetricFF does for numbers, or analogous forms for other types. Three useful relaxations are RPG(Identity, Relaxation), RPG(Finite, Relaxation) and RPG(Identity, Bounds). These are interesting for the following reasons:

RPG(Identity,Relaxation) is the analogue of the classical 'ignore delete-list' relaxation.

RPG(Identity,Bounds) is the analogue of the MetricFF relaxation over numbers. However, for types such as sets and multisets, the appropriate choice of bounds and how to interpret them is less obvious.

Algorithm 3: constructRPG(state,projection) builds RPG

```
r-state = project state;
layer = 0;
rpg[layer] = r-state;
while (goals unsat) do
    action-set = actions, a, such that
    satisfies(pre_a,rpg[layer]);
    state-set = states reachable using a ∈ action-set;
    r-state = fold(r-state,state-set);
    increment layer;
    rpg[layer] = r-state;
```

RPG(Finite,Relaxation) is an example in which the base projection is not the Identity. This is useful because RPG construction might never level off in the presence of infinite types. By ignoring the infinite aspects of the search-space, the RPG might level-off without reaching the goals, identifying a dead-end state.

Armed with a method to construct RPGs, we have the necessary infrastructure to implement a heuristic search algorithm. We have implemented several variations, using different relaxations and various search strategies.

6 Experiments

We now provide some analysis of the performance of our planning system. For this analysis we use a new domain, called the travelling teachers, which is designed to illustrate the potential of set-based domain models. Our evaluation is at an early stage, but we have considered several other domains including the jugs-and-water problem (which uses numbers in an interesting way), the "dumper-truck" logistics domain outlined above, the Settlers domain and the Airport domain (all of which can benefit from a set-based representation). We are restricting our attention to the first two of due to space constraints.

6.1 The skill-transfer domain

In order to compare modelling in PMT and modelling in PDDL, we introduce the 'travelling teachers' domain. In this domain there is a set of teachers, a set of students and a set of skills. The teachers each know a different (possibly intersecting) set of skills. The students begin with no skills at all. Teaching a student results in the student learning all of the skills that the teacher has, though the student does not acquire the ability to teach the skills. A teacher might teach skills which a student has already learnt, in which case the student can learn nothing from the teacher. A PMT encoding of this domain is as follows:

```
(define (domain skill-transfer)
  (:types   teacher student skill)
  (:modules   integer set)
  (:functions  (skills ?t - teacher) - set of skills
               (learnt ?s - student) - set of skills
               (skill-set)           - set of skills)

  (:action teach
   :parameters (?t - teacher ?s - student)
   :precondition (true)
   :effect  ((union (learnt ?s) (skills ?t)))))
```

Our goals will require set equality, set cardinality and expressions involving the skills learnt by the students jointly: any equivalent PDDL model must be able to express these goals. The number of skills could be large (at least up to 20 skills). A propositional encoding of all skill subsets is ruled out immediately, as 2^{20} subset objects are required and 2^{40} facts required to encode the subset relation. Therefore, some other encoding must be used. Another problem in using PDDL arises from the fact that numbers and quantification need to be used in concert. In order to perform set-union in PDDL, it is natural to use quantification. Reasoning about the cardinalities of sets is difficult using this model, as counters cannot be incremented within a PDDL `forall` structure, so it is impossible to update the size of a set in a single action. With neither a purely propositional model nor the ability to update numbers in iterative structures, it is necessary to introduce dummy actions to emulate the desired behaviour. Thus, we arrive at the following PDDL model:

```
(define (domain travelling-teacher)
  (:requirements :fluents :adl)
  (:types teacher student - person skill)
  (:predicates
      (teaches  ?t - teacher ?sk - skill)
      (learnt   ?s - student ?sk - skill)
      (teaching ?t - teacher ?s - student)
      (learnt2  ?sk - skill))
  (:functions
      (num-skills ?s - student))

(:action start
 :parameters (?t - teacher ?s - student)
 :precondition
   (not (exists (?ss - student ?tt - teacher)
                  (teaching ?tt ?ss)))
 :effect (teaching ?newt ?news))

(:action switch
 :parameters
  (?ot ?nt - teacher ?os ?ns - student)
 :precondition
  (and  (not (exists (?sk - skill)
               (and (teaches ?ot ?sk)
                    (not (learnt ?os ?sk)))))
        (teaching ?ot ?os))
 :effect
  (and  (not (teaching ?ot ?os))
        (teaching ?nt ?ns)))

(:action teach
  :parameters (?t - teacher ?s - student ?sk - skill)
  :precondition
   (and  (not (learnt ?s ?sk))
         (teaches ?t ?sk)
         (teaching ?t ?s))
  :effect
   (and  (learnt  ?s ?sk)
         (learnt2 ?sk)
         (increase (num-skills ?s) 1)))
```

This model works in the following way: teachers teach skills individually, so the cardinalities of the student skill sets can be incremented as each skill is taught. It is necessary to ensure that, once committed, a teacher teaches a student all of the skills he knows. This is achieved by the first precondition of the`switch` action. This action specifies that before a new teacher-student pair is selected there should be no skills taught by the current teacher that the current student has not learnt. We present a comparison of solving various problems from this domain. We vary the number of teachers, the number of skills and the type of goals.

Results We provide a comparison between PMTPlan and MetricFF on the skill-transfer domain documented in the previous section. PMTPlan uses the PMT model shown earlier. MetricFF uses models with counting and dummy actions. In order to compare the planners, we provided two

	PMTPlan		MetricFF	
Instance	Saturate	Equality	Saturate	Equality
inst-5-2	0.59	0.58	0.03	0.03
inst-5-3	0.70	0.61	0.04	0.03
inst-5-4	0.69	0.66	0.03	0.04
inst-5-5	0.81	0.80	0.03	0.04
inst-5-6	0.64	0.71	0.03	0.18
inst-5-7	0.61	0.57	0.04	2.38
inst-5-8	0.68	0.65	0.04	88.19
inst-5-9	0.61	0.64	0.04	
inst-5-10	0.61	0.58	0.03	
inst-5-12	1.02	0.61	0.04	
inst-5-14	2.79	0.63	0.04	
inst-5-16	0.97	0.71	0.04	
inst-5-18	1.03	0.68	0.03	
inst-5-20	2.61	0.72	0.03	

Table 2: Time taken (seconds) to solve a collection of instances from the skill-transfer domain. There are two different goal classes, Saturate and Equality. Each instance, of the form *inst-t-sk* has t teachers and sk skills.

different goal types that will be used on the same basic problems: saturation and equality. Both of these problem classes require the students to learn at least half of the skills; the saturation problems require all of the skills to be learnt by at least one of the students, while the equality problems require that the students all learn the same skills. In all of the test instances, there are two students and five teachers; we vary the number of skills in the instance. Table 2 shows the comparative performance of the two planners on the instances. It can be seen that MetricFF performs better on the saturate instances. On the equality instances it is PMTPlan that is most effective: MetricFF fails to solve any instances with more than 8 skills.

MetricFF can perform well on the saturate instances because it can avoid explicitly dealing with sets. The students gain skills monotonically, so we can model the problem to use simple counters to track whether all of the skills have been learnt globally. When reasoning about whether two students have the same skills it is impossible to avoid explicit reasoning over sets. This can either be done explicitly by introducing a set object (but we have already ruled that option out as impractical) or by using quantification as in the model shown above. This is shown to suffer from exactly the same impracticalities, and for any reasonable size sets, the PDDL approach is infeasible.

7 Conclusions

In this paper we have introduced the idea of Planning Modulo Theories, inspired by the successful SAT Modulo Theories framework. The proliferation of different planning applications, each requiring distinct extensions to the language and reasoning capabilities, motivates the development of a modular version of the standard language, PDDL, and integration of specialised solvers into general planners.

We have developed a heuristic forwards-search planner that solves problems in this new language. We have demonstrated that modelling and solving problems directly in a PMT framework can lead to models that are closer to the problem specification and can have more expressive goals than a PDDL equivalent model. We have also demonstrated that for some problem classes current state-of-the-art planners scale exponentially, whilst our PMT planner does not.

References

Barrett, C.; Stump, A.; and Tinelli, C. 2010. The SMT-LIB Standard: Version 2.0: http://combination.cs.uiowa.edu/smtlib/.

Cambon, S.; Alami, R.; and Gravot, F. 2009. A Hybrid Approach to Intricate Motion, Manipulation and Task Planning. *I. J. Robotic Res.* 28(1):104–126.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proc. Int. Conf. AI Planning and Scheduling (AIPS)*, 300–307.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008a. A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains. In *Proc. 18th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008b. Planning with problems requiring temporal coordination. In *Proc. 23rd AAAI Conf. on Artificial Intelligence*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2009. Temporal Planning in Domains with Linear Processes. In *Proc. 21st Int. Joint Conf. on AI (IJCAI)*.

de Moura, L. M., and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *Conf. on Tools and Alg. for the Construction and Analysis of Systems (TACAS)*.

Do, M. B., and Kambhampati, S. 2003. Sapa: A Multi-objective Metric Temporal Planner. *J. Art. Int. Res. (JAIR)* 20:155–194.

Dutertre, B., and de Moura, L. M. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. 18th Int. Conf. Computer Aided Verification (CAV)*, 81–94.

Dvorak, F., and Barták, R. 2010. Integrating Time and Resources into Planning. In *Proc. 22nd Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, 71–78.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *J. Art. Int. Res. (JAIR)* 20:61–124.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Art. Int. Res. (JAIR)* 27:235–297.

Geffner, H. 2000. Functional Strips: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. chapter 9, 187–212.

Golden, K. 1998. Leap Before You Look: Information Gathering in the PUCCINI Planner. In *Proc. Int. Conf. on AI Planning and Scheduling (AIPS)*, 70–77.

Gupta, S. K.; Nau, D. S.; and Regli, W. C. 1998. IMACS: A case study in real-world planning. *IEEE Expert and Intelligent Systems* 13(3):49–60.

Hoffmann, J.; Weber, I.; Scicluna, J.; Kaczmarek, T.; and Ankolekar, A. 2008. Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In *Proc. 8th Int. Conf. on Web Engineering (ICWE)*.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *J. Art. Int. Res. (JAIR)* 20:291–341.

Hoffmann, J. 2008. Towards Efficient Belief Update for Planning-Based Web Service Composition. In *Proc. 18th European Conf. on AI (ECAI)*, 558–562.

Maris, F., and Régnier, P. 2008. TLP-GP: New Results on Temporally-Expressive Planning Benchmarks. In *Proc. 20th IEEE Int. Conf. on Tools with AI (ICTAI)*.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical orderd planner. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*.

Nieuwenhuis, R.; Oliveras, A.; and Tinelli, C. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM* 53(6):937–977.

Reddy, S. Y.; Iatauro, M. J.; Kürklü, E.; Boyce, M. E.; Frank, J. D.; and Jónsson, A. K. 2008. Planning and monitoring solar array operations on the ISS. In *Proc. Scheduling and Planning App. Workshop (SPARK), ICAPS*.

Shin, J., and Davis, E. 2005. Processes and Continuous Change in a SAT-based Planner. *Art. Int. (AIJ)* 166:194–253.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *Art. Int. (AIJ)* 168(1-2):38–69.

Wolfman, S., and Weld, D. 1999. The LPSAT System and its Application to Resource Planning. In *Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI)*.

A Monte-Carlo Policy Rollout Planner for Pathfinding in Real-Time Strategy (RTS) Games

Munir Naveed, Diane Kitchin and Andrew Crampton
Department of Informatics
University of Huddersfield, England, UK.
Emails: {m.naveed, d.kitchin, a.crampton}@hud.ac.uk

Abstract

In this paper, we present a novel Monte-Carlo policy rollout algorithm (called MCRT-GAP) that uses a hybrid of focussed exploration of new actions and exploitation of promising actions to expand the look-ahead search from the current state in a Monte-Carlo simulation model. The initial estimates of the action values are computed using a combination of the distance heuristic and a collision estimate. To balance the exploration of new actions and the exploitation of already explored promising actions, MCRT-GAP uses a greedy action selection approach to exploit the best action and a random selection approach to explore new actions within a small set of useful actions (i.e. smaller than the size of the action set in a domain world). This small set is called a corridor. In this paper, we describe the motivation and the algorithmic details of MCRT-GAP. The paper also presents the theoretical properties of the algorithm.

INTRODUCTION

Monte-Carlo Policy Rollout algorithms (Bertsekas, Tsitsiklis, and Wu 1997) (Tesauro and Galperin 1996) (Kocsis and Szepesvári 2006) approximate action values using Monte-Carlo simulations. Monte-Carlo simulations require a model that can simulate the effects of an action using the state transition function and assign a reward for the state-action pair. These algorithms run several rollouts (or iterations from the current state) to approximate the values of the action applicable at the current state and select the action that has the highest value compared with other actions applicable at that state. These algorithms have been successfully applied to solve planning problems in deterministic domains (e.g. in Go (Lee et al. 2009)) and non-deterministic domains (e.g. Solitaire (Bjarnason, Fern, and Tadepalli 2009)).

These algorithms are suitable for online planning in domains with large state and action spaces, for example, real-time strategy (RTS) games. The domain world of a RTS game not only has a huge state space but it also imposes tight time constraints on the planning time. Furthermore, the state-space in a RTS game can be changed during the game play. Path planning is one of the most challenging tasks in a RTS game because it is required by all movable characters in a game. Many pathfinding approaches e.g. A* (Nilsson 1982), Navigational Mesh (Hamm 2008) and anytime algorithms (Likhachev, Gordon, and Thrun 2004) are not suitable due to the challenging issues (time constraints, dynamic changes and large state space) in a RTS game world.

In this paper, we describe a novel Monte-Carlo policy rollout algorithm that uses a simple and real-time Monte-Carlo simulation model. The work is an extension of MCRT (Naveed et al. 2011) - a Monte-Carlo simulation based path planner. MCRT is based on a sparse sampling scheme while MCRT-GAP is a pure policy rollout algorithm with one stage. The main contribution of MCRT-GAP is a heuristic based exploration scheme. The exploration of new actions in a Monte-Carlo simulation is kept focussed in a small space around the best action. The focussed region (also called a corridor) at a state is changed if the simulation model finds a better action than the current best action at that state. The new algorithm is evaluated using the benchmark problems of one map.

NOTATION

We are interested in solving path planning problems in a domain world that is partially visible, real-time and dynamic (like a typical RTS game). The planning agent knows about its current location and the goal location. The planning agent can see only a limited part of the domain world near its current position. The visible part is a circular region of radius of ten states with the current state of the planning agent at the centre of the visibility circle. It is assumed that the states that are beyond the visibility range are not occupied by static obstacles. In this section, we formalise the planning problem (that we are interested in).

Definition 1
The domain world has a finite set of states S and a finite set of actions A.

Definition 2 For each state $s \in S$, we define $A(s)$ to be the set of applicable actions at s where $A(s) \subset A$.

Definition 3
$Next(s, a)$ is a set of states that are possible to reach from s by taking action $a \in A(s)$.

Definition 4
$Transition(s, a)$ is a stochastic transition function that selects a next state $s' \in Next(s, a)$ for a state-action pair (s, a) where $s \in S$ and $a \in A(s)$.

Definition 5
$R(s, a)$ is a reward function ($R : S \times A \rightarrow \mathbb{R}$) that assigns a value to a state action pair (s, a) in a simulation model.

Definition 6
A simulation model is a part of MCRT-GAP that uses the function $Transition$ to randomly select the next state s' of a state-action pair (s, a) for all $s \in S$ and $a \in A(s)$ and then uses R to assign a reward value for the pair. The simulation model uses a probability distribution P to prioritise the selection of the most likely transition. $P : S \times A \times S \mapsto [0, 1]$ is a function that gives the likelihood of reaching a next state $s' \in Next(s, a)$ if an action $a \in A(s)$ is applied at state s. For example, if s_1, s_2 and s_3 are the possible next states of the current state s_c with action $a \in A(s_c)$ then $Next(s_c, a) = \{s_1, s_2, s_3\}$. Suppose the transition probabilities are $P(s_c, a, s_1) = 0.2$, $P(s_c, a, s_2) = 0.6$, $P(s_c, a, s_3) = 0.2$. This means it is most likely that the planning agent would reach s_2 if a is applied at s_c. If P is not available in a domain, MCRT-GAP updates the probabilities using the online interaction with the environment. The probability of moving to an occupied state s_b from s with any action $a \in A(s)$ is always zero i.e. $P(s, a, s_b) = 0$.

Definition 7
$V(s)$ is the state value and $Q(s, a)$ is the estimated value for action $a \in A(s)$ at state s. These values are computed using the simulation model of MCRT-GAP.

Definition 8
The best action $a_b \in A(s)$ at s has the highest estimated value at s. it is computed using equation (1) (adapted from (Tesauro and Galperin 1996)).

$$a_b = arg\,max_a(Q(s, a)) \tag{1}$$

MCRT-GAP

MCRT-GAP expands the look-ahead search from the current state s to a fixed depth D in a simulation. At s, it chooses the best action $a_s \in A(s)$ as a sample. It explores the actions at the successor state $s' = T(s, a_s)$ of s seen in the look-ahead search. At s', it selects an action $a' \in A(s')$ from a small list of action $A_c(s') \subset A(s')$ such that $a' \in A_c(s')$. The small list of actions is called a corridor. $A_c(s')$ is constructed using the best action a at s'. This sampling continues until the look-ahead search reaches depth D. For each state-action pair seen during the look-ahead search, MCRT-GAP keeps computing the reward values using R function. The state seen at depth D is evaluated using an admissible heuristic and this value is added to the sum of the rewards of all state-action pairs seen in the look-ahead search in a simulation. This accumulated value is used to update the expected long term reward of the action sampled at the current state i.e. $Q(s, a_s)$. If the returned value is bigger than the current $Q(s, a_s)$ then the estimated action value is modified otherwise it remains unchanged. If $Q(s, a_s)$ remains unchanged for $n_{limit} > 0$ consecutive simulations, then MCRT-GAP selects the best action at s by excluding a_s from the action list. If s is converged or if the time to run the simulations expires, it selects the best action at s as a plan and returns a for execution.

OBJECTIVE

In Monte-Carlo simulations, the exploration of new actions is an important and essential part of the search. It is important because the action values at the current state of the planning agent are estimated using the local information. However, the exploration of new actions that are not useful to solve the current planning problem is computationally very expensive for the planning search algorithm. It is intuitive to limit the exploration of the search space - during the Monte-Carlo simulations - within the vicinity (or corridor) of the effect(s) of the current best action of a state. The exploration scheme also gives importance to the actions that are less explored by increasing their chance of selection. The chance of selection is computed using $1/n(s, a)$ where $n(s, a)$ is the number of times an action $a \in A(s)$ is sampled at state s. The corridors are kept overlapping i.e. two corridors (each one by a different action) can have one or more common actions, so the exploration of the new actions can move from one corridor to another one in two consecutive simulations. At current state s, the exploration of the new action is performed if the estimated value of the best action $a_b \in A(s)$ does not change for some iterations.

ALGORITHMIC DETAILS

MCRT-GAP makes two main changes to the original MCRT (Naveed et al. 2011). First it uses the estimated action values to draw a sample at the current state of the planning agent. Second, it uses a corridor based exploration scheme to draw the action samples at the successor states of the current state. To explore the new actions, MCRT-GAP uses a small set of the actions that are relevant to the best action in the current state. MCRT-GAP performs exploration of the new actions within that corridor. The construction of the corridor for an action is done automatically. This can be done by using the angle between the directional lines of two actions if the actions are directional. A corridor for an action is built before the start of planning and is stored in memory for online use. The construction of a corridor for an action is a trivial process. We use angle between the directions of two actions to build a corridor. For example, a corridor of an action "MOVE TO NORTH" is a set "MOVE TO NORTH WEST", "MOVE TO NORTH", "MOVE TO NORTH EAST" where each member of the set has an angle of $45°$ or less with action "MOVE TO NORTH". The general overview of MCRT-GAP is given in Figure 1. At the current state s_c, if s_c is converged then MCRT-GAP returns the best action at s_c (Line 2, Figure 1). The convergence of a state s_c is discussed later in this paper. If s_c is not converged yet, then MCRT-GAP runs several rollouts depending on the time limit to estimate the action values at s_c. In each rollout, MCRT-GAP chooses (Line 6, Figure 1) the best action $a \in A(s_c)$ at s_c as a sample. If s_c is seen for the first time, or any of its actions are not sampled yet, then the action is selected randomly (from the unseen actions). The next state s_n of s_c with sampled action a is estimated by using the stochastic state transition function $Transition(s_c, a)$ (Line 7, Figure 2). The immediate reward $R(s_c, a)$ of the state-action pair is computed and stored in r_n (Line 8, Figure 2). s_n is expanded for a length of $depth - 1$

using a combination of exploration of new actions and exploitation of the previously seen actions. s_n also chooses the best action if each applicable action $a \in A(s_n)$ has been sampled at least once. If the best action a' at s_n has been sampled in the previous searches then it selects an action a randomly from the corridor of a'. The chances of selection of an action in the random selection in the corridor depends on the number of times the action is sampled in the previous searches. An action $a \in A(Corridor(a'))$ in the corridor of a' has more chances of selection as a sample than other members of $Corridor(a')$ if a is the least explored. The immediate reward of the state action pair (s_n, a) is computed and added to r_n (Line 15).

Function $MCRT - GAP(s_c, g)$
Read access $depth, timelimit, n_{limit}$;
1. IF s_c is converged
2. $a := ChooseGreedyAction(s_c)$;
3. RETURN the action a;
4. ELSE
5. REPEAT
6. $a := ChooseAction(s_c)$;
7. $s_n := Transition(s_c, a)$;
8. $r_n := R(s_c, a)$;
9. FOR: i=1 to depth -1
10. $a := ChooseAction(s_n)$;
11. IF a is invalid at s_n THEN
13. $Q(s_c, a) := 0$;
12. CONTINUE;
13. IF a have previously been sampled at s_n THEN
14. $a := ChooseActionFromCorridor(a)$;
15. $r_n := r_n + R(s_n, a)$;
16. $s_{next} := T(s_n, a)$;
17. $s_n := s_{next}$;
18. END FOR
19. $r_n := r_n + 1/dist(s_n, g)$;
20. $n(s_c, a) := n(s_c, a) + 1$;
21. IF $Q(s_c, a) < r_n$ THEN
22. $Q(s_c, a) := r_n$;
23. $timelimit - -$;
24. UNTIL $(timelimit > 0)$;
25. $a := ChooseGreedyAction(s_c)$;
26. RETURN the action a
End $MCRT - GAP$

Figure 1: High Level Design of MCRT-GAP

The next state s_{next} of s_n is determined by using $Transition$ function (Line 16). s_{next} is expanded in the same way that s_n is done. This continues for $depth - 2$ iterations. At the depth $depth$ of the look-ahead search, the leaf node s_n is evaluated using the distance heuristic $dist$ and the inverse of the heuristic is added to r_n (Line 19). If the current estimate of the long term reward r_n of the state action (s_c, a) is greater than or equal to the previous value i.e. $Q(s_c, a)$, then MCRT-GAP updates $Q(s_c, a)$ (Line 21). MCRT-GAP keeps running the simulations - with s_c as the start node for each simulation - until the time is out. At the end of the simulations, MCRT-GAP selects the best action at s_c and returns it for execution. The details of the immediate reward function $R(s_n, a)$ are given in Figure 2. This reward function has been used in the previous work (Naveed et al. 2011) and (Naveed, Crampton, and Kitchin 2010).

Function $R(s_n, a)$
Read access MDP, g;
1. $s_{next} := Transition(s_n, a)$;
2. $rw := \frac{\|\{s_t : p(s_n, a, s_t)) > 0 \, \forall s_t \in Next(s_n, a)\}\|}{dist(s_{next}, g)}$;
3. RETURN rw
End R

Figure 2: Reward Function

R (Figure 2) computes the reward of a state-action pair seen during Monte-Carlo simulations. The reward is computed using the current goal g of the planning problem. R estimates the transition (i.e. next state s_{next}) of the given state-action pair (Line 1, 2) and estimates the distance between s_{next} and goal state g using a heuristic function $dist$. R uses the list of all possible states (i.e. $Next(s_n, a)$) reachable directly from the state-action pair (s_n, a). The number of states in $Next(s_n, a)$ which are unoccupied divided by the distance heuristic $dist(s_{next}, g)$ is the immediate reward of (s_n, a) (Line 2). A state $s_t \in Next(s_n, a)$ is unoccupied if the probability of reaching s_t from s_n with action $a \in A(s_c)$ is greater than zero.

COMPLEXITY ANALYSIS

MCRT-GAP selects an action from current state s_c using $ChooseAction$ (Lines 6, Figure 1). It takes $O(1)$ to select an action randomly at s_c if s_c is seen first time. If all actions at s_c have been sampled in the previous searching efforts, then it takes $O(|A(s_c)|)$ to select the best action at s_c (Line 1, Figure 1). The average time complexity of $ChooseAction$ is $O(1)$ and the worst case time complexity is $O(|A|)$. The worst case time complexity of $ChooseActionFromCorridor$ (Line 14, Figure 1) is $O(C)$ where C is the size of the corridor. The space complexity of MCRT-GAP per simulation is $O(D + C|A|)$. it has the worst case time complexity of $O(D|A|C)$ and the average time complexity per simulation is $O(DC)$.

CONVERGENCE AND OPTIMALITY

From Figure 2 (Line 22), it is obvious that $Q(s, a)$ for any state $s \in S$ and action $a \in A(s)$ remains the same or increases with the increase in the number of rollouts. At any simulation time $t > 1$, the $Q_t(s, a) \forall s \in S,\ a \in A(s)$ is monotonically non-decreasing as shown in Equation 2.

$$0 < Q_{t-1}(s, a) \leq Q_t(s, a),\ \forall t > 1 \tag{2}$$

.

MCRT-GAP also preserves the monotonicity of the state value $V(s)$ for each state $s \in S$. The optimal state value $V^*(s)$ for state $s \in S$ is the supremum of the monotonic sequence $V(s)$ and the optimal action value $Q^*(s, a)$ is $sup(Q(s, a))$ for $a \in A(s)$. In other words, the optimal

action value of a state action pair (s, a) is an upper bound on the monotonic sequence of $Q_t(s, a)$ for all $t > 1$ (Equation (2)).

Lemma 1. *A monotonic function (of real numbers) converges if it is bounded.* (Copson 1970)

THEOREM 1. *MCRT-GAP eventually finds the optimal action value $Q(s, a)$ for a given state s and action $a \in A(s)$ if repeated for several iterations.*

Proof:

The proof follows Lemma 1. MCRT-GAP generates a monotonic sequence $Q(s, a)$ of action values for the state action pair (s, a) by running several simulations from $s \in S$. It is given that the optimal value $Q^*(s, a)$ is an upper bound of monotonic sequence $Q(s, a)$. Therefore, MCRT-GAP eventually converges to the optimal action value for a given state action pair (s, a). □

It is not clear how many iterations MCRT-GAP requires to converge the action value function $Q(s, a)$ to an optimal value for a given state s and action a. For practical reasons, we use an error bound based on the algorithmic parameter $n_{limit} > 0$ to define the convergence of the action value function Q. If MCRT-GAP does not change the action value of a state-action pair (s, a) for n_{limit} consecutive simulations, then it is assumed that $Q(s, a)$ has converged relative to parameter n_{limit}. If all applicable actions $a \in A(s)$ at s are converged with respect to n_{limit} then s is declared a converged state relative to n_{limit}.

PATH PLANNER

MCRT-GAP is embedded in a complete real-time planner. The real-time planner interleaves planning and plan execution. In each planning episode, MCRT-GAP plans an action at the current state of the planning agent and the action is returned for execution. After execution, the planning agents move to a new state. At the new state, the planner selects an action using MCRT-GAP, executes it and moves to another state. This process continues until the planning agent reaches the goal state. A high level design of the planner is given in Figure 3.

Procedure $Planner$
Read(s_o, g);
1. initialise parameters of MCRT-GAP and state $s := s_o$;
2. REPEAT
3. $a := MCRT - GAP(s, g)$;
4. $s := Execute(a, s)$
5. $UpdateP$
6. UNTIL $s.pos = g$;

End $Planner$

Figure 3: A Real-Time Planner

The planner reads a planning problem with the initial state s_o of the planning agent and the goal state g it tends to move to. The planner initialises the MCRT-GAP parameters e.g. $timelimit$, n_{limit} and look-ahead depth $depth$ (Line 1, Figure 3). The planner calls MCRT-GAP for the initial state s_o to select an action a to move towards g (Line 3). The action selected by MCRT-GAP is executed at s_o and the agent moves to a new state s (Line 4). The state transition probabilities are updated for the transition (Line 5).If s is a goal state then the planner stops otherwise it keeps planning and executing until the planning agent finds the goal state.

RELATED WORK

Tesauro (Tesauro and Galperin 1996) explored the policy rollout algorithm in a stochastic board game called Backgammon. The simulation model in Tesauro's work takes several iterations per move to decide an action at the current move. This approach is expensive for a real-time application. Kearns et al. (1999) present a spare sampling based approach that generates a look-ahead tree of fixed depth H but each action applicable at state (seen during the look-ahead search) is sampled C times in a simulation. The number of samples in a simulation are exponential in H. To avoid exploring all actions in a sparse sampling scheme, Auer et al. (2002) demonstrate an adaptive action sampling approach (called Upper Confidence Bounds or UCB) that selects only one action per state in the look-ahead search in a simulation. It selects the best action as a sample at a state. To balance the exploration of new actions and exploitation of the best action, Auer et al. present upper bounds on the selection of an action as a sample at a state. However, Auer et al.'s sampling approach continues the exploration of new actions forever. Kocsis and Szepesvári (2006) present a variation of UCB, called Upper Confidence Bounds applied to Trees (UCT), that performs selective action sampling in a policy rollout fashion. The default rollout policy is random. UCT has been successful in Go (Lee et al. 2009) and Solitaire (Bjarnason, Fern, and Tadepalli 2009) games. However, UCT and UCB are applicable in a domain if the action values are in the range [0,1]. In RTS games, action values are beyond this range. Balla et al. (Balla and Fern 2009) present a variation of UCT in a RTS game to solve the tactical assault problem. The variation of UCT uses a reward function that can optimise one parameter (time or health factor) in a simulation model.

MCRT-GAP performs exploration of the new actions using the heuristic function. It avoids the exploration of the actions that are not useful according to the current estimates. It exploits the knowledge it discovers in the previous efforts to draw samples in the look-ahead search. It estimates the action values for a given state under the tight real-time constraints. MCRT-GAP also shares some characteristics with the recent real-time path planners like LSS-LRTS (Koenig and Sun 2009) and Real-time D* Lite (RTD) (Bond et al. 2010). RTD and LSS-LRTA interleave planning and plan execution like MCRT-GAP. LSS-LRTA only updates the cost values of the actions if they are increasing. It does not decrease the cost of an action if it is reduced due to a dynamic change in a domain world.

EXPERIMENTAL DETAILS

MCRT-GAP is empirically evaluated using a benchmark map called $Arena2$. Arena2 (Figure 4) has 281×209 states. We use the MAI tool to implement MCRT-GAP and the

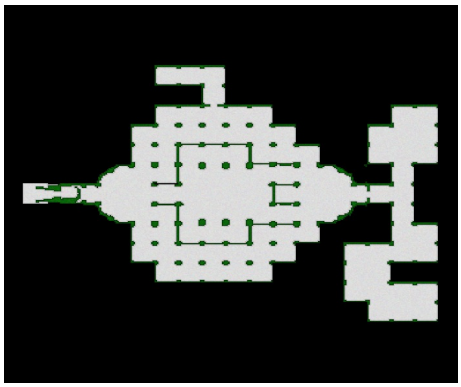


Figure 4: Arena2 Map

planner. The tool has been used in a previous study (Bond et al. 2010). It provides an easy to use programming environment for the exploration of real-time path planning in a partially visible and dynamic world. The tool also supports STRIPS planning. For the benchmark map, the tool reads the first 900 planning problems and passes them to the planner sequentially. The maximum path length in this problem set is 359.397. The planner reads a planning problem and solves it. The tool imposes a limit on the solution time. If the planner does not reach the goal state in the allowable time, then the problem is declared unsolvable by the planner.

IMPLEMENTATION DETAILS

To load the benchmark map in the MAI tool, we use HOG-GridWorld library. The MAI tool encodes actions as a navigational Compass. The action set is MOVE TO NORTH, MOVE TO EAST, MOVE TO SOUTH, MOVE TO WEST, MOVE TO NORTH EAST, MOVE TO SOUTH EAST, MOVE TO SOUTH WEST, MOVE TO NOTH WEST. A state is a (x, y) location on the map. $n(s, a)$ for each sampled action a at state s, action value $Q(s, a)$ and the transitions probabilities P are stored in the hash tables. $n(s, a)$, P and Q are empty at the start of the planning and are populated during the planning episodes. P is updated after a state transition occurs (Line 5, Figure 3) or when a state occupied by an static obstacle is seen. if a state-transition has an entry in P then it is accessed through the key otherwise MCRT-GAP uses the default value. The default value for a state transition probability is 0.000001. If a state-action (s, a) does not have an entry in Q table, then MCRT-GAP assumes that a is not sampled at s. Every sampled action a at any state $s \in S$ has an entry in Q. The states that are not seen during the planning search, they are not added to Q.

PERFORMANCE MEASUREMENT

The performance of MCRT-GAP is measured using two parameters: time to solve 900 planning problems and sub-optimality of the solution by the planner. The sub-optimality is measured using a ration of l_p to l_o where l_p is the length of the solution by the planner and l_o is the length of the optimal path given in the benchmark. The higher values of these two parameters represent poor performance of the planner.

Timelimit	Depth	Avg Time (sec)	Avg Subopt
10	3	87990.20	10.66
10	9	137728.53	9.25
10	12	200145.47	8.72
10	15	208809.61	8.99
30	3	1136.68	15.64
30	12	64843.06	8.54
30	15	65091.56	9.0

Table 1: Time and Suboptimaity (Subopt) of MCRT-GAP planner (with $n_{limit} = 1$) on Arena2.

RESULTS

MCRT-GAP Planner is run on seventeen different machines of same hardware and software configurations. Each machine has Intel(R) Core (TM) 2 Quad processors each of speed 2.6 GHz CPU speed and 8 GB ram. MCRT-GAP planner is run for different $timelimit$ and $depth$ values. The visibility of the planning agent is kept the same in all experiments. The experimental results on Arena2 are shown in Table 1. These results are the average of five runs. There are twenty five problems in first 900 planning problems that are not included in the set because MCRT could not solve them under the given time limit. The initial results show that the look-ahead depth plays an important role in reducing the sub-optimality of the planning algorithm, however, it also increases the time to solve the planning problems. Increasing $timelimit$ can reduce the time to solve the planning problems. This is due to the convergence of the states with respect to n_{limit}. $n_{limit} = 1$ means if the action value of a state-action pair does not change between two consecutive simulations, then this action is not explored in the future search efforts. With small look-ahead depth e.g. 3, the states are converged quickly relative to $n_{limit} = 1$ and at a converged state MCRT-GAP selects the best action using the previously computed estimates, therefore, it takes a small duration to solve a planning problem but at the cost of optimality. The sub-optimality is highest at the shallow look-ahead depth with a higher simulation time.

CONCLUSION AND FUTURE WORK

We present a work in progress in this paper. The paper presents a new policy rollout algorithm, called MCRT-GAP, that learns the action values online using a simulation model. MCRT-GAP explores new actions in a focussed part of the look-ahead search to avoid exploration of the actions that are not useful. The paper describes the algorithmic details and the theoretical details of the algorithm. The results of the initial experiments are described on a benchmark map $Arena2$. The initial results show that the performance of MCRT-GAP depends on look-ahead depth and convergence parameter n_{limit}. The quality of solution by MCRT-GAP is poor in the initial results. However, the results also indicate that the higher values of n_{limit} and $timelimit$ can be useful to improve performance of the algorithm.

In future work, we are aiming to extend the experimental work for a detailed analysis of the relationship between $depth$, $timelimit$ and n_{limit} in MCRT-GAP. In future experiments, we plan to use six more maps from the benchmark problems. These maps are Den401d, Lake303d, Orz103d, Orz701d, Orz702d and Orz900d. These maps vary in size and the number of obstacles (or blocked states). The results of MCRT-GAP will be compared against RTD, LSS-LRTA and MCRT.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.* 47:235–256.

Balla, R., and Fern, A. 2009. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 40–45.

Bertsekas, D. P.; Tsitsiklis, J. N.; and Wu, C. 1997. Rollout algorithms for combinatorial optimization. *Journal of Heuristics* 3:245–262.

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *Proceedings of the 19th International Conference on Automated Planning & Scheduling*.

Bond, D.; Widger, N.; Ruml, W.; and Sun, X. 2010. Real-Time Search in Dynamic Worlds. In *Proceedings of the Third Annual Symposium on Combinatorial Search* .

Copson, E. 1970. On a generalisation of monotonic sequences. In *Proc. Edinburgh Math. Soc.*, volume 17, 159–164.

Hamm, D. 2008. Navigational Mesh Generation: An empirical approach. In Rabin, S., ed., *AI Game Programming Wisdom 4*. Charles River Media. 113–114.

Kearns, M.; Mansour, Y.; and Ng, A. Y. 1999. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*, 1324–1331. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, 282–293.

Koenig, S., and Sun, X. 2009. Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents. *Journal of Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

Lee, C.; Wang, M.; Chaslot, G.; Hoock, J.; Rimmel, A.; Teytaud, O.; Tsai, S.; Hsu, S.; and Hong, T. 2009. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transaction on Computational Intelligence and AI in Games* 1:73–89.

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. Ara*: Anytime a* with provable bounds on sub-optimality. In *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 16: PROCEEDINGS OF THE 2003 CONFERENCE (NIPS-03*. MIT Press.

Naveed, M.; Crampton, A.; Kitchin, D.; and McCluskey, T. 2011. Real-Time Path Planning using a Simulation-Based Markovian Decision Process. In *To appear: AI-2011 Thirty-first SGAI International Conference on Artificial Intelligence*.

Naveed, M.; Crampton, A.; and Kitchin, D. 2010. Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games. In *Proceedings of PlanSIG 2010. 28th Workshop of the UK Special Interest Group on Planning and Scheduling*, 125–132.

Nilsson, N. J. 1982. *Principles of Artificial Intelligence*. Springer-Verlag.

Tesauro, G., and Galperin, G. R. 1996. On-line policy improvement using monte-carlo search. In *NIPS'96*, 1068–1074.

Hierarchical Task Based Process Planning For Machine Tool Calibration

S. Parkinson, A. P. Longstaff, G. Allen, A. Crampton, S. Fletcher, A. Myers

Centre for Precision Technologies, School of Computing and Engineering, University of Huddersfield, Queensgate, Huddersfield, HD1 3DH, UK

simon.parkinson@hud.ac.uk

Abstract

A literature search has indicated that artificially intelligent planners have not previously been used to address the planning problem of machine tool calibration, even though there are potential advantages. The complexity of machine tool calibration planning requires the understanding and examination of many influential factors, such as the machine's configuration and available instrumentation. In this paper we show that machine tool calibration planning can be converted into a Hierarchical Task Network by the process of task decomposition. It is then shown how the Simple Hierarchical Ordered Planner architecture can be used to provide all the identified complete process plans in a given time frame, and secondly, how the branch-and-bound optimisation algorithm can find the optimal solution in the same frame. The results for generating the process plans and optimal process plans for both a three and five axis machine are evaluated to examine the planner's performance.

Introduction

Generating process plans automatically is a challenging yet advantageous quality. The economic advantages are seen as significant to engineers (Satyandra, 1998). This is because the ability to create both an efficient and complete process plan can result in minimising the risk of problems occurring that could ultimately result in excessive expenditure. This is true for the process of machine tool calibration planning (Bringmann et al., 2008).

The requirement to manufacture more accurate parts and minimise manufacturing waste is resulting in the continuing requirement for machine tools which are more accurate. Therefore, machine tool calibration is required regularly to gain an understanding of a machine's capability. When planning a machine tool calibration, an engineer will derive a calibration plan based on many influencing factors. For the work undertaken within this paper, we are only concerned with (1) the machine's configuration of constituent parts, (2) the errors associated with the machine, (3) the available instrumentation, and (4) scheduling and resource constraints. Other constraints, for example, the possibility of different test methods, have been excluded in an attempt to identify a simplified set that allows for the creation of an initial prototype. The complexity and quantity of knowledge that is required and processed during machine tool calibration planning is sufficient to require the use of a computational reasoning.

The way that the process of machine tool calibration can be broken down into smaller tasks makes it well suited to being represented by a Hierarchical Task Network (HTN). An HTN planner will recursively decompose nonprimitive tasks into smaller subtasks until primitive tasks are reached which can be performed directly using planning operators (Nau et al., 2003). The literature suggests that HTNs have been widely used as a planning technique because they are a convenient way to write problem-solving recipes that correspond to how a human domain expert would think about solving the problem (Ghallab et al., 2004). The Simple Hierarchical Ordered Planner 2 (SHOP2) is a domain-independent planning system that allows for the implementation of a domain-specific problem-solving planner (Goldman, 2011). For this reason, the SHOP2 system has been selected for use.

In this paper, a solution to the problem of machine tool calibration planning is presented by adopting a cross-discipline approach to develop a Hierarchical Task Network (HTN) which is implemented using the SHOP2 architecture. First, a literature review of HTN's being applied to engineering planning problems is presented. Next, the problem of machine tool calibration planning is described in more detail. This leads to the implementation of an initial HTN system using SHOP2. The results of the prototype solution are presented and discussed describing the scope for future work.

Literature survey

There is currently an absence of any literature indicating the advancement of process planning for machine tool calibration. For this reason, planning advancements in other engineering processes of a similar nature are

examined to identify any intelligent approaches.

Significant effort has been spent in the improvement of automated planning techniques for industrial applications. There have been many successful implementations within mechanical engineering. The Interactive Manufacturability Analysis Critiquing System (IMACS) was developed to evaluate the manufacturability of machined parts and to suggest improvements to increase the ease of manufacture (Satyandra, 1998). The system processes the geometric features of a Computer Aided Design (CAD) model to determine the required machining operations. The authors have identified the complexities with populating a general purpose planner with domain-specific knowledge. Instead, they integrate the domain-specific knowledge into the planning algorithms themselves. The finished IMACS made use of an HTN planning system using a depth-first branch-and-bound search strategy to find the optimal complete process plan.

A similar application named the computer-aided process planning (CAPP) system was also developed to find both a complete and optimal solution for the manufacturing of a part based on (1) a description of the blank part, (2) description of the finished part, (3) available resources, and (4) technical knowledge (Deák et al., 2001). The CAPP system is represented in HTN form by using the SHOP architecture. The motivation behind the selection of an HTN is very similar to that as IMACS. It was found that traditional general purpose planners did not allow for the specification of the domain-specific knowledge.

In conclusion, it is evident that significant work throughout the 1990s has been performed to optimise the process of manufacturing parts, which has been largely successful. The significance of earlier work can be seen in that many commercially available Computer Aided Manufacture (CAM) packages now implement intelligent functionality to improve the part's design and proposed machining operations to reduce both manufacturing time and cost (Delcam, 2011).

Previous work has shown that the process of machine tool calibration can be represented in first-order logic (Parkinson et al., 2011) to provide a means of modelling all the possible tests that can be performed during the calibration of a specific machine tool. However, this knowledge needs interpreting to decide on the most feasible set of tests to reach the state of having a calibrated machine.

Machine tool calibration

As previously identified in the introduction, machine tool calibration is based on many influencing factors. For the context of this paper, the following section contains enough information regarding the influencing factors of machine tool calibration to allow the reader to understand the planning problem in sufficient detail.

Machine configuration

A machine can be designed and constructed in many different ways to perform its task. Figure 1 shows a machine tool with three perpendicular linear axes, while Figure 2 shows a gantry machine tool with three perpendicular linear axes and two rotary axes. In addition to the number of linear and rotary axes, the configuration (stacking) of these axes can cause errors to propagate differently throughout the machine. The configuration of a machine tool will determine how many error components it has. While there are a few common machine configurations, there are a lot of different configurations which require in-depth consideration to identify all their error components.

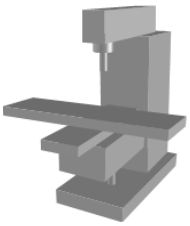


Figure 1 - Three-axis machine tool

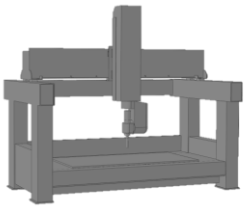


Figure 2 - Five-axis machine tool

Machine errors

The configuration of the machine's constituent parts determines the potential geometric errors that a machine might have. The geometric errors associated with linear and rotary axes are well known (Bohez et al., 2007). For example, a linear axis will have six error components (six-degrees-of-freedom) plus a squareness error with the perpendicular axis, which is illustrated in Figure 3. From this it is possible to deduce that a three axis machine tool will have in total 21 geometric errors (Ramesh et al., 2000)

A machine tool will, however, actually experience more error sources such as thermal, dynamic and non-rigid (Mekid, 2009). For the scope of this paper, only the calibration planning problem for geometric errors in machine tools is considered.

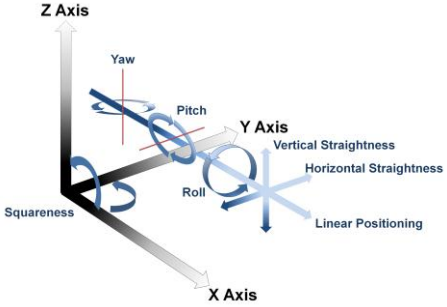


Figure 3 - Six-degrees of freedom and squareness errors for the X-axis of a machine tool with three perpendicular linear axes

Instrumentation

The extensive variety of instrumentation available for performing a machine tool calibration adds complexity to

deciding the optimum solution when measuring each error component. There are many different reasons why a specific instrument might be selected. The following list supplies two sample Key Performance Variables (KPVs) which would influence the instrumentation selection.

1. The time to install and align the equipment may be lower
2. The resolution and accuracy of the instrument might be greater

For example, measuring the y-axis linear positioning error using the Renishaw XL-80 laser interferometer would require the configuration of the optics as seen in Figure 4. Next, measuring the y-axis pitch error would require the use of the optics aligned as seen in Figure 5. However, because the optics' base and the laser are already aligned, it is possible to carefully exchange the optics with only a small adjustment.

Given that the number of potential KPVs for selecting a given instrument is large, the work undertaken in this paper will only be concerned with the time required to install and setup the instrument, and the time to adjust the equipment from a previous setup.

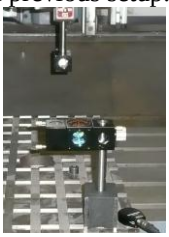


Figure 4 – Linear position optics

Figure 5 - Pitch optics

Scheduling

Once a decision has been made to establish which instrument is to be used to measure each error component, the ordering of these measurements needs to be decided. As previously highlighted, there are many cases where the instrumentation will only need to be readjusted slightly to allow the measurement of two different error components. For this reason, finding the optimal sequence of measurements can reduce the time taken to perform the calibration by saving on instrumentation setup time.

HTN implementation

As identified in the introduction, the planning problem of machine tool calibration is well suited to being represented as an HTN. The following section shows how machine tool calibration was broken down into smaller tasks to create an HTN.

Task decomposition

Task decomposition is the process of breaking tasks into smaller tasks until primitive actions are reached. Figure 6 shows the abstract task decomposition for calibrating a machine tool, which takes into consideration what has been regarded as the main calibration tasks. A description for each primitive subtask can be found in the following list:

1. Find all linear errors based on the machine's configuration.
2. Find all rotary errors based on the machine's configuration.
3. Find all cross-axis errors based on the configuration of the linear and rotary axes.
4. Select an error component for measuring.
5. Select the suitable equipment for measuring the error.
6. Setup the equipment in a suitable way to measure the error component.
7. Measure the error component using the instrumentation and the current setup.

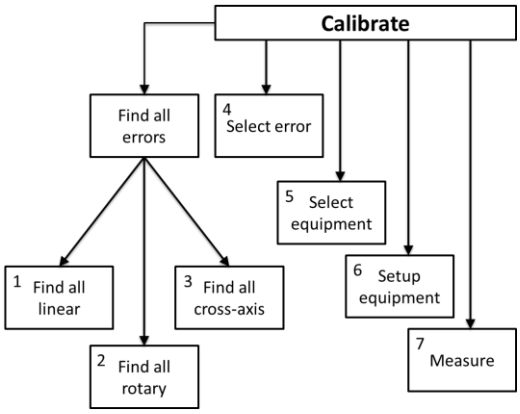


Figure 6 - Task decomposition tree

The process of performing this manual task decomposition to convert the nonprimitive task of machine tool calibration into the primitive tasks will serve as the basis for creating an HTN network.

System definition

An HTN planning problem is a 4-tuple

$$P = (s0, w, O, M)$$

Where $s0$ is the initial state, w is the initial task network, O is the set of operators, and M is the set of HTN methods. Applying this to the planning problem of machine tool calibration would mean that $s0$ is the initial non calibrated state of the machine tool, and w is the initial task network for performing the calibration. O would be the set of operators which describe how to perform a primitive task which cannot be decomposed any further to reach the state of a calibrated machine. M is the set of methods which perform the task decomposition based on a logical precondition.

Initial state

The initial state definition *s*0 can be regarded as the facts that describe the current planning problem. As previously described, a primitive version of machine tool calibration can be represented in first-order logic (Parkinson et al., 2011). An expansion of this work is used here to define the initial state. The following shows a sample of the facts that describe *s*0:

```
;;Axis
(axis X)

;;Axis Type
(linear X)

;;Linear geometric error + cost in priority
(linear-geometric-error PITCH 10)

;;Equipment + setup and adjust time (mins)
(equipment LASER 10 5)

;;Measurement + cost of performing (mins)
(measures PITCH LASER 10)
```

The size of *s*0 for representing a three axis machine tool which is used for testing in section contains a total of 34 facts. For a comprehensive representation, additional parameters would be included. For example, axis length, feed rate, number of targets, dwell time, etc. These additional parameters will be included once a working prototype has been achieved.

Initial task network

The initial high level task network *w* for performing a machine tool calibration is simply:

```
(perform-calibration)
```

In practice, it is highly possible that the initial task network might be more detailed than this. It is possible that there will be machine-specific preconditions that must be considered.

Operators

An operator is a description of how to perform a primitive task, which cannot be decomposed further. An operator's description is:

```
(:operator h P D A [c])
```

Where `h` = head, `P` = preconditions, `D` = delete list, `A` = add list, `c` = optional cost.

The following set *O* contains the operators that are required for the HTN implementation.

```
(:operator    (!select-error ?a ?e ?c)
              ((meas_required ?x ?y ?c ))
              ()
              ((meas_selected ?a ?e ))
              (*1 ?c))

(:operator    (!select-equip ?a ?e ?i ?mc ?c ?ac)
              ((meas_selected ?a ?e ))
                ()
                ((equip_selected ?a ?e ?i ?mc ?c ?ac)))

(:operator (!set-up-equip ?a ?e ?i ?mc ?c )
                ((equip_selected ?a ?e ?i ?mc ?c ?ac))

                ()
                ((equip_setup ?a ?e ?i ?mc))
                (* 1 ?c))

(:operator    (!adjust-equip ?a ?e ?i ?mc ?c ?pe
              ?pmc ?ac)
              ((equip_selected ?a ?e ?i ?mc ?c ?ac))
              ()
              ((equip_setup ?a ?e ?i ?mc))
              (* 1 ?ac))

(:operator    (!measure ?a ?e ?i ?mc )
              ((equip_setup ?a ?e ?i ?mc )(equipment
              ?i ?c ac)
              (equip_selected ?a ?e ?i ?mc ?c ?ac))
              ((meas_required   ?a ?e
              ?c)(meas_selected ?a ?e)
              (equip_selected ?a ?e ?i ?mc ?c ?ac)
              (equip_setup ?a ?e ?i ?mc ))
              ()
              (* 1 ?mc))

(:operator    (!!assert ?g)
              ()
              ?g
              0)

(:operator    (!!remove ?g)
              ?g
              ()
              0)
```

Methods

This section contains the set of methods *M* for performing the task decomposition in the HTN. The methods can been seen in the decomposition tree shown in Figure 6. Other methods can be seen here which are responsible for keeping track of the current error, instrumentation and instrumentation setup selection. A method's description is:

```
(:method h [n1] C1 T1 [n2] C2 T2.. [nk]Ck Tk)
```

Where `h` = head, n_i = name for each succeeding C_i T_i pair, C_i = precondition, T_i = task list (tail).

```
(:method    (perform-calibration)
            ()((find-all-required)(calibrate)))

(:method    (find-all-required)
            ((linear ?a)(linear-geometric-error ?e
            ?c)
            (not(meas_required ?a ?e ?c)))
            ((!!assert ((meas_required ?a ?e ?c)))
            (find-all-required))
            ((linear ?axis)(cross-axis-error ?e
            ?c)(not(meas_required ?a ?e ?c)))
            ((!!assert ((meas_required ?a ?e ?c
            )))(find-all-required))
            ((rotary ?axis)(rotary-geometric-error ?e
            ?c)
            (not(meas_required ?a ?e ?c)))
            ((!!assert ((meas_required ?a ?e ?c)))
            (find-all-required))
```

```
            nil
            nil)

(:method    (calibrate)
            ((meas_required ?a ?e ?c)
            (not(meas_selected ?a ?e ))
            (not(measured ?a ?e )))
            ((!select-error ?a ?e ?c )(select-
            equipment)(calibrate))
            nil
            nil)

(:method    (select-equipment)
            ((meas_selected ?a ?e )(equipment ?i ?c
            ?ac)
            (measures ?e ?i ?mc)
            (not(equip_selected ?a ?e ?i ?mc ?c
            ?ac)))
            ((!select-equip ?a ?e ?i ?mc ?c ?ac)
            (set-up-equipment)(select-equipment))
            nil
            nil)

(:method    (set-up-equipment)
            ((equip_selected ?a ?e ?i ?mc ?c ?ac)
            (not(previous_error ?a ?pe ?i ?pmc))
            (not((equip_setup ?a ?e ?i ?mc))))
            ((!set-up-equip ?a ?e ?i ?mc ?c)(measure-
            error)
            (set-up-equipment))

            ((equip_selected ?a ?e ?i ?mc ?c ?ac)
            (previous_error ?a ?pe ?i ?pmc)
            (not((equip_setup ?a ?e ?i ?mc))))
            ((!adjust-equip ?a ?e ?i ?mc ?c ?pe ?pmc
            ?ac)
            (measure-error) (set-up-equipment))
            nil
            nil)

(:method    (remove-previous)
            ((previous_error ?a ?e ?i ?mc))
            ((!!remove((previous_error ?a ?e ?i
            ?mc)))
            (remove-previous))(not(previous_error ?a
            ?e ?i ?mc))
            nil)

 (:method   (measure-error)
            ((equip_setup ?a ?e ?i ?mc)(meas_required
            ?a ?e ?c))
            ((!measure ?a ?e ?i ?mc )
            (remove-previous)
            (!!assert((previous_error ?a ?e ?i ?mc)))
            (!!remove ((meas_required ?a ?e ?c))))
            nil
            nil)))
```

Branch-and-bound

The branch-and-bound algorithm is used for finding the lowest cost solution to optimisation problems (Nau et al., 2003). The computational expense for exploring every potential partial plan to find the optimal complete solution can be large, or even infinite. For example, a machine tool with three linear axes which each have six error components plus three squareness would result in the generation of 21 calibration tasks. There is then a potential 21^{21} sequences. To find the plan with the lowest cost, each of potential plans must be explored and evaluated. The number of potential sequences will increase with the addition of different instrumentation and measurement techniques. SHOP2 allows for the use of the branch-and-bound algorithm without any change to the HTN domain or problem specification.

Cost calculation

A SHOP2 operator also expresses a cost for performing the primitive task. The operators used in the machine tool calibration HTN have a cost assigned which is originally acquired from the initial state facts. The motivation behind an operator's cost is explained below:

1. Error selection – this is the importance of an error component. An error component that is regarded as having a high significance, or that should be measured first, is assigned a lower cost value.
2. Equipment setup – the cost in minutes that is required for setting up the instrumentation out of the box.
3. Equipment adjustment – this is the cost in minutes for adjusting the equipment. For example, realigning the optics of a laser interferometer.
4. Performing the measurement – this is the cost in minutes for measuring the error component using the selected equipment.

The implementation of an operator's cost can allow the branch-and-bound algorithm to find the optimal solution in a lower computational time.

Results

To evaluate the HTN's performance, empirical observations have been made using the two following planning problems. For the scope of the work presented in this paper, the selection of the planning problem and assigned cost values is arbitrary and not comprehensive. The cost values do however correctly show that some error components are more important than others and that different instrumentation requires a different length of time for setting up, adjusting and taking the measurement.

1. A machine tool with three linear axes. Each linear axis will have six geometric plus one squareness error components. There are a total of five different instruments available, and each error component can be measured by using at least two of the available instruments. The size of $s0$ for this problem is 53.
2. A five axis machine tool with three linear and two rotary axes. Each linear axis will have six geometric plus one squareness error components, and each rotary axis will have nine error components. There will also be a total of five different instruments available, and each error component can be measured by using at least two of the available instruments. The size of $s0$ for this problem is 99.

The experiments were carried out on an Ubuntu 10.04 virtual machine with 2GB of RAM and two cores of the host's AMD Phenom™ II X4 970 assigned. SHOP2 (v2.8.0) was executed in the Steel Bank Common Lisp environment (v1.0.51).

Plan exploration

Executing the HTN with both the three- and five-axis planning problems will result in the generation of all the complete potential plans. The HTN was executed initially to return the first complete plan. Next the HTN was executed in five seconds increments up to sixty seconds. SHOP2 returns information for each execution regarding the number of complete plans found, and the minimum and maximum cost.

As seen in Figure 7, it is noticeable that the number of complete plans generated for the three-axis machine is more than twice that of the five-axis machine. This highlights the higher computational effort for more complex problems. Figure 8 also shows the efficiency increase in terms of the time saved when comparing the first identified plan with the plan of the lowest cost discovered within the specified timeframe. For the tests that are executing in 5 second intervals, the plan with the lowest cost stabilise at 200 minutes for a three-axis machine after exploring 574 plans, and 18 minutes for a five-axis machine in just 50 plans. This shows that with no optimisation, the lowest cost plan from the 60 second period was discovered in 15 seconds, and 10 seconds for the five-axis machine.

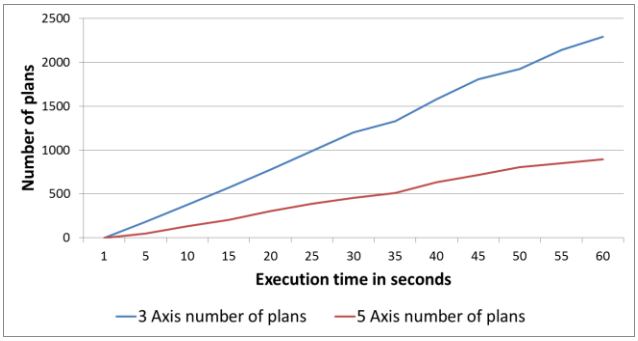


Figure 7 - Plan exploration

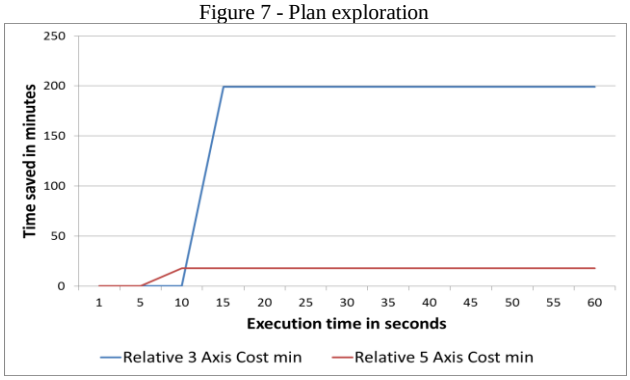


Figure 8 - Efficiency

Plan optimisation

Next, the same experiment was performed with the addition of the branch-and-bound optimisation. This is done by specifying the `:optimize-cost` flag in the problem definition. It is evident from Figure 9 that the number of complete plans generated in the allocated time frame is much lower with the use of the branch-and-bound algorithm.

It is also noticeable in Figure 9 that the number of optimised plans for the three-axis machine rises quickly, peaking at 22 before rapidly dropping to 6 where it stabilises. For the five-axis machine, the number of plans fluctuates between a maximum of 6 and a minimum of 2. This behavior is because the branch-and-bound optimisation is continuously trying to identify partial plans of a lower cost. Once a lower cost partial plan is identified, the algorithm will then explore it to find a complete plan that is of an overall lower cost than the previous plan. Figure 10 shows the increase in efficiency for the discovered plans. It is evident that the time saved for both the three- and five-axis machines increases gradually within the first 10 seconds. The time saved then stabilises for both the problems until 25 seconds for the three axis machine, where it reaches an efficiency saving of 19 minutes. The five axis problem increases rapidly until it stabilises with an efficiency gain of 74 minutes in 50 seconds of execution time.

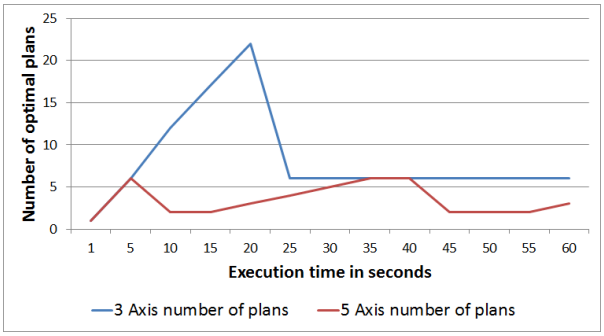


Figure 9 - Plan exploration (optimised)

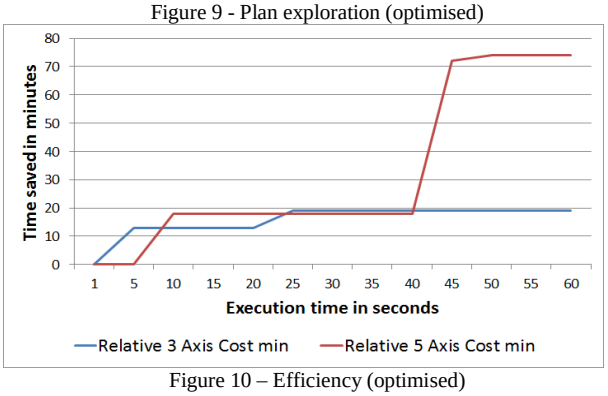


Figure 10 – Efficiency (optimised)

Plan comparison

In comparison, the number of plans generated when using the branch-and-bound optimisation algorithm is significantly lower. However, the number of explored plans is insignificant providing that the identified plans are the most efficient.

It is evident from Table 1 that the first identified plan for the three-axis machine when using the branch-and-bound algorithm has a lower cost by 186 minutes. The initial cost for a five-axis machine has the same cost for both tests. As seen in Table 2 the difference between the identified lowest cost plans in the whole sixty second period is 6 minutes for a three-axis machine, and 56 for a five-axis machine. This shows that the branch-and-bound algorithm can identify plans of a lower cost within the sixty second period even if the efficiency gain is only small.

Plan	First plan cost	First optimised plan cost	Difference in minutes
3-axis	2745	2559	186
5-axis	4777	4777	0

Table 1 - Comparison of the first identified plan

Plan	Lowest cost plan	Lowest optimised cost plan	Difference in minutes
3-axis	2546	2540	6
5-axis	4759	4703	56

Table 2 - Comparison of the identified lowest cost plan

Table 3 shows the execution time taken to identify the plan with the lowest cost with and without the use of the branch-and-bound optimisation. It is noticeable that the plans of a lower cost are discovered in the last third of the allocated time frame, and in the first quarter without the optimisation. Even though the time taken to find the optimal is 35 seconds longer for both problems when using the branch-and-bound optimisation, the overall efficiency gained makes its use beneficial. It is also evident that the cost reduction for the five-axis problem when using the branch-and-bound optimisation is higher than the three-axis problem. This potentially indicates that the efficiency of the optimisation algorithm increases as the problem's complexity also increases.

Problem	Not optimised time	Optimised time
3 axis	15	50
5 axis	10	45

Table 3 –Comparison of the execution time to find the lowest cost plan

Plan justification

The optimisation of the produced plan should result in an ordered set of tasks that exhibit the same or better time-saving decisions that a domain expert would make when creating a calibration plan. To examine whether this is true, an extract from the optimised plan for the three-axis machine problem was examined to highlight the justification behind the time-saving decisions that were made. This will validate the ability to encode expert knowledge and decision making skills in an HTN.

Comparing the first identified plan against the optimised plan will highlight the different ordering of tasks which results in the optimisation. From this difference in ordering, it is possible to derive the reasoning that resulted in the HTN making these decisions. Table 4 shows an extract taken from the optimised calibration plan for a three-axis machine and provides justification for the ordering.

Test	Instrumentation use	Justification
X Positioning	Laser interferometer	The positioning error of the X axis in this case has a high importance
X Straightness in Z	Laser interferometer	The laser is already aligned parallel to the X axis. From this setup the optics can be changed, therefore, saving time.
X Straightness in Y	Laser interferometer	
X about Y (Yaw)	Laser interferometer	
X about X (Roll)	Electronic level	Instrumentation is quicker to use than the laser
X about Z (Pitch)	Electronic level	The electronic level is already setup.

Table 4 – Extract from the optimised HTN plan with justification for the provided selection.

After examining the ordering of tests and their justification, it is evident that both the knowledge and decision making skills of a domain expert can be encoded and automated using a HTN. The justifications provided in Table 4 state the logical reasoning behind the test's ordering. This includes reasoning about which is the most efficient instrumentation to be used based on the previous and next error component.

Plan validation

The validity of the ordering can be verified by observing that the same decisions have been made in a traditional handmade plan for the same three-axis machine problem.
An extract from a calibration plan for a three-axis machine tool that was produced by a domain expert is shown in Table 5. When comparing this extract with the automated plan extract shown in Table 4 it is noticeable that the domain expert has made similar decisions regarding the sequencing of tests based on the premise of trying to minimise instrumentation setup and adjustment time.

Even though comparing an extract from an expert's calibration plan against the one generated by the HTN only provides for a very simplistic initial validation procedure, it does provide enough validation to warrant the continuation of this project by highlighting that calibration planning can be successfully automated without the loss of expert knowledge.

Test	Instrumentation used	Justification
X axis positional accuracy	Laser interferometer	
X axis accuracy and repeatability	Laser interferometer	The laser is already aligned parallel to the x-axis
X about X (Roll)	Electronic level	Depending on whether the optics will be realigned for the vertical Z, or removed for the Y. If removed for the Y, then remove the laser and setup the electronic level.
Z axis positional accuracy	Laser interferometer	
Z axis accuracy and repeatability	Laser interferometer	The laser is already aligned parallel to the z-axis
Squareness of X axis to Z axis	Ballbar or granite square	If the machine was also being tested for dynamic errors, then using the ballbar would be beneficial and save time later on.

Table 5 - Extract from a handmade plan with justification for the provided selection.

Conclusion

The work undertaken in this paper has shown how the process of machine tool calibration can be broken down by task decomposition to create a suitable HTN. The developed HTN was written in the common LISP format for execution in the SHOP2 architecture. Making use of a well-tested HTN architecture like SHOP2 means that no effort is wasted in implementing the HTN algorithm itself. It was then shown how the SHOP2 architecture can be used to execute problems against the created HTN. Two basic problem definitions of different complexities were created for testing the HTN's performance. The first was for a three-axis machine tool and the second a five-axis machine tool.

Each problem was tested by executing the HTN for durations in five second increments up to sixty seconds. This allowed for the evaluation of the quantity of complete plans found, and the minimum and maximum potential cost. In addition, this cycle was also performed using the branch-and-bound algorithm as a search optimisation strategy. After analysing the results it was evident that the use of the branch-and-bound algorithm improved the performance for both the three- and five-axis planning problems. It is suggested that the cost reduction provided by the branch-and-bound optimisation increases as the problems complexity also increases.

The results show that an HTN is viable solution for machine tool calibration process planning. The next stage is to develop the HTN further to include a more comprehensive representation of machine tool calibration. This would include better consideration of the instrumentation setup based on the machine's characteristics. Currently the HTN only looks for the solution with the lowest cost in terms of time and does not provide a solution for deciding which measurement techniques should be used. The HTN will require expanding to include and process a higher quantity of knowledge to remove this problem, and improve efficiency in terms of measurement traceability, repeatability and uncertainty. Verification of the complete HTN will not only include the evaluation of its efficiency, but the comparison the proposed plan against the plan of a subject expert. This way we can establish confidence in the HTN's planning power.

References

BOHEZ, E. L. J., ARIYAJUNYA, B., SINLAPEECHEEWA, C., SHEIN, T. M. M., LAP, D. T. & BELFORTE, G. 2007. Systematic geometric rigid body error identification of 5-axis milling machines. *Computer-Aided Design,* 39**,** 229 - 244.

BRINGMANN, B., BESUCHET, J. P. & ROHR, L. 2008. Systematic evaluation of calibration methods. *CIRP Annals - Manufacturing Technology,* 57**,** 529-532.

DEÁK, F., KOVÁCS, A., VÁNCZA, J. & DOBROWIECKI, T. P. 2001. Hierarchical Knowledge-Based Process Planning in Manufacturing. *Proceedings of the IFIP TC5 / WG5.2 & WG5.3 Eleventh International PROLAMAT Conference on Digital Enterprise - New Challenges: Life-Cycle Approach to Management and Production* Kluwer, B.V.

DELCAM. 2011. Knowledge-Based CAD/CAM: Separating the Hype from True Machining Intelligence. Available: http://featurecam.com/general/events/techart/knowbased.asp [Accessed 2/10/2011].

GHALLAB, M., NAU, D. & TRACERSO, P. 2004. *Automated Planning Theory and Practice* Morgan Kaufmann Publishers.

GOLDMAN, R. P. 2011. Documentation for SHOP2. University of Maryland.

MEKID, S. 2009. *Introduction to precision machine design and error assessment*, CRC Press.

NAU, D., AU, T., LLGHAMI, O., KUTER, U., MURDOCK, J., WU, D. & YAMAN, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20**,** 379-404.

PARKINSON, S., LONGSTAFF, A. P., CRAMPTON, A., ALLEN, G., FLETCHER, S. & MYERS, A. 2011. Representing the Process of Machine Tool Calibration in First-order Logic. *17th International Conference on Automation and Computing (ICAC'11).* Huddersfield.

RAMESH, R., MANNAN, M. A. & POO, A. N. 2000. Error compensation in machine tools -- a review: Part I: geometric, cutting-force induced and fixture-dependent errors. *International Journal of Machine Tools and Manufacture,* 40**,** 1235 - 1256.

SATYANDRA, K. G. 1998. IMACS: A Case Study in Real-World Planning. *In:* DANA, S. N. & WILLIAM, C. R. (eds.)

Performing a lifted Reachability Analysis as a first step towards lifted Partial Ordered Planning

Bram Ridder & Maria Fox
King's College London
Department of Informatics

Abstract

In this paper we describe a new algorithm to do lifted reachability analysis on a domain and produce the same results compared to performing reachability analysis on a grounded Relaxed Planning Graph. Most reachability algorithms and heuristics used in planners today require the problem domain to be fully grounded. We prove that, by extending the domain analysis done by TIM (Fox and Long 1998), this requirement is not necessary and sometimes even undesirable. By not grounding we can treat objects equivalently, which significantly increases the performance. We show some experimental results which - although not competitive yet - support the correctness of the algorithm. This work is a first step towards a lifted Partial Ordered Planner.

Introduction

Most planners nowadays ground planning problems before solving them, most heuristics used also have the prerequisite that the domain is fully grounded. There are however cases where grounding is not desirable, especially with larger problems where grounding takes a long time or can even make the planner run out of memory. Attemps have been made in the past to either reduce or forego grounding all together, especially in Partial-Order planners like VHPOP (Younes and Simmons 2003). None of these planners are able to compete with planners which ground the entire domain like Fast-Downward (Helmert 2006), LAMA (Richter and Westphal 2009), and FF (Hoffmann 2001).

The goal of this paper is to introduce a new reachability algorithm which does not require the entire domain to be grounded but yields the same results compared to doing the same analysis on a grounded domain with a Relaxed Planning Graph. The work presented in this paper is a precursor to fully lifted partial ordered planning.

The structure of the paper is as follows, we begin with defining a planning problem and summarise the analysis performed by TIM (Fox and Long 1998). Next, based on the transition rules, types and invariants extracted by TIM we construct *Lifted Transitions*. During the construction of these transitions we decide which variables need to be grounded. A subset of the preconditions of every *Lifted Transition* will be combined into a *node*. Together they will be used to create a *Lifted Transition Graph*. This graph is reminiscent of the Domain Transition Graphs created by Fast-Downward (Helmert 2006) but with some notable differences.

In the next section we introduce an equivalence relationship between domain objects, show how they are constructed and show that they can be used interchangeably. This forms the basis of the lifted reachability algorithm and allows us to prove sets of properties to be reachable for a set of objects at the same time - as opposed to proving properties for a single object at a time. Using a naive reachability algorithm we will prove that the lifted algorithm produces the same results as doing grounded RPG reachability analysis.

After the algorithm is described and proven to be correct we will show some experimental results to support this clain and present the direction this research will be going in future work.

Translation

We begin by briefly summarising the static domain analysis performed by the TIM system. This analysis extracts the underlying types and invariants of the domain, this allows us to recognise where grounding is necessary.

TIM analysis

The following definitions are from the TIM paper (Fox and Long 1998) but are briefly repeated here for reference.

Definition 01 *A typed planning task is a tuple* $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ *where:*

- T *is a set of types. Every type* $t \in T$ *has a set of supertypes,written* $SuperType(t)$.
- O *is a set of objects, each object* $o \in O$ *is associated with a type* $t \in T$*, written* $Type(o)$.
- P *is a set of predicates, where a predicate* $p \in P$ *is a tuple* $\langle name, types \rangle$*. A variable* v *is a pair* $\langle t, D_v \rangle$*, where* $t \in T$ *and* D_v *is the domain which is a set of objects. The set is initialized by:* $o \subseteq O | Type(o) \in SuperTypes(t) \cup \{t\}$*. An atom is a tuple* $\langle p, V \rangle$*, where* $p \in P$ *and* V *is a set of variables. We refer to the* i*th variable with the notation* V_i*. A grounded atom is an atom where the size of every domain* D_v *of every variable* $v \in variables$ *has a size of one.*

- *A is a set of operators, where an operator $a \in A$ is a tuple $\langle name, parameters, precs, effects \rangle$. $parameters$ is a set of variables. $precs$ and $effects$ are atoms, which are pairs $\langle p, v \rangle$, where $p \in P$ and $v \subset variables$.*
- *s_0 is set of grounded atoms called the initial state.*
- *s_g is set of grounded atoms called the goal.*

TIM builds a transition rule for each variable of each operator.

Definition 02 *A property is a predicate subscripted by a number between 1 and the arity of that predicate. Every predicate of arity n defines n properties.*

Definition 03 *A transition rule is an expression of the form: $property \Rightarrow property \rightarrow property$ in which the three components are bags of zero or more properties called enablers, start and finish, respectively.*

These rules are built up by TIM by constructing a triplet of bags of properties. This structure is termed a *Property Relating Structure (PRS)*. For each operator a PRS is constructed with respect to each parameter. The first bag of properties, called p_prec, is formed from the preconditions of the operators, and the number used to form the property is the argument position of the parameter being considered. For example, if the precondition is *at(X, Y)*, and the parameter being considered is Y, the property formed is at_2. The second bag, called $p\_deleted\_precs$, contains all the properties which appear as delete effects in the operator with respect to the same parameter. The third bag, called $p\_add\_elements$, contains all the add effects with respect to the same parameter.

From this structure the transition rules are constructed according to the following formula:

$p\_precs \ominus p\_deleted\_precs \Rightarrow p\_deleted\_precs \rightarrow p\_add\_elements$

Where $\ominus$ signifies the bag difference. If any of these rules contains a property in both the $p\_deleted\_precs$ and $p\_add\_elements$ multiple exchanges of that property is said to be *exchanged*. If multiple properties are exchanged in the same transition rule it is split up such that only a single property is exchanged per rule. Rules which have an empty start of finish bag are *attribute transition rules*.

Based on these transition rules properties are separated into equivalence classes from which the property and attribute spaces are constructed. This is done by forming collections of properties for every transition rule by combining all the properties in the *start* and *finish* bags. If a property appears in the start or finish of two rules then a single collection will be formed from the two rules. These collections are used to associate transition rules with them, every transtition rule which contains any of the properties in the collection will be added. Those collections which have a *attribute transition rule* associated with them are attribute spaces and those who have not are property spaces.

TIM then analyses the initial state in order to assign the domain objects to their appropriate spaces. This analysis also identifies the initial properties of individual objects and uses them to form states of the objects in the property spaces. The initial states in a property space are then extended by the application of the transition rules in that space to form complete sets of states accounting for all of the states that objects in that property space can possibly inhabit. States in this context much resemble *Domain Transition Graph Nodes* (Helmert 2006), they denote the set of facts which can be true at any given time and their relation with other objects and the possible transitions which exchange one property for another.

Definition 04 *Properties which are part of a property state are* balanced

Definition 05 *On the other hand, properties which are part of an attribute space do not have this property and either decrease or increase in number and are* unbalanced.

After doing the TIM analysis we will now extend the structures generated by TIM, as will be described in the next section. These extended structures will form the bases of the lifted reachability analysis as is described in the next section. In addition to this it allows us to detect which objects are equivalent, that is objects which can be used interchangably during the reachability analysis.

Constructing the Lifted Transition Graph

Given the sets of property and attribute spaces and the associated transition rules we evalulate all transitions rules and create a *Lifted Transition* for each of them.

Definition 06 *Given a transition rule from property state ps_{from} to property state ps_{to} a* Lifted Transition *$\langle from, to, operator, free \rangle$ is constructed where:*

- *$from/to$ are nodes which contain a set of pairings of atoms and properties. For every property $p \in ps_{from} \cup ps_{to}$ an atom is created whose predicate is that of p. The set of properties is equal to p.*
- *$operator$ is a copy of the operator of the transition rule.*
- *$free$ is a set of pairings of atoms and properties. The set of atoms is equal to all the preconditions which are not part of $from$.*

The properties paired with these atoms are formed based on the *exchanged* property of the transition rule - remember there can be only a single such property per transition rule due to the way transition rules are constructed by TIM. For every atom which has a variable which is the same as the parameter linked by the said property a property is added to that atom based on the predicate of the atom and index of the variable, i.e. the variable is *balanced*.

The variable domains of the atoms in $from$ and to are bound to the variable domains of the corresponding preconditions and effects of the $operator$, respectively. After constructing the transitions we end up with a set of paired sets of atoms and transitions between them. In order to do lifted reachability analysis some requirements will have to be met, in particular once a set of $free$ atoms is found it must be applicable to any set of atoms found for $from$. To illustrate this, consider the driverlog domain where a package is at a particular location. The transition rule constructed so far looks as follows:

- $from = \{(at\ package\ location)\}$.

- $to = \{(in\ package\ truck)\}$
- $operator = load$
- $free = \{(at\ truck\ location)\}$

In this case we can unify any truck at any location with the $free$ set and the same for package in the $from$. Which results in the package being loaded into any truck at any location, this is clearly undesirable. To remidy the situation we need to specify that the location of the package and the truck must be equal and reflect this in the resulting to as well. The updated transition rule looks as follows, variables which are underlined are grounded:

- $from = \{(at\ package\ \underline{location})\}$.
- $to = \{(in\ package\ truck), (at\ truck\ \underline{location})\}$
- $operator = load$
- $free = \{(at\ truck\ \underline{location})\}$

Now this transition rule tells us that if any package at a specific location can be loaded into a truck at that location, all packages at that location can. This results in that package being in a truck at that specific location.

Effectively we are splitting the set of preconditions of every *Lifted Transition* into two disjunctive sets. We must guarantee that every possible variable domain assignment is consistent with all other assignments. We will now explain how this is achieved

Merging and Grounding The naive way of making sure the requirement above is satisfied is by adding all preconditions of an operator to the $from$ set or grounding all terms. In the first case all the preconditions are part of the same set and in the second case the variable domain of every atom is of size one. However, we can do much better. Consider the following operator:

Example 01 *LiftObject* - $object, hoist, location$
Preconditions: $(at\ object\ location) \wedge (available\ hoist)$
Effects: $(lifting\ hoist\ object) \wedge \neg(available\ hoist) \wedge \neg(at\ object\ location)$

If this operator is part of a transition rule where the property at_1 is exchanged for $lifting_2$, then we can see that the precondition related to the hoist is independent of the object being lifted. In this case we do not need to add $(available\ hoist)$ to the $from$ set because all objects which satisfy the preconditions related to object can be lifted by any available hoist.

If we alter the operator slightly and demand that the location of the hoist matches that of the object:

Example 02 *LiftObject* - $object, hoist, location$
Preconditions: $(at\ object\ location) \wedge (available\ hoist) \wedge (at\ hoist\ location)$
Effects: $(lifting\ hoist\ object) \wedge \neg(available\ hoist) \wedge \neg(at\ object\ location)$

We cannot leave location lifted as we have done in the previous example, because in this case the location actually matters and determines if we can pickup the object or not. By grounding the term $location$, if this case we conclude that any object at a specific location can be lifted if there is an available hoist at that location and this holds for all objects at that location.

This greatly reduces the number of actions to consider during reachability and allows us to do reachability on domains where the traditional method fails due to grounding due to memory constraints.

For every *Lifted Transition* $\langle from, to, operator, free\rangle$ the following rules are enforced for every paired precondition $\langle\langle p, v\rangle, \{properties\}\rangle \in free$:

- If there is a variable $v' \in v$ and a property $p \in properties$ whose index matches that of v' (i.e. the variable is *balanced*) and the variable is shared with any atom $a \in from$ then $\langle p, v\rangle$ is added to $from$ and removed from $free$. If $\langle p, v\rangle$ is not removed by the effect of the operator it is added to to also.
- If no such property is found and v' is *unbalanced* and the parameter is shared with an *unbalanced* variable of any atom $a \in from$ then v' needs to be grounded. Note that because variables are linked to the parameters of the operator all variables that are made equal to that parameter will be grounded.
- Likewise, variables which appear in both $from$ and to but are not part of a property which is *balanced* and are not persistent also need to be grounded.

The first rule concerns itself with preconditions that contain a variable which is *balanced*. These preconditions must be included in the *Lifted Transition* because if it is not we could end up with two sets of bindings which unify a parameter with different domains. This would violate the first property.

If a variable is shared which is *unbalanced* it is flagged to be grounded. After the $from$ and to sets are finalised all variables marked for grounding will be grounded. Grounding terms does not break any of the properties. By grounding we guarantee consistency since the domain of the variable is reduced to a single possible value.

The latter case handles *unbalanced* variables which appear in the precondition and are present in an effect which is relevant to the final outcome. To guarantee that these variables have the same domains in the $from$ and to nodes we ground them.

Theorem 01 *In every* Lifted Transition $\langle from, to, operator, free\rangle$ *where the rules are enforced, every possible assignment of the variables of* $from$ *and* $free$ *are consistent. We will prove this theorem by showing that the negation leads to a contradiction.*

Given two variables $v_{from} \in from$ *and* $v_{free} \in free$ *which refer to to the same parameter, we can assign both variable domains different values. We will consider all possible situations:*

- *Both variables are lifted: If* v_{from} *is lifted it means that it is* balanced*, otherwise it would have been grounded when processing* v_{free}*. Given that* v_{free} *is lifted means it is* balanced *and should have been added to* $from$*, thus this situation is impossible.*

- v_{from} *is lifted and* v_{free} *is grounded:* v_{free} *can only be grounded if it refers to an* unbalanced *variable since* v_{from} *is lifted it is* balanced *thus this situation is impossible.*
- v_{from} *is grounded and* v_{free} *is lifted: when* v_{from} *is grounded then all variables referring to the same parameter are grounded so this situation is impossible.*
- *Both variables are grounded: Whenever* v_{from} *is grounded all variables which refer to the same parameters are grounded too so the only possible assignment to both variables are the same.*

From this it follows that the set of variables shared between $from$ *and* $free$ *are those which are grounded.*

Lifted Transition for Attribute Spaces At this point we have only defined *Lifted Transitions* for property spaces. We create the same structures for attribute spaces but the way in which they are constructed is different. We are only interested in attribute spaces which add a fact, this is because we assume that no fact is ever deleted, so those attribute spaces which remove atoms are not of interest to us.

Unlike property spaces there is no single property being exchanged for another so it is not clear how to determine which variables are *balanced*. Therefore for every transition rule from an attribute space which has not yet been created for a property space - it is possible that an operator is part of multiple transitions rules - we create a *Lifted Transition*$\langle from, to, operator, free\rangle$ where:

- $from$ contains all preconditions of the operator which contains any of the variables of either the atom added by the attribute space or any of the atoms in $from$.
- to contains the atom added by the attribute space.
- $operator$ is the operator.
- $free$ contains all the preconditions which were not added to $from$.

Note that none of the variables are grounded.

Merging After creating all the lifted transitions we proceed by merging those nodes which are equivalent. This process - as we shall see - will create structures which will be reminiscent to those familiar with DTGs but with some notable differences. First and foremost, where DTGs ground all the atoms in the nodes, we try to limit grounding as much as possible. Secondly the graph generated for each object by Fast-Downward are disconnected, while we allow nodes within the same connected graph to belong to different objects. Apart from reducing the number of nodes in the final graph it also helps us during the reachability analysis. Consider for example the following node: $(at\ truck\ location) \wedge (driving\ driver\ truck)$, this node is part of every DTG of the types truck and driver. In the graph we generate only a single node per location - since it is grounded - and we do not care if it is reached by the *Lifted Transition* from $(at\ driver\ location)$ or the *Lifted Transition* from $(at\ truck\ location) \wedge (empty\ truck)$ they both yield the same result.

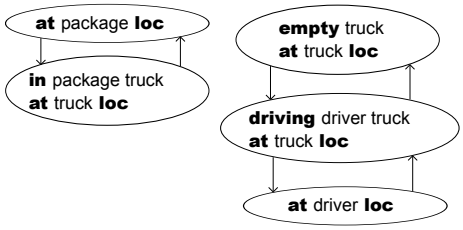


Figure 1: Driverlog Example - the bold terms (loc) are grounded

Definition 07 *Two nodes are equivalent if there exists a bijection of both sets of atoms where the predicate and variable domains of each mapped pair are the same. In addition, if two variables of any pair of atoms are equal in one node then the variables corresponding to the mapped atoms must also be equal. The same is true for any two variables which are not the same.*

The latter requirement is necessary for some domains where - based on the predicate and variable domains alone - multiple bijections are possible. One such example is the Blocksworld domain where some nodes contain the pair $(on\ block\ block) \wedge (on\ block\ block)$ where the relationships between the variables become important to create the correct bijection.

For every found bijection we group together all the nodes and select one of these nodes to be representative for all equivalent nodes. For every atom of the representative node the set of properties is united with the set of all properties of the atoms it is mapped to. After all this is done the Lifted Transitions are migrated to the representative nodes. The parameters of all the operators are updated to point to the variables of the representative nodes.

In figure 1 we see the example of the *Lifted Transition Graph* for the driverlog domain. Note that the variables of the atoms refer to the types of objects they represent, the variables of the type location are grounded. Also note that - as discussed before - the node $(driving\ driver\ truck) \wedge (at\ truck\ loc)$ is connected by both the node referring to the type driver and the node referring to the type truck.

Reachability

Having discussed how the *Lifted Transition graph*(LTG) is constructed we will now detail the reachability analysis algorithm. First of all we describe how the initial state is used to initialise the *Lifted Transitions* and *Lifted Transition Graph* nodes.

Next, to prove the correctness of the proposed lifted algorithm we will introduce a naive approach which does not utilise the liftedness of the structure, but produces the same results as doing grounded reachability analysis.

To improve upon this implementation - and utilising the lifted structure - an equivalence relationship is introduced between objects which allows the algorithm to use them interchangebly. Finally we introduce the lifted reachability analysis and prove, based on the naive algorithm, that it produces the same result as the grounded RPG analysis.

Initial Setup

The starting point of the reachability analysis is the atoms in the initial state. These values are mapped to the nodes of the LTG and similary to the $free$ set of every *Lifted Transition*.

Definition 08 *A* Reachable Transition *is defined as the tuple* $\langle l, m \rangle$*, where:*

- l *is a* Lifted Transition*.*
- m *is a set of mappings from every variable of the atoms* l_{free} *to an* Equivalent Object Set Group *(defined below).*

Definition 09 *A* Reachable Node *is defined as the tuple* $\langle n, m \rangle$*, where:*

- n *is a node of the* Lifted Transition Graph*.*
- m *is a set of mappings from every variable of the atoms* m_{from} *to an* Equivalent Object Set Group *(defined below).*

Both structures are similar and are initialised in the same way.

- For every object $o \in O$ we search for all initial atoms whose variable domain contains o.
- We create a new *Reachable Transition* T or *Reachable Node* N if the found atom can be unified with T_{free} or N_{from}, respectively. The mappings of the variables are made accordingly.
- For every created structure we search for any atom in the initial state which can be unified with any of the atoms in the set which have not been assigned yet. For every such atom found the structure is copied and the assignment made.
- This process is repeated until no new assignments can be made, all duplicate structures are removed.

We end up with two sets of mappings of atoms in the initial state to the nodes and $free$ preconditions of the lifted transitions.

Simple Reachability Analysis We will now present a simple algorithm which takes the initilised *Lifted Transitions* as input and show that that it computes the same set of facts as using the grounded domain to build an RPG till the leveoff point.

1. Add all initial atoms to the reachable set s.
2. Record the size of s.
3. For every *Lifted Transition* $\langle from, to, operator \rangle$ find all consistent mappings from s to $from$ and $free$.
4. For every $operator$ apply every mapping of $from$ and $free$ and add all effects to s.
5. If the size of s hasn't chanced we are done. Otherwise go back to (2).

Theorem 02 *The proof that this yields the same result is quite trivial. Note that for every* Lifted Transition *the following holds:* $[from \cup free \equiv operator_{prec}]$ *and we are guaranteed that none of the variable domains are inconsistent. So the only reachable atoms are those which are part of the initial state or those which have been reached by operators because their preconditions were achieved.*

The presented reachability analysis only considers grounded atoms. Next we will establish an equivalent relationship between objects which allows us to consider lifted atoms and speed up the reachability analysis.

Equivalent Objects

To speed up the reachability analysis we create *Equivalent Object Set Groups*. For every object $o \in O$ an *Equivalent Object Set* and *Equivalent Object Set Group* are created. The created *Equivalent Object Set* is part of the *Equivalent Object Set Group*. These structures will help in establishing equivalence relationships between objects. Essential in establishing this relationship are the atoms containing o in the initial state and its relationship to other objects. In the simplest case two objects have no connection to other objects in the initial state and the atoms they are part of have the same predicate. In that case we can conclude that these objects are equivalent, if an atom containing one of these objects is reachable the same holds for the other object - they are interchangeable.

Most planning problems, however, will have more interesting relationships between objects. We will show that we do not need to consider all these relationships to establish equivalence relationships. To give an example, if there is an atom in the initial state which is not part of any precondition we can ignore it. This is similar to the work done on RIFO(Nebel, Dimopoulos, and Koehler 1997). A more interesting example can be found in the *driverlog* domain, if two trucks start at the same location and have no other relation with any other object we can conclude that they too are equivalent, this is closely related to the 'functional symmetry'(Fox and Long 1999). More interesting cases arise when multiple relationships between different objects are present, e.g. both trucks have a different driver and a multitude of packages loaded. In this section we will present the data structures used to establish the equivalence relationships.

Definition 010 *A* Equivalent Object Set *is a tuple* $\langle I, N, o, G \rangle$*, where:*

- I *is a mapping from every node of the LTG to the tuple* $\langle A, M \rangle$*, where*
  - A *is the set of atoms for which a (partial)mapping was found from the initial state.*
  - M *is a mapping from the initial state to every other variable in the node to a set of* Equivalent Object Sets*.*
- N *is initially the same as* I*, but when new facts are proven to be reachable they will be added to this structure.*
- O *is the object this* Equivalent Object Set *is created for.*
- G *is the* Equivalent Object Set Group *this* Equivalent Object Set *is part of and contains all the* Equivalent Object Sets *it is equivalent with.*

Definition 011 *An* Equivalent Object Set Group *is a tuple* $\langle E, f, parent \rangle$*, where:*

- E *is a set of* Equivalent Object Sets *which are equivalent.*
- f *is a* signature*. Every* Equivalent Object Set Group *can only contain objects of a specific type and some its super*

types. The signature is generated by checking all the variables of all atoms of the Lifted Transition Graph*'s nodes, all variable whose type is equal to or a super type of the type this* Equivalent Object Set Group *is created are combined as the signature. A* Equivalent Object Set Group *can only be concluded to be equivalent if the signatures match.*

- $parent$ *is a link to another* Equivalent Object Set Group *and is initialy* $NULL$*. When* Equivalent Object Set Groups *get merged a new* Equivalent Object Set Group *is created with the result of the merge and the parent values of the merged* Equivalent Object Set Groups *are updated to point to this group.*

We can conclude that two *Equivalent Object Set Groups* $eosg1$ and $eosg2$ are equivalent if there exists two *Equivalent Object Sets* $eos1 \in eosg1$ and $eos2 \in eosg2$ such that $eos1_N$ is a super set of $eos2_I$ and $eos2_N$ is a super set of $eos1_I$. In other words, if the both can reach the initial state of the other we conclude that the *Equivalent Object Set Groups* they are part of must be equivalent. Thus by proving that two members of both groups are equivalent we have proven that this relation holds for all members of both groups.

Establishing these relationships will create an overhead during the reachability analysis, but once these relationships have been established it can speed up reachability significantly, because given a set of n equivalent objects we only need to prove that a fact is reachable once, for all equivalent objects, instead of having to prove this relationship n times. One of the best illustrations of this can be found in the *gripper* domain where there are many balls in a single room. Since the ball objects have no other relation to other objects, except for the room they are in - which is grounded - all the balls in the same room are equivalent.

Objects which are part of nodes in the initial state which contain mappings to other lifted parameters are not as easy to establish equivalence relationships between because we need to prove that the same relationship between the set of objects is reachable. Going back to the driverlog domain, if we have two trucks at different locations, being driven by different drivers and both carrying a distinct set of packages we need to prove - on top of the fact that both trucks can reach the same location - that the drivers and packages can swap truck. On the other hand, the packages in each trucks are equivalent.

We will now present the proof that two equivalent objects can be used interchangebly.

Theorem 03 *Given two objects* $o1$ *and* $o2$ *that are equivalent, i.e. their* Equivalent Object Sets *are part of the same* Equivalent Object Set Group*, then for every atom* $\langle p, v\rangle$ *that is reachable and there is a variable* $x \in v$ *such that* $o1 \in D_x$*, then the atom* $\langle p, v'\rangle$ *is also reachable where:*

$$x' \in v' = \begin{cases} x & : o1 \notin D_x \\ x \setminus o1 \cup o2 & : otherwise \end{cases}$$

We will prove this by proving a contradiction in the inverse: an atom $a1$ *containing* $o1$ *is reachable but the same atom* $a2$ *constructed by the method above containing* $o2$ *is not reachable.*

This means there is a Lifted Transition *which has an effect which can reach* a*, but the preconditions which contains variable(s) united with* $o1$ *are not reachable for* $o2$ *and thus cannot make* $a2$ *reachable. Note that* $o1$ *must be present in the preconditions, because otherwise the variable would be free. Because* $o1$ *and* $o2$ *are of the same type and all relations to other objects are identical, it must mean that there is a property which is true for* $o1$ *and not for* $o2$*. If this property is made true by a transition than that transition has a precondition which can be made true for* $o1$ *but not for* $o2$*. This means there must be an initial atom which declares this property true for one and not the other. If this is the case however, than the two objects cannot be equivalent.*

The Lifted Algorithm

Having established the initial mapping from the initial state to the nodes of the *LTG* and created the *Equivalent Object Set Groups* we can now present the algorithm which performs the reachability analysis and makes use of the lifted structures. We will prove that the results produced by this algorithm are identical to that of the RPG reachability analysis without having to ground all the operators and by utilising the equivalence relationships between objects we only need to prove an atom to be true for a whole set of objects at the same time. A possible problem is that the overhead caused by establishing the equivalance relationships might outweigh the benefits. We will return to this topic when discussing the results.

The reachability analysis is as follows:

- For every *Reachable Node* where all atoms have been mapped and who has a *Reachable Transition* where all atoms have been mapped we propagate the results and generate the effects.
- The established atoms are mapped to the *Reachable Nodes* and *Reachable Transitions*, new ones are created as appropriate.
- All the *Equivalent Object Set Groups* are evalulated and new equivalence relationships are created where appropriate.
- These steps are repeated as long as new atoms are proven to be reachable.

Propagate This is the easiest bit of the reachability analysis. Previously we have seen how the set of preconditions of every transition rule has been split into two sets. Because these sets are consistent and the results are proven to be reachable by the RPG analysis, we can combine the mappings of any *Reachable Node* and any *Reachable Transition* linked to that node - provided that it is a complete mapping - which will map an *Equivalent Object Set* the parameters of the *Reachable Transition*'s operator. Note however that not every parameter may have an *Equivalent Object Set* mapped to it. This can happen when an operator has a parameter which is not part of any precondition. For example the to variable is not bound by any precondition:

Example 03 *Move -* $object, from, to$
Preconditions: $(at\ object\ from)$
Effects: $\neg(at\ object\ from) \wedge (at\ object\ to)$

In these situations we map all *Equivalent Object Sets* whose object is of the correct type.

After the mappings have been made we can generate the effects. We continue propagating until we cannot generate more effects.

Map results For all effects generated we need to update the *Reachable Nodes, Reachable Transitions* as well as the *Equivalent Objects Sets*.

Definition 012 *For every effect* $e \in effects$ *if it has been achieved before we ignore it. Otherwise, we try to map it to any* Reachable Nodes *and* Reachable Transitions *which have a partial mapping. If such a mapping is possible we create a copy and do the mapping.*

The created mappings above will be added to the relevant Equivalent Objects Sets*.*

Establish new object equivalence relationships After the mappings have been updated we evalulate each *Equivalent Objects Set Group* and check if we can generate new equivalence relationships. For every pair of *Equivalent Objects Set Groups* $eogs1$ and $eogs2$ which are referring to an object of the same type and are not yet equivalent we check if two emphEquivalent Objects Set of both groups have the property $eos1_N \subset eos2_I \wedge eos2_N \subset eos1_I$. If this is true we conclude that both *Equivalent Objects Set Groups* are equivalent and can be merged.

When merging two *Equivalent Object Set Groups* we create a new *Equivalent Object Set Group* whose set of *Equivalent Object Sets* is equal to the union of both groups; the signature is the same. The parent of the merged *Equivalent Object Set Groups* are updated to point to the merged group.

Theorem 04 *The set of facts proven to be reachable by the above reachability analysis algorithm is equal to the set of facts proven to be reachable by the RPG. The proof is equal to that of theorem 02, with the exception of the equivalence relationships. Given that we have proven that this relationship does not break reachability the same proof holds.*

Results

We have implemented the algorithm described in the previous section to confirm that lifted reachability finds the same reachable goals as grounded reachability. Our code is not optimised, so we are not yet able to show a performance that is competitive with the RPG algorithms in use in state-of-the-art planners. However, we are able to confirm that the lifted approach works as expected. In all the problem instances considered, this algorithm produces the same set of atoms as the grounded RPG analysis.

To get a better idea of where we expect to do better than grounded reachability we compare the following features: 1) The number of paired action / fact layers that are constructedbefore the graph levels off; 2) The minimum number of sets of objects that are equivalent. Note that this number is equal to the total number of objects in the planning problem for the grounded case; 3) The total number of transitions. The results are depicted in Table 1.

	Grounded RPG / Lifted Algorithm		
Problem instance	#Layers	#Objects	#Transitions
Satellite01	4/2	13/7	79/6
Satellite10	4/2	39/6	7165/6
Satellite20	4/2	70/6	43290/6
Driverlog01	4/4	12/9	230/48
Driverlog10	4/3	27/19	2520/174
Driverlog20	5/4	99/63	218330/744
Zeno01	6/3	14/13	3687/59
Zeno10	5/3	24/15	30375/63
Zeno20	7/3	60/32	959530/97
Gripper01	3/3	306/6	2404/8
Gripper10	3/3	531/6	4204/8
Gripper20	3/3	781/6	6204/8
Rovers01	5/4	14/13	281/46
Rovers10	4/4	30/24	8220/223
Rovers20	5/5	61/58	423064/1351

Table 1: Grounded versus Lifted Reachability

As is clear from the table the number of iterations / objects and transitions are always equal or less for the lifted case. This is because in the worse case scenario we fall back to a completely lifted structure.

The ability to treat sets of objects equivalently is the main reason why we expect our approach to do better, so we think that the number of objects compared to the sets of objects we are able to construct is a key indicator to identify domains where our solution will do better. So based on the results in Table 1 we expect very good results for Satellite and Gripper. In the case of Zeno we are able to group together more objects as the problem instances becomes bigger so we expect this will be reflected by the future results. In the case of Driverlog we can only group together about a third of the objects for all problem instances, future results will have to show if that is enough to offset the overhead. We do not expect our approach to do very well on the Rovers domain because we can hardly group any objects together. This is due to the fact that the domain can equip every rover with a different set of instruments and allow each rover to traverse different parts of the waypoints. In this case we expect to pay the overhead but gain nothing in return. We hope that these hypothesis will be reflected in the future results.

Future Work

Having shown that doing lifted reachability is in theory possible we will first focus on speeding up the implementation to check if the speedup gained outweight the overhead incurred by this algorithm.

Landmarks

An obvious candidate to consider are landmarks, after the introduction of LAMA (Richter and Westphal 2009) a lot of work has been focussed on producing and utilising land-

marks in new ways (Richter, Helmert, and Westphal 2008) (Hoffmann, Porteous, and Sebastia 2004), especially after winning the IPC in 2008 (Richter and Westphal 2010) and 2011.

Three techniques are available for finding landmarks of which only one has been possible for lifted planning (Richter, Helmert, and Westphal 2008).

- Given all the set of actions which can achieve a certain atom, all the preconditions which are shared between the actions are landmarks for reaching the given atom. This method is easily adaptable to lifted planning.
- Given a DTG graph and a node matching an atom from the initial state and a node resembling an atom from the goal state, then set set of nodes which is part of every possible path between these two nodes are landmarks. The *Lifted Transition Graph* described in this paper has many features of the DTGs, so this technique can be adopted too.
- If an atom g is proved to be reachable under the delete relaxation constraint and we do the same relaxed reachability but prevent a certain atom r from being reached and g is no longer reachable we conclude that r is a landmark for g. In order to do this analysis we need way to do reachability analysis. With the algorithm discussed in this paper we could generate lifted landmarks.

Landmarks have been proven to be very powerful as part of the heuristic of grounded planners. We would like to test if the same is true if landmarks are introduced in lifted planners.

Recursive Structures

In some domains we have noticed a few interesting relations between objects which could be better exploited. Especially if there are relations between objects of the same type, like blocks in Blocksword or crates in Depots. For these relations between objects a recursive precondition would allows us to create equivalence relationships quicker.

For example, consider a large stack of blocks in blocksworld with the top block being clear. The precondition $(clear\ block)$ of the operator unstack can only be applied to the top block, making the block beneath it clear. Doing more analysis on this domain would, however, that any block can be made clear if the block above it is or can be made clear. So we can redefine the transition:

Example 04 *Unstack* - $block, block'$
Preconditions: $(clear\ block) \wedge (on\ block\ block') \wedge (handfree)$
Effects: $(holding\ block) \wedge (clear\ block') \wedge \neg(clear\ block) \wedge \neg(on\ block\ block') \wedge \neg(handfree)$

as

Example 05 *Unstack* - $block, block'$
Preconditions: $rec(block) \wedge (on\ block\ block') \wedge (handfree)$
Effects: $(holding\ block) \wedge (clear\ block') \wedge \neg(clear\ block) \wedge \neg(on\ block\ block') \wedge \neg(handfree)$

where $rec(block) = clear(block) \vee (\exists_{x \in Blocks}(on\ x\ block) \wedge (rec(x)))$.

This maps all blocks in that stack (except the top block and the bottom block) to the same node, once we prove that $(handfree)$ is true we can conclude that all blocks can be unstacked.

Conclusions

In this paper we have described a new algorithm for doing lifted reachability analysis and proved that its results are identical to the classic RPG approach. The presented results, although not yet competative, supports the correctness of the algorithm.

The *Lifted Transition Graph* can be used in a heuristic much like the Fast-Downward heuristic. The reachability analysis can be used for heuristic estimates as is done in FF. Because the reachability analysis algorithm presented in this paper is able to work with lifted atoms we expect it to be a better match for partial order planning. To see why consider the goal $(at\ package1\ s1)$, we choose the operator $drop\ package1\ truck\ s1$ and generate the following open conditions: $at\ truck\ s1$ and $in\ package\ truck$. The variable of type truck is lifted which we can achieve with a *Lifted Transition* without having to ground it.

We believe that the structures presented and the proposed future work offers a lot of scope for new algorithms and heuristics for planners working on lifted domains.

References

Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI'99*, 956–961.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)* 26:191–246.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)* 22:215–278.

Hoffmann, J. 2001. Ff: The fast-forward planning system. *AI magazine* 22:57–62.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proc. ECP-97*, 338–350.

Richter, S., and Westphal, M. 2009. The lama planner. In *ICAPS-06*.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In Fox, D., and Gomes, C. P., eds., *AAAI*, 975–982. AAAI Press.

Younes, H. L. S., and Simmons, R. G. 2003. Vhpop: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20:405–430.

Maintaining Partial Path Consistency in STNs under Event-Incremental Updates

Ot ten Thije and **Léon Planken** and **Mathijs de Weerdt**
Delft University of Technology

Abstract

Efficient management of temporal constraints is important for temporal planning. During plan development, many solvers employ a heuristic-driven backtracking approach, over the course of which they maintain a so-called Simple Temporal Network (STN) of events and constraints. This paper presents the *Vertex-IPPC* algorithm, which efficiently enforces partial path consistency when an STN is extended with an event and all its associated constraints. This algorithm uses some new results on directed path consistency on subgraphs. We prove that the worst-case time complexity of our algorithm is competitive with extant approaches. Our algorithm integrates well with a recently discovered vertex-incremental triangulation method, which, to the best of our knowledge, we are the first to have implemented. While experiments show that the new algorithm is outperformed on realistic networks, it is competitive on chordal graphs.

1 Introduction

Quantitative temporal constraints are essential for many real-life planning and scheduling domains (Smith et al., 2000). The *Simple Temporal Network* (STN; Dechter et al., 1991) is a popular model for temporal events and simple temporal constraints among them. The central role of STNs in deployed planning systems (Bresina et al., 2005; Castillo et al., 2006; Labone and Ghallab, 1995) makes efficient inference with STNs especially important.

During plan development, new events and temporal constraints are added and existing constraints may be tightened, and the consistency of the whole temporal network is frequently checked. Recent work has shown that enforcing *partial path consistency* provides an efficient means of propagating temporal information for an STN (Planken, 2008; Xu and Choueiry, 2003), answering queries such as:

- Is the information represented by the STN consistent?
- How can the events be scheduled to meet all constraints?
- What are possible times at which event x_i could occur?
- What relation between two events x_i and x_j is implied?

Dechter et al. (1991) identified these types of queries as the most relevant when solving STNs.

This paper introduces a new algorithm dubbed *Vertex-IPPC*, which maintains partial path consistency when a set of constraints associated with a new *event* is added to an STN. Vertex-IPPC is complementary to the *IPPC* algorithm designed by Planken et al. (2010), which maintains partial path consistency when a new *constraint* is added. To emphasise the difference between the two approaches, we will refer to the original IPPC as *Edge*-IPPC from here onward.

We first give the necessary definitions and discuss related techniques. Then we show that the concept of directed path consistency also applies to subgraphs, and how this can be used in our algorithm. We present proofs of correctness and upper bounds on run time. We then empirically validate the derived bounds of Vertex-IPPC and compare it with a straightforward extension of Edge-IPPC. Ideas for future work conclude the paper.

2 Preliminaries

A Simple Temporal Problem instance consists of a set $X = \{x_1, \ldots, x_n\}$ of time-point variables representing events, and a set C of m constraints over pairs of time points, bounding the temporal difference between events. Every constraint $c_{i\to j}$ has a weight $w_{i\to j} \in \mathbb{R} \cup \{\infty\}$ corresponding to an upper bound on the difference, and thus represents an inequality $x_j - x_i \leq w_{i\to j}$. Two constraints $c_{i\to j}$ and $c_{j\to i}$ can be combined into a single constraint $c_{i\leftrightarrow j} : -w_{j\to i} \leq x_j - x_i \leq w_{i\to j}$ or, equivalently, $x_j - x_i \in [-w_{j\to i}, w_{i\to j}]$, giving both upper and lower bounds. An unspecified constraint is equivalent to a constraint with an infinite weight; therefore, if $c_{i\to j}$ exists and $c_{j\to i}$ does not, we have $c_{i\leftrightarrow j} : x_j - x_i \in (-\infty, w_{i\to j}]$.

Instances of this problem represented as an undirected graph are called Simple Temporal Networks (STNs). In an STN $\mathcal{S} = \langle V, E\rangle$, each variable x_i is represented by a vertex $v_i \in V$, and each constraint $c_{i\leftrightarrow j}$ is represented by an edge $\{i, j\} \in E$ between vertex v_i and vertex v_j with two associated weights, viz. $w_{i\to j}$ and $w_{j\to i}$. *Solving* an STN $\mathcal{S}$ is often equated with determining an equivalent minimal network $\mathcal{M}$ (or finding in the process that $\mathcal{S}$ is inconsistent). Such a minimal network has a set of solutions (assignment of values to the variables that is consistent with all constraints) identical to that of the original STN $\mathcal{S}$; however, $\mathcal{M}$ has the property that any solution can be extracted from it in a backtrack-free manner. For STNs, the minimal network can be determined by enforcing *path consistency* (PC), which in turn coincides with calculating all-pairs shortest paths (APSP) on the constraint graph. Determin-

ing APSP for $\mathcal{S} = \langle V, E \rangle$, having n vertices and m edges, yields a complete graph; it takes worst-case time $\mathcal{O}(n^3)$ or $\mathcal{O}(n^2 \log n + nm)$ using Floyd–Warshall or Johnson's algorithm, respectively. Note that we always assume the (constraint) graph to be connected, so $m \geq n - 1$.

Instead of enforcing PC on an STN $\mathcal{S}$, one can opt to enforce *partial path consistency* (PPC; Bliek and Sam-Haroud, 1999). Although this yields a network $\mathcal{M}^*$ which is not minimal, it is nonetheless equivalent to $\mathcal{S}$ and a solution can still be extracted without backtracking. Moreover, $\mathcal{M}^*$ can be used to answer the queries listed in the introduction for those pairs of time points in which one is interested. Furthermore, while $\mathcal{M}$ is represented by a complete graph (having $\Theta\left(n^2\right)$ edges), $\mathcal{M}^*$ is represented by a *chordal graph* (sometimes also called triangulated), requiring that every cycle of length four or more has an edge joining two vertices that are not adjacent in the cycle. We denote the number of edges in such a chordal graph by m_c. Since $m_c = \mathcal{O}(nw_d)$, where w_d is the *graph width* induced by an ordering d of the vertices in V, $\mathcal{M}^*$ is potentially much sparser than $\mathcal{M}$. As in $\mathcal{M}$, all edges $\{i, j\}$ in $\mathcal{M}^*$ are labelled by the lengths $w_{i\to j}$ and $w_{j\to i}$ of the shortest paths from i to j and from j to i, respectively. In summary, an STN is partially path consistent if it is (i) chordal and (ii) every edge is labelled by the weights of the shortest paths between its endpoints.

Partial path consistency can be enforced by the P^{3}C algorithm in $\mathcal{O}(n(w_d)^2)$ time (Planken et al., 2008). Regarded as the current state of the art for solving an STN non-incrementally, P^{3}C builds on the concept introduced by Xu and Choueiry (2003). The minimum possible value of w_d is exactly the treewidth of the graph.

Determining treewidth is NP-hard in general (Arnborg et al., 1987). However, if the graph is already chordal, we can—in $\mathcal{O}(m)$ time, using *lexicographic breadth-first search* (Lex-BFS; West et al., 2001)—find a so-called *simplicial elimination ordering* d, a vertex ordering yielding minimum induced width w_d. Moreover, even a suboptimal choice of d usually results in a shorter run time than that of PC enforcing algorithms, as empirically demonstrated by Planken et al. (2008).

Simplicial elimination orderings also disclose a useful structural property of chordal graphs. In particular, every vertex v_k in a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$ of the graph G is *simplicial* in the graph G_k induced by the vertices $\{v_1, \ldots, v_k\}$, i.e. all vertices $N_k(v_k) = \{v_i \mid \{v_i, v_k\} \in E, i < k\}$ neighbouring on v_k in G_k induce a clique.

The P^{3}C algorithm

Since our algorithm is closely related to the P^{3}C algorithm, we will provide some more details about it here. P^{3}C requires the network to be chordal and takes a simplicial elimination ordering as input. It then performs two sweeps along this ordering.

The first sweep uses the DPC algorithm (reproduced here as Algorithm 1) to enforce a property called directed path consistency, introduced by Dechter et al. (1991). This property guarantees that every edge (v_i, v_j) is labelled with the weight of the shortest $v_i - v_j$ path through the subgraph

Algorithm 1: DPC

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ with associated weights $w_{i\to j}$ and an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$.
Output: CONSISTENT if $\mathcal{S}$ has been made DPC along d, or INCONSISTENT otherwise.

```
for k ← n to 1 do
    foreach i < j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
        w_{i→j} ← min{w_{i→j}, w_{i→k} + w_{k→j}}
        w_{j→i} ← min{w_{j→i}, w_{j→k} + w_{k→i}}
        if w_{i→j} + w_{j→i} < 0 then
            return INCONSISTENT
return CONSISTENT
```

induced by all vertices that precede both v_i and v_j in the ordering d. Formally, it is defined as follows:

Definition 1. A chordal network $\mathcal{S}$ is *DPC along the simplicial elimination ordering* $d = (v_n, v_{n-1}, \ldots, v_1)$ (also called d-DPC) if $w_{i\to j} \leq w_{i\to k} + w_{k\to j}$ for all $i, j < k$ where $v_i, v_j \in N(v_k)$.

The second sweep then follows the reverse of the elimination ordering d, using the fact that the STN is already DPC to inductively compute the globally shortest paths required for the graph to be PPC.

3 More on DPC

Our new algorithm relies on two key properties concerning DPC. First, if an STN is PPC, we can show that it is DPC along *any* simplicial elimination ordering. Second, it follows from the DPC algorithm that if the constraints adjacent to the last vertex in the ordering change, they cannot affect the constraints that have already been eliminated. In other words, the only constraints that need to be updated in this case to re-enforce DPC are those in the *direct neighbourhood* of the last event in the ordering. In this section we prove these two properties.

PPC and DPC

Lemma 1. *Given a chordal STN $\mathcal{S}$ which is PPC. Then, $\mathcal{S}$ is DPC along* any *simplicial elimination ordering.*

Proof. Since $\mathcal{S}$ is PPC, all edges are labelled with the length of the shortest path. Thus, for all triangles i, j, k it holds that $w_{i\to j} \leq w_{i\to k} + w_{k\to j}$. Now, let d be any simplicial elimination ordering. Since running DPC along d also does not introduce any new edges and no weights are adjusted, $\mathcal{S}$ is DPC along d. □

Next, we discuss how to select an ordering that minimises the amount of updates.

Lexicographic Breadth-First Search

As can be seen in Algorithm 1, when the DPC algorithm visits a vertex v_k in the simplicial elimination ordering d, it only considers the constraints between v_k and its direct, lower-numbered neighbours when adjusting weights. The weights of these neighbouring constraints may be affected by changes made earlier in the traversal. Now, if v_k appears in d *before* a newly inserted vertex a, and is not one of its neighbours, we observe that the DPC algorithm will not adjust any weights during its visit of v_k. We would therefore like an ordering in which a and its neighbours are visited as late as possible, ideally last. Assuming that the graph is chordal after insertion of vertex a, which then has $\delta_c(a)$ neighbours, the following lemma states that the ordering produced by Lex-BFS fulfils the requirement just stated:

Lemma 2. *Given a chordal graph $G = \langle V, E\rangle$ and a vertex $a \in V$ having $\delta(a)$ neighbours, the simplicial elimination ordering d of G produced by* Lex-BFS *starting at a satisfies $d(n) = a$ and $\{d(n - \delta_c(a)), \ldots, d(n-1)\} = N(a)$.*

Lex-BFS is a breadth-first search procedure where all vertices receive a label upon their first check, and labels are extended with the current value of a global counter. Vertices are then visited in lexicographical order of this label.

DPC on induced subgraphs

We now use Lemma 2 to demonstrate that recomputing DPC along this ordering in the subgraph induced by $\{a\} \cup N(a)$ *only* is sufficient to re-enforce DPC for the entire graph.

This is done in three steps. First we demonstrate that the subsequence d' of a simplicial elimination ordering d is a valid simplicial elimination ordering for the subgraph induced by the vertices in d'. We then revisit the definition of DPC, and show that it can also be expressed as a property of vertices relative to an ordering. Finally, we show that running DPC along the ordering in the subgraph induced by $\{a\} \cup N(a)$ re-enforces this property for all vertices, thus re-enforcing DPC on the entire graph.

Let $G\backslash\{v\}$ be a shorthand for $G_{V\backslash\{v\}}$, the graph induced by all vertices in V other than v. In other words, $G\backslash\{v\}$ is the graph from which the vertex v and all edges connected to v have been removed. We then have the following result:

Lemma 3. *Let $d = (v_n, v_{n-1}, \ldots, v_1)$ be a simplicial elimination ordering of the chordal graph G. Then the ordering $d' = (v_n, \ldots, v_{k+1}, v_{k-1}, \ldots, v_1)$ is a simplicial elimination ordering of the graph $G' = G\backslash\{v_k\}$ for every $1 \leq k \leq n$.*

Proof. First consider the vertices v_i with $i > k$. Since d is a simplicial elimination ordering of G, the vertices $C_i = \{v_j \mid v_j \in N(v_i), j < i\}$ induce a clique in G. A subset of vertices in a clique also induces a clique, so $C_i\backslash\{v_k\}$ is a clique in G'. Therefore v_i remains simplicial in d'.

Furthermore, for v_i with $i < k$, we know that C_i is a clique in G which cannot contain v_k since $i < k$. Since G and G' only differ in the removal of v_k, this clique remains unchanged in G', and v_i remains simplicial in d'.

Since v_k itself does not occur in d' and all other vertices remain simplicial, d' must indeed be a valid simplicial elimination ordering of G'. □

Let $d' = d\backslash V'$ be a sequence of all vertices $v \in V\backslash V'$, appearing in the same order as they appear in d. Repeatedly applying Lemma 3 for different vertices of the graph G yields the following corollary:

Corollary 1. *Given a simplicial elimination ordering d of a chordal graph $G = \langle V, E\rangle$ and an arbitrary subset $V' \subseteq V$. Then the ordering $d' = d\backslash V'$ is a simplicial elimination ordering for the graph $G' = G_{d'} = G_{V\backslash V'}$.*

Given an elimination ordering for the entire graph G, Corollary 1 allows us to derive an elimination ordering for the subgraph $G_{\{a\}\cup N(a)}$ induced by $\{a\} \cup N(a)$. We now need to show that re-enforcing DPC in this subgraph suffices to re-enforce DPC in the entire graph.

Unfortunately, DPC is defined in terms of an ordering over the *entire* graph. For Vertex-IPPC, we want to be able to reason about DPC in the context of an induced *subgraph*. Definition 2 addresses this issue by providing an alternative definition of DPC at the level of individual vertices.

Definition 2. A vertex v_k of an STN $\mathcal{S}$ has the *DPC property relative to the ordering $d = (v_n, v_{n-1}, \ldots, v_1)$* if $w_{i\to j} \leq w_{i\to k} + w_{k\to j}$ for all $i, j < k$ where $v_i,\ v_j \in N(v_k)$.

The following proposition shows that these definitions express the same notion of directed path consistency.

Proposition 4. *A network $\mathcal{S}$ is DPC along the ordering d (d-DPC) if and only if all its vertices have the DPC property relative to d.*

Proof. ($\Rightarrow$) If $\mathcal{S}$ is DPC along d, by Definition 1 the inequality $w_{i\to j} \leq w_{i\to k} + w_{k\to j}$ holds for every vertex v_k, and hence every v_k has the DPC property relative to d.

($\Leftarrow$) For the other direction, if every vertex has the DPC property relative to d, we know by Definition 2 that the inequality $w_{i\to j} \leq w_{i\to k} + w_{k\to j}$ holds for every v_k, and therefore that $\mathcal{S}$ is DPC along d. □

We use the shorthand *d-DPC* both to indicate that a vertex has the DPC property relative to d and to indicate that an STN is DPC along d.

The concept of the DPC property of a vertex allows us to prove the following intuitive notion. Suppose the first k vertices of an ordering d already have the DPC property. Then we need only run the DPC algorithm along the remaining vertices of d to enforce DPC on the entire graph. We formalise this in Lemma 5, where we use $\mathcal{S}_{d_2}$ to denote the subnetwork induced by the vertices in d_2 only.

Lemma 5. *Given an STN $\mathcal{S}$ and an elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$ of which the vertices $d_1 = (v_n, v_{n-1}, \ldots, v_{k+1})$ have the DPC property along d. Then running the DPC algorithm along $d_2 = (v_k, v_{k-1}, \ldots, v_1)$ in $\mathcal{S}_{d_2}$ makes $\mathcal{S}$ d-DPC.*

Proof. Because d_2 is a subsequence of d, it follows from Corollary 1 that d_2 is a simplicial elimination ordering of

$\mathcal{S}_{d_2}$. This means that we can run the DPC algorithm in $\mathcal{S}_{d_2}$ to enforce the DPC property along d_2 on all vertices in d_2.

When all vertices in d_2 are d_2-DPC, we know that all weights $w_{i\to j}$ are correctly set for $i, j < \ell$, where $v_i, v_j \in N(v_\ell)$ and $1 \leq \ell \leq k$. Since the vertices in d_1 are still d-DPC, the same holds for $k+1 \leq \ell \leq n$. Together, this means that the weights are correctly set for all $1 \leq \ell \leq n$, and therefore that $\mathcal{S}$ is d-DPC. □

We now have enough information to prove the principal theorem of this section. Let $\mathcal{S} = (V, C)$ be a PPC STN and consider the STN $\mathcal{S}' = (V', C')$ obtained by adding to $\mathcal{S}$ a new vertex a connected to $N(a) \subseteq V$ by the set of constraints C_a, i.e. with $V' = V \cup \{a\}$ and $C' = C \cup C_a$. Still assuming that graph is chordal after insertion of vertex a, which then has $\delta_c(a)$ neighbours, we show that running the DPC algorithm along a simplicial elimination ordering visiting only a and its neighbours re-enforces DPC on the entire network.

Theorem 1. *Let $\mathcal{S}'$ be an STN as just defined. Given a simplicial elimination ordering $d = (v_n, v_{n-1}, \dots v_1)$ of $\mathcal{S}'$ such that $\{v_{\delta_c(a)+1}, \dots, v_2\} = N(a)$ and $v_1 = a$. Then, running the DPC algorithm along $d' = (v_{\delta_c(a)+1}, \dots, v_1 = a)$ in $\mathcal{S}'_{\{a\}\cup N(a)}$ makes $\mathcal{S}'$ d-DPC.*

Proof. It suffices to show that each vertex $v_k \in (v_n, v_{n-1}, \dots, v_{\delta_c(a)+2})$ has the DPC property along d in $\mathcal{S}'$. The remainder of the proof then follows from Lemma 5.

Assume for a contradiction that one such v_k is *not* d-DPC; then there must be some $v_i,\ v_j$ such that $i, j < k$ and $v_i,\ v_j \in N(v_k)$, but $w_{i\to j} > w_{i\to k} + w_{k\to j}$. Since $\mathcal{S}$ was PPC already and none of the weights in $\mathcal{S}$ have changed, this inequality cannot hold if v_i, v_j and v_k are all in $\mathcal{S}$. Hence one of them must be a, the only vertex not in $\mathcal{S}$.

Since $k > \delta_c(a) + 1$ and $a = v_1$, v_k cannot be a. Furthermore, since $k > \delta_c(a) + 1$ and all neighbours v_ℓ of a have $\ell \leq \delta_c(a) + 1$, v_k cannot be a neighbour of a. But since v_i and v_j are neighbours of v_k, this means that v_i and v_j cannot be a either, a contradiction. Thus, vertices $(v_n, v_{n-1}, \dots, v_{\delta_c(a)+2})$ must indeed be d-DPC in $\mathcal{S}'$. □

4 Vertex-incremental PPC

In our new algorithm, we apply the observations on DPC from the previous section to present an algorithm that re-enforces PPC on an STN extended with a new event a and associated new constraints C_a. It is possible that these new constraints C_a break the chordality of the original STN $\mathcal{S}$. We therefore require that chordality be re-enforced on the network in a pre-processing step before we run Vertex-IPPC. We end this section with the discussion of an algorithm by Berry et al. (2006) which accomplishes this.

After pre-processing, we use the Lex-BFS algorithm to find a simplicial elimination ordering ending in a. Since the network is chordal, this ordering is guaranteed to exist by Lemma 2. Now we know by Theorem 1 that DPC may only be violated in the subgraph induced by a and its direct neighbours $N(a)$. Hence, it suffices to run the DPC algorithm in this subgraph to re-enforce DPC in the entire graph.

Algorithm 2: Vertex-IPPC

Input: A chordal STN $\mathcal{S}'$ obtained by extending the PPC STN $\mathcal{S}$ with (1) a new event a, (2) a set of constraints connecting a to the events in $U \subseteq V$ with weights $C_a = \{w_{a\to u}, w_{u\to a} \mid u \in U\}$, and (3) a set of constraints with infinite weight needed to make $\mathcal{S}'$ chordal.

Output: INCONSISTENT if the new constraints yield inconsistency, CONSISTENT if PPC has been enforced on the modified network $\mathcal{S}'$.

```
d ← Lex-BFS (S', a)
call DPC (S'_{N(a)∪{a}}, d)
return INCONSISTENT if DPC did
call SSSP-PPC (S', d, a)
return CONSISTENT
```

Algorithm 3: SSSP-PPC($\mathcal{S}, d, a$)

Input: A DPC STN $\mathcal{S} = \langle V, E\rangle$ which is PPC on $\mathcal{S}\backslash\{a\}$ and has associated weights $\{w_{i\to j}, w_{j\to i} \mid \{i, j\} \in E\}$, and a simplicial elimination ordering $d = (v_n, v_{n-1}, \dots, v_2, v_1 = a)$.

Output: The PPC network $\mathcal{S}$.

```
1 D_{a→}[a] ← 0; D_{a←}[a] ← 0
2 VISITED[a] ← TRUE
3 for k ← 2 to n do
4 |   D_{a→}[v_k] ← ∞; D_{a←}[v_k] ← ∞
5 |   call Visit(v_k)
```

Since the only change is the introduction of a new event, we know that if a constraint $c_{i\to j}$ needs to be tightened, the new shortest path must contain the newly added vertex a. We therefore maintain single-source shortest paths from a to every vertex and single-sink shortest paths from every vertex to a, and tighten all constraints in one outward sweep along d. Since at this point the network is already DPC, we can use a slightly modified version of the SSSP procedure used by Planken et al. (2010) and referred to here as DPC-SSSP.

The original DPC-SSSP algorithm calculates the single-source/sink shortest paths (SSSP) distances from a single vertex a to every other vertex in the graph and vice versa. By exploiting the knowledge that the graph is already DPC, DPC-SSSP visits every edge in the graph only twice, achieving a run-time bound of $\mathcal{O}(m_c)$.

Since any new shortest path from i to j must go through a, it must be of the form $i \dashrightarrow a \dashrightarrow j$. Moreover, DPC-SSSP calculates exactly the weights of these paths $i \dashrightarrow a$ and $a \dashrightarrow j$ for all i and j. Thus we can re-enforce partial path consistency in the same sweep. Our modified algorithm, dubbed SSSP-PPC, is shown in Algorithm 3.

We prove the correctness of SSSP-PPC in two steps. First, in Lemma 6 we adapt the correctness proof of DPC-SSSP to show that SSSP-PPC does in fact calculate the correct weights for the shortest paths between a and any other ver-

Procedure Visit($v \in V$)

1 **foreach** $u \in N(v)$ **such that** VISITED$[u]$ = TRUE **do**
2 $\quad D_{a\rightarrow}[v] \leftarrow \min\{D_{a\rightarrow}[v], D_{a\rightarrow}[u] + w_{u\rightarrow v}\}$
3 $\quad D_{a\leftarrow}[v] \leftarrow \min\{D_{a\leftarrow}[v], w_{v\rightarrow u} + D_{a\leftarrow}[u]\}$
4 VISITED$[v] \leftarrow$ TRUE
5 **foreach** $u \in N(v)$ **such that** VISITED$[u]$ = TRUE **do**
6 $\quad w_{u\rightarrow v} \leftarrow \min\{w_{u\rightarrow v}, D_{a\leftarrow}[u] + D_{a\rightarrow}[v]\}$
7 $\quad w_{v\rightarrow u} \leftarrow \min\{w_{v\rightarrow u}, D_{a\leftarrow}[v] + D_{a\rightarrow}[u]\}$

tex. This information is then used in Lemma 7 to show that a run of SSSP-PPC enforces partial path consistency on the network we encounter in line 4 of Vertex-IPPC.

Lemma 6. *When* VISITED$[v]$ *is* TRUE*,* $D_{a\leftarrow}[v]$ *and* $D_{a\rightarrow}[v]$ *are set to the total weight of the shortest paths from* v *to* a *and from* a *to* v *respectively.*

Proof. The proof is by induction along the simplicial *construction* ordering $d^{-1} = (v_1, v_2, \ldots, v_n)$ with $v_1 = a$. For the base case, $D_{a\leftarrow}[a]$ and $D_{a\rightarrow}[a]$ are set correctly in line 1 of SSSP-PPC. Now, for the induction step, assume that the weight $D_{a\rightarrow}[v]$ is set correctly for all $v \in \{v_1, \ldots, v_k\}$; we show that SSSP-PPC sets the correct weight for $D_{a\rightarrow}[v_{k+1}]$. The proof for $D_{a\leftarrow}[v_{k+1}]$ is analogous.

Assume for a contradiction that VISITED$[v_{k+1}]$ is TRUE but that $D_{a\rightarrow}[v_{k+1}]$ does not hold the weight of shortest path. Then there must be some path $\pi = a \rightarrow v_{j_1} \rightarrow \cdots \rightarrow v_{j_\ell} \rightarrow v_{k+1}$ with weight $w_\pi < D_{a\rightarrow}[v_{k+1}]$. If there is a vertex $j_{\max} = \max_i j_i$ on π such that $j_{\max} > k$, we know by DPC that we can replace this vertex by a shortcut from its predecessor to its successor, without increasing the weight. By repeating this procedure we can remove all such v_{j_i} from the path π without increasing its total weight.

In particular, this means that $j_\ell \leq k$, which implies that VISITED$[v_{j_\ell}]$ is TRUE and therefore, by the induction hypothesis, that $D_{a\rightarrow}[v_{j_\ell}]$ has been correctly set. But then the value of $D_{a\rightarrow}[v_{k+1}]$ is correctly updated by the assignment on line 2 of Visit(v_{k+1}), contradicting our assumption that $w_\pi < D_{a\rightarrow}[v_{k+1}]$. □

Lemma 7. *Given a* d*-DPC STN* $\mathcal{S}$*, where* a *is the last vertex of* d *and* $\mathcal{S}\backslash\{a\}$ *is PPC. Then the network* $\mathcal{S}'$ *produced by running SSSP-PPC on* $\mathcal{S}$ *along the reverse of* d *is PPC.*

Proof. Consider the edges adjacent to a, i.e. $\{a, v\}$ with $a \in N(v)$. Before the second loop of Visit(v) is executed, VISITED$[a]$ and VISITED$[v]$ are both TRUE, by lines 2 of SSSP-PPC and 4 of Visit, respectively. Hence, by Lemma 6, $D_{a\rightarrow}[v]$ and $D_{a\leftarrow}[v]$ hold the weights of the corresponding minimum paths between v and a. So the weights of $w_{a\rightarrow v}$ and $w_{v\rightarrow a}$ are correctly updated in lines 6 and 7 of Visit.

Now consider any other edge $\{u, v\}$ neither endpoint of which is a. Since $\mathcal{S}\backslash\{a\}$ was PPC, the only way in which the weight $w_{u\rightarrow v}$ can be further reduced is if there is some shorter path $u \dashrightarrow a \dashrightarrow v$ through a. Without loss of generality, assume that Visit(u) is called before Visit(v). Then VISITED$[u]$ and VISITED$[v]$ are both TRUE before the execution of the second loop in Visit(v). By the same reasoning as used above for a and its neighbours, line 6 of Visit correctly updates $w_{u\rightarrow v}$. The proof for $w_{v\rightarrow u}$ is analogous, reversing all arcs and using line 7 of Visit(v) instead. □

Correctness and efficiency

Having shown the correctness of the individual steps of the Vertex-IPPC algorithm, we now complete our analysis by demonstrating that these steps do indeed re-enforce partial path consistency when executed in sequence.

Theorem 2. Vertex-IPPC *correctly re-enforces PPC or decides inconsistency in* $\mathcal{O}(m_c + \delta_c(a)w_d^2)$ *time.*

Proof. The simplicial elimination ordering d of $\mathcal{S}'$ is obtained using Lex-BFS, and therefore, by Lemma 2, the last vertices in d are the neighbours of a, followed by a itself as final vertex. Since $\mathcal{S}$ was PPC, by Theorem 1 the call to DPC on the sub-graph consisting of a and its neighbours re-enforces DPC along d for the new $\mathcal{S}'$, or concludes inconsistency. Finally, by Lemma 6, running SSSP-PPC along $\mathcal{S}'$ re-enforces PPC on $\mathcal{S}'$.

We now turn to the time complexity. Lex-BFS takes $\mathcal{O}(m_c)$ time to construct the ordering d, and running DPC on an induced graph with $\delta_c(a)+1$ nodes takes $\mathcal{O}(\delta_c(a)w_d^2)$ time. Finally, SSSP-PPC takes $\mathcal{O}(m_c)$ time: Visit is called once per vertex and all operations in Visit take amortised constant time per edge. □

Incremental triangulation

So far we have focused exclusively on the *correctness* of Vertex-IPPC. We now discuss an important *advantage* of our algorithm over Edge-IPPC: it integrates well with an incremental triangulation algorithm. Recall that Edge-IPPC requires that the edge (a, b), representing the constraint $c_{a\rightarrow b}$ to be tightened, is already part of the underlying structure. If the edge is not present, a new triangulation must be found in which it does exist. We cannot simply include (a, b), since it may introduce a new chordless cycle. This would break the chordality required for PPC algorithms.

For Vertex-IPPC, we can address this issue by using a recently discovered algorithm by Berry et al. (2006). This algorithm performs *vertex-incremental minimal triangulation*: given a chordal graph $G = \langle V, E\rangle$, a new vertex a, and a set of edges E_a connecting a to existing vertices of G, it can compute the *minimal* fill F such that the graph $G' = \langle V \cup \{a\}, E \cup E_a \cup F\rangle$ is chordal.

Their algorithm needs $\mathcal{O}(n)$ time to insert one edge, and since there are m edges it takes $\mathcal{O}(nm)$ time to triangulate a graph from scratch. An interesting observation here is that if we average this time over the number of vertices, we obtain an amortised bound of $\mathcal{O}(m)$ time to insert one vertex. Since $m = \mathcal{O}(m_c)$, this fits comfortably in the $\mathcal{O}(m_c + \delta_c(a)w_d^2)$ bound for one run of Vertex-IPPC.

We learned from the authors that they were unaware of any existing implementation of this algorithm. To evaluate our algorithm, we therefore had to implement it ourselves. This turned out to be not exactly trivial, but after having found an efficient way to maintain references to the cliques in the maintained clique-tree, we were able to stay within the established bounds (ten Thije, 2011).

5 Empirical evaluation

Having established the theoretical soundness and efficiency of our algorithm, we now turn to assess its performance in practice. In particular, we want to answer two questions. First, does our implementation meet the run time bound of $\mathcal{O}(m_c + \delta_c(a)w_d^2)$ derived in Theorem 2, and second, how does its performance compare to Edge-IPPC?

Regarding the second question, comparing a vertex-incremental algorithm to an edge-incremental one may seem strange. However, we observe that Edge-IPPC can be used to enforce PPC after a vertex has been inserted, using the following simple approach. After the graph has been pre-processed such that it contains a and will still be chordal after all new constraints are added, we simply run Edge-IPPC once for all constraints ($\delta(a)$ in number) attached to a. Since each run of Edge-IPPC takes $\mathcal{O}(m_c)$ time, this sequential algorithm re-enforces PPC in $\mathcal{O}(\delta(a) \cdot m_c)$ time.

We expect that Vertex-IPPC outperforms this sequential method. The main reason is that w_d^2 is the number of edges in the largest clique, which is usually far smaller than m_c, the number of edges in the entire chordal graph. While $\delta(a)$ is smaller than $\delta_c(a)$, we expect this will have only little effect. After all, increasing $\delta_c(a)$ only results in more work in the new clique containing a, while increasing $\delta(a)$ yields more runs spanning the entire graph.

Hypothesis 1 and Hypothesis 2 present the expected answers to respectively the first and second question posed above.

Hypothesis 1. *The run time of our implementation of Vertex-IPPC meets the theoretical bound of* $\mathcal{O}(m_c + \delta_c(a)w_d^2)$.

Hypothesis 2. *When inserting a new vertex* a *in a graph, running Vertex-IPPC is faster than running Edge-IPPC* $\delta(a)$ *times.*

Data sets

Since we aim to assess asymptotic behaviour, we require graphs with large numbers of vertices. Moreover, since we concern ourselves with problems related to scheduling, we would like input graphs from this domain as well.

We therefore used a generator for STNs based on *Hierarchical Task Networks* (HTNs), a planning paradigm formalised by Erol et al. (1994). This generator was used by Planken et al. (2010) to empirically evaluate the Edge-IPPC algorithm. We briefly summarize their discussion of HTNs; for more details we refer to the original papers.

In short, an HTN represents a hierarchy of tasks, in which high-level tasks are progressively decomposed into collections of smaller tasks as we go further down the hierarchy. A general HTN can contain multiple different such decompositions of the same task. However, such choices are beyond the expressive power of STNs. We therefore only consider HTNs whose tasks can be decomposed in at most one way.

Constraints in an HTN occur only between a parent task and its children, or between sibling tasks. The latter constraints are called sibling-restricted (SR) edges. Given this restriction, it is not possible to coordinate the execution of tasks in the different branches of the hierarchy when we strictly adhere to the HTN format. In order to circumvent this, the definition of an HTN may be extended to include so-called *landmark variables* (Castillo et al., 2006), which do allow synchronisation between branches. Finally, each task is modelled as a pair of events in the corresponding STN: its start and its end.

Tasks	100
Branch depth [min,max]	$[3, 6]$
Branches [min,max]	$[3, 10]$
Landmark ratio	0.20
Probability of SR-edge	0.50

Table 1: Parameters for generating HTN graphs.

In the interest of reproducibility, the settings we used to generate our HTNs are listed in Table 1. Using these settings, each generated graph contained 220 vertices. Our benchmark set consisted of 80 such graphs.

To assign the weights, we used a utility provided with the HTN generator by Planken et al. (2010). We configured this utility to first ensure a solution exists, by assigning a random time between -50 and 100 to each event. Then, the weight for each constraint in the network was calculated as follows. First, the utility determined the minimum possible weight such that the random solution was still possible. It then increased this weight by a value selected randomly from the range $[0, 150]$. Hence the original solution still existed, but had been "obscured" by the randomly added slack.

In order to test the influence of the graph *structure* on performance, we also generated a set of chordal graphs. This set was also created with the HTN generator and the weight setting utility. However, for this set we first added the fill edges as inserted by the minimum degree heuristic, before assigning weights.

Finally, we used sets of job-shop and scale-free graphs which the original Edge-IPPC algorithm was benchmarked on. We refer to Planken et al. (2010) for a discussion of the properties of these graphs.

Method

In order to assess performance, we used the following procedure. For each experiment we incrementally constructed three graphs: two graphs used to benchmark Vertex-IPPC and Edge-IPPC, and one control graph on which we ran the P^3C algorithm to verify that our implementation was correct.

All three graphs were initially empty. During the experiment we inserted the vertices from the input graph into the three experiment graphs one by one, as well as all constraints connecting the new vertex to the other vertices already in the graph. The order in which vertices were inserted was chosen randomly. When a vertex and its constraints were inserted, we first used the vertex-incremental triangulation scheme by Berry et al. (2006) to ensure the resulting graph was chordal. We then re-enforced partial path consistency by three different methods. On the first graph we ran Vertex-IPPC, starting from the newly added vertex. On the second graph, we ran Edge-IPPC for each constraint attached to the new vertex a,

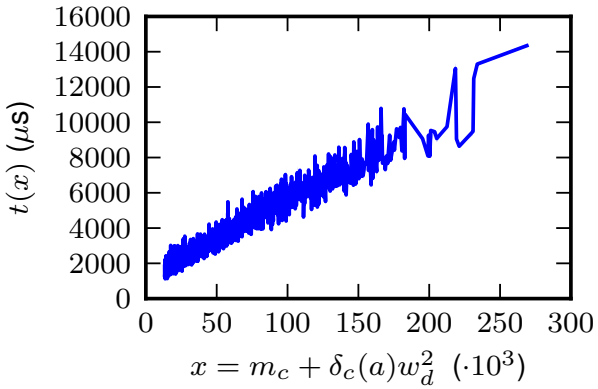


Figure 1: Run time for Vertex-IPPC on HTN graphs consisting of 220 vertices.

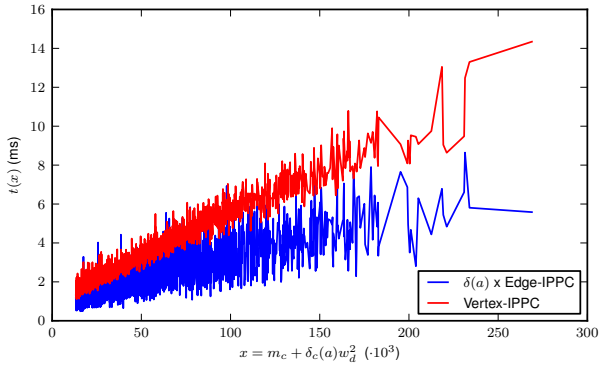


Figure 2: Run time comparison between Vertex-IPPC and Edge-IPPC on HTN graphs.

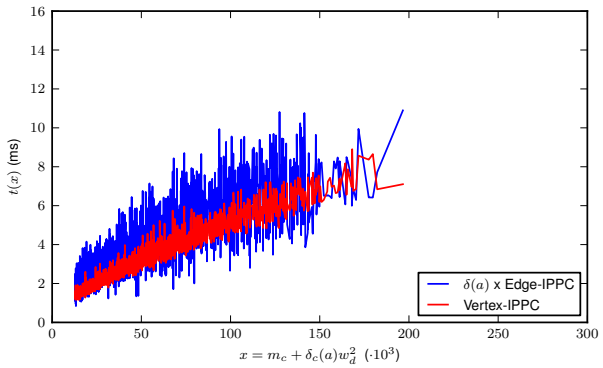


Figure 3: Run time comparison between Vertex-IPPC and Edge-IPPC on chordal HTN graphs.

for a total of $\delta(a)$ runs. Finally, we ran P^{3}C on the third graph.

At the end of each step, we made sure the weights in the graphs maintained by Vertex- and Edge-IPPC were equivalent to the weights enforced by P^{3}C, thus verifying that the implementation was correct.

Results and discussion

Verification of theoretical bounds The aim of our first experiment was to verify our implementation against the theoretical upper bound of $\mathcal{O}(m_c + \delta_c(a)w_d^2)$ derived in Theorem 2. We therefore plot the run time required to insert a vertex as function of the complexity of the graph at the time of insertion. By taking $x = m_c + \delta_c(a)w_d^2$ as measure of complexity, we should see a straight line if our implementation meets this bound on the given input graphs. Fig. 1 shows this plot for the normal HTN graphs, and we do indeed observe a line with a clear linear trend. In other words, our experiments support Hypothesis 1 and we conclude that we cannot reject it.

Comparison to Edge-IPPC In our second experiment we compare the performance of using Vertex-IPPC to insert a vertex at once, to the performance of running Edge-IPPC once for every constraint adjacent to the new vertex. We initially ran this experiment on the set of normal HTN graphs and obtained the results summarised in Fig. 2.

Rather surprisingly, we find that Edge-IPPC is significantly faster than Vertex-IPPC in this benchmark. The results for the job-shop and scale-free graphs were similar, so in the interest of saving space we omit these graphs. Hence, Hypothesis 2 does not hold in general, and we reject it.

A possible explanation for this somewhat disappointing result is that the degrees of the new vertices in the chordal graph may have been much higher than their degrees in the original graph. Recall that Edge-IPPC needs to be executed only once for each constraint in the *original* graph, and completes in time linear in the number of edges. On the other hand, the run time of the DPC algorithm used by Vertex-IPPC depends on the size of the neighbourhood in the *triangulated* graph. Running DPC is relatively expensive, so if there are only a few real constraints but many fill edges, this investment may not pay off.

In particular, we may have the following pathological case. Suppose the new vertex a has only two constraints involving older vertices in the original graph. However, these constraints create a cycle involving all n vertices, and thus after triangulation, a is connected to *all* vertices in the chordal graph. In this case we need only run Edge-IPPC twice, which can be done in $\mathcal{O}(m_c)$ time, while the DPC step alone takes $\mathcal{O}(nw_d^2)$ time. Conversely however, if nearly all edges in the chordal graph correspond to actual constraints, the Edge-IPPC method will take $\mathcal{O}(nm_c)$ time, which is worse than the $\mathcal{O}(nw_d^2)$ that dominates the run time of Vertex-IPPC in this case.

We summarise the above in the following new hypothesis, a more specific form of Hypothesis 2:

Hypothesis 3. *Given that $\delta(a) = \delta_c(a)$ when a new vertex a is inserted in a graph, running Vertex-IPPC is faster than running Edge-IPPC $\delta(a)$ times.*

To test this hypothesis, we ran our experiment on the input set consisting of chordal HTNs. Since these graphs are already triangulated, we do not need to add *any* fill edges, which should give Vertex-IPPC an advantage over Edge-

IPPC. The results for this experiment are shown in Fig. 3. This figure does indeed show a better relative performance for Vertex-IPPC, but the difference is small. Since Vertex-IPPC does not clearly perform better than Edge-IPPC, we have to reject Hypothesis 3.

A final possibility to explain these results is the following. It may be that the consecutive runs of Edge-IPPC in the neighbourhood of the new vertex "help" each other. More specifically, it may be that two constraints c_1 and c_2 imply a lower weight for some constraint c_3 than c_3 is itself labelled with in the source graph. If Edge-IPPC enforces c_1 and c_2 first, it can then immediately detect that the weight of c_3 need not change, thus reducing the execution cost. If this happens frequently, then Edge-IPPC essentially gets a few runs "for free". Vertex-IPPC on the other hand always needs to pay the cost of running DPC, whether constraints subsume each other or not.

While this hypothesis does not offer a way to improve the performance of Vertex-IPPC, it can be used to improve that of Edge-IPPC. In particular, it would be interesting to see whether there is some way to put the constraints inserted by Edge-IPPC in such an order that the "help" is maximised. We will not pursue this idea here, but note it as an interesting direction for future work.

6 Conclusions and future work

We showed that DPC can be defined on subnetworks of an STN as well, and that running the DPC algorithm along the neighbours of an inserted vertex is sufficient to guarantee DPC on the complete graph. This idea is used in Vertex-IPPC, a new algorithm to re-enforce partial path consistency in an STN when it is extended with a new event and associated constraints. The run time of this algorithm is bounded by m_c, the number of edges in the chordal graph and nw_d^2, where n is the number of vertices in the network and w_d is the network's induced width.

Our algorithm integrates nicely with the vertex-incremental triangulation method recently presented by Berry et al. (2006). We implemented this triangulation method for our experiments, which, to the best of our knowledge, had not been done before.

Surprisingly, these experiments on HTN, job-shop and scale-free graphs show that Vertex-IPPC is outperformed by sequentially inserting the new constraints using the existing Edge-IPPC algorithm. On chordal networks, which offer the greatest advantage for Vertex-IPPC relative to Edge-IPPC, both algorithms show similar performance in practice. We conjecture that the sequential method may benefit from the order in which constraints are inserted. As part of our future work we aim to investigate this idea and determine if an optimal ordering can be found to further optimise the performance of Edge-IPPC.

Secondly, methods are under development to also deal with constraint loosening and edge deletions (ten Thije, 2011). With these additions, we expect to arrive at a fully dynamical method that not only performs significantly better than any existing approach for dynamically solving STNs, but can immediately be used as part of a planning or scheduling algorithm.

Bibliography

S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods.*, 8(2):277–284, 1987.

A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Mathematics*, 306(3):318–336, 2006.

C. Bliek and D. Sam-Haroud. Path consistency on triangulated constraint graphs. In *Int, Joint Conf. on Artificial Intelligence*, volume 16, pages 456–461, 1999.

J. Bresina, A. Jónsson, P. Morris, and K. Rajan. Activity planning for the mars exploration rovers. In *Proc. 14th Intl. Conf. on Automated Planning and Scheduling*, pages 40–49, 2005.

L. Castillo, J. Fernández-Olivares, O. Garcıa-Pérez, and F. Palao. Efficiently handling temporal knowledge in an HTN planner. In *Proc. 16th Intl. Conf. on Automated Planning and Scheduling*, 2006.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, pages 249–254, 1994.

P. Labone and M. Ghallab. Planning with sharable resource constraints. In *Proc. 14th Intl. Joint Conf. on Artificial Intelligence*, pages 1643–1649, 1995.

L. R. Planken. Incrementally solving the STP by enforcing Partial Path Consistency. In *Proc. 27th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 87–94, 2008.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. P3C: A New Algorithm for the Simple Temporal Problem. In *Proc. Intl. Conf. on Automated Planning and Scheduling*, pages 256–263, 2008.

L. R. Planken, M. M. de Weerdt, and N. Yorke-Smith. Incrementally solving STNs by enforcing Partial Path Consistency. In *Proc. 20th Intl. Conf. on Automated Planning and Scheduling*, pages 129–136, 2010.

D. Smith, J. Frank, and A. Jónsson. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, 15(1):47–83, 2000.

J. O. A. ten Thije. *Towards a Dynamic Algorithm for the Simple Temporal Problem*. MSc thesis, Delft University of Technology, 2011.

D. West et al. *Introduction to graph theory*, volume 1. Prentice Hall Upper Saddle River, NJ, 2001.

L. Xu and B. Y. Choueiry. A new efficient algorithm for solving the Simple Temporal Problem. In *Proc. 10th Intl. Symposium on Temporal Representation and Reasoning and 4th Intl. Conf. on Temporal Logic*, 2003.

Integration of Node Deployment and Path Planning in Restoring Network Connectivity

Thuy T. Truong, Kenneth N. Brown and Cormac J. Sreenan
Mobile & Internet Systems Laboratory and Cork Constraint Computation Centre,
Department of Computer Science, University College Cork, Ireland
Email: {tt11, k.brown, cjs}@cs.ucc.ie

Abstract

A wireless sensor network can become partitioned due to node failure, requiring the deployment of additional relay nodes in order to restore network connectivity. This introduces an optimisation problem involving a tradeoff between the number of additional nodes that are required and the costs of moving through the sensor field for the purpose of node placement. This tradeoff is application-dependent, influenced for example by the relative urgency of network restoration. We propose two heuristic algorithms which integrate network design with path planning, recognising the impact of obstacles on mobility and communication. We conduct an empirical evaluation of the two algorithms on random connectivity and mobility graphs, showing their relative performance in terms of node and path costs, and assessing their execution speeds. Finally, we examine how the relative importance of the two objectives influences the choice of algorithm.

Introduction

Wireless Sensor Networks consist of multiple sensing and relay units which communicate with each other using radios, exchanging information, making joint decisions on network and sensing configurations, and transmitting their data over multiple hops to one or more sink nodes which have access to the wider world. WSNs are becoming increasingly important for monitoring phenomena in remote or hazardous environments, including pollution monitoring, chemical process sensing, disaster response, and battlefield monitoring. As these environments are uncontrolled and may be volatile, the network may suffer damage, from hazards, direct attack or accidental damage from wildlife and weather. They may also degrade through battery depletion or hardware failure. The failure of an individual sensor node may mean the loss of particular data streams generated by that node; more significantly, node failure may partition the network, meaning that many data streams cannot be transmitted to the sink. This creates the network repair problem, in which we must place new radio nodes in the environment to restore connectivity to the sink for all sub-partitions.

There are four main subtasks in the problem: (i) determining what damage has occurred (i.e. which nodes have failed and what radio links have been blocked); (ii) determining what changes, if any, have happened to the accessibility of the environment (i.e. what positions can be reached, and what routes are possible between those positions); (iii) deciding on the positions for the new radio nodes; and (iv) planning a route through the environment to place those nodes. The problem thus involves both exploration and optimisation, and depending on circumstances may require the placement of nodes before the changes to connectivity and accessibility have been fully mapped. In this paper, we consider the simpler problem in which we assume the exploration tasks have already been completed, and so our aim is to optimise our use of resources in the static fully observed problem. We assume possible locations for new radio nodes are limited to a finite set of positions where a node can be securely placed and which can be accessed. Radio nodes are expensive, and thus solutions which require fewer nodes are preferred. Physically moving around the environment may be expensive in energy use, may take significant time, or may expose the agent placing the nodes to danger, and thus solutions which allow cheaper path plans are also preferred. Depending on the application, either one of the two objectives may be more important: placing expensive nodes in, for example, agricultural pollution monitoring favours solutions with fewer nodes, while restoring connectivity during disaster response favours solutions that can be deployed quickly even if they require more nodes. Thus the network repair problem is multi-objective.

Our contribution is the novel problem of simultaneous network connectivity restoration with constrained route planning, in the presence of obstacles, and the development and analysis of two heuristic algorithms. We assume a known connectivity graph which includes all possible new node locations and existing nodes, and where the edges indicate that two positions could communicate with each other. We also assume a mobility graph over the same set of positions, but where the edges represent a possible path between two positions. We propose two heuristic algorithms which integrate network design with path planning, and which trade-off the objectives of node cost and path cost. Our first algorithm, called Shortest Cheapest Path (SCP), prioritises node cost, and first finds the minimum number of nodes re-

quired to heal the network; it then finds the cheapest path which visits all the new node positions. The second algorithm Integrated Path (IP) integrates the two objectives by adding weights into the connectivity graph to approximate the mobility cost of establishing each link, and then searches for the cheapest tree that connects all existing nodes. We conduct an empirical evaluation of the two algorithms on random connectivity and mobility graphs. The SCP algorithm tends to find graphs with fewer nodes, while the IP algorithm finds slightly larger solutions but with cheaper mobility costs. The SCP algorithm is significantly faster, particularly on dense graphs. Finally, we examine how the relative importance of the two objectives influences the choice of algorithm.

Preliminaries

An undirected graph G is a pair (V,E), where V is a set of vertices, and E is a set of edges = $\{(x, y) : x \in E, y \in E\}$. We can augment a graph with a cost function, which is normally either a vertex-weight $w : V \rightarrow N$ assigning a cost to each vertex, or an edge-cost $c : E \rightarrow N$ assigning a cost to each edge. A subgraph S of G is a graph $S = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. The vertex-weight of S is then $\Sigma_{v \in V'} w(v)$, while the edge-cost of S is $\Sigma_{e \in E'} c(e)$. A path P from vertex x_0 to x_n in a graph G is a sequence of edges $< (x_0, x_1), (x_1, x_2), ..., (x_{n-1}, x_n) >$, where each edge $(x_i, x_{i+1}) \in E$. The edge cost of a path P is $\Sigma_{e \in P} c(e)$. A circuit is a path in which $x_n = x_0$. A cycle is a circuit in which $x_0 (= x_n)$ is the only vertex that appears twice. A connected graph is one in which there is a path between every pair of vertices. The problem of finding a minimal cost circuit in an edge-weighted connected graph is NP-hard. A tree is a connected graph with no cycles, and a subtree of a graph is simply a subgraph which is also a tree. For a graph G=(V,E), a spanning tree is a subtree $T = (V, E')$. If we select a subset τ of V (the terminals), then a Steiner tree for τ in G is a subtree $T = (V', E')$ in which $\tau \subseteq V'$. In other words, a spanning tree is a subtree that spans all vertices of a given graph while a Steiner tree is a subtree that spans a given subset of vertices. The problem of finding a minimal Steiner tree in an edge-weighted graph is NP-hard, and remains NP-hard even if all edge-costs are equal. If the edge costs are all the same, then the problem is equivalent to that of finding a minimal vertex-weighted Steiner tree with equal costs.

A wireless network can be represented by an undirected graph G=(V,E), where the vertices represent radio nodes and the edges represent viable radio links between the nodes. We assume all radio links are symmetric, and thus the graph is undirected with at most one edge between any pair of vertices.

The Network Repair Problem

We assume all possible accessible positions for the radio nodes are known, as are the potential radio links between them. Some connected components of the original network will still exist, and our aim is to select enough new positions for radio nodes to create a connected graph. We represent this problem as a set τ, where each $v \in \tau$ is a connected component, and a connectivity graph $G_C = (V, E_C)$, where $\tau \subseteq V$ and each vertex $v \in V - \tau$ is a possible location, and each edge $(v_i, v_j) \in E_C$ represents a potential radio link between the two positions. If $v_i \in \tau$ or $v_j \in \tau$ then the edge represents a potential link from the new position to any radio in the connected component. We assume all radios have the same cost, and so we associate a unit vertex weight function w with G_C, such that $w(v) = 1$ for each $v \in V$, representing the cost of positioning a radio at that position. Our first aim is then to find a Steiner tree S in G_C for τ; that is, a connected set of vertices that includes all components in τ. If we find a minimal Steiner tree (minimising $w(S)$) then we ensure as few radios are used as possible.

We assume accessibility paths are known between the different candidate radio positions, and that there is a known cost of moving between any pair of positions, where the cost may represent time, energy, distance or hazard. We represent this as a graph $G_M = (V, E_M)$ with an associated edge cost function c. For any set $V' \subseteq V$, a circuit P in G_M that visits all vertices in V' represents a tour for the agent, and we can compute the associated path cost. For any given Steiner tree $S = (V', E'_C)$, a circuit P in G_M that visits every vertex $v \in V' - \tau$ then represents a possible tour in which the agent can place all necessary radio nodes to reconnect the network. Minimising $c(P)$ ensures that the cheapest circuit is selected. Note that the two objectives may conflict: a larger Steiner tree may allow a cheaper path, and more expensive path may be required for a smaller Steiner tree.

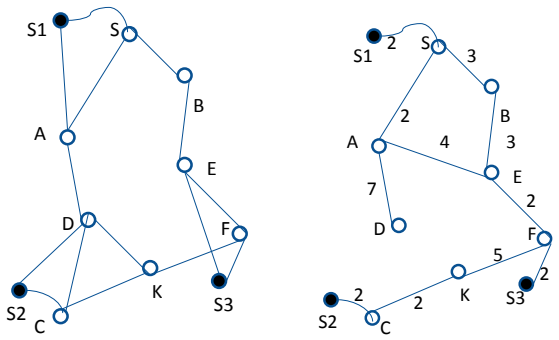
We can now state the formal problem:

- PROBLEM: Network Repair.
- INSTANCE: A graph $G_C = (V, E_C)$ with a unit vertex weight function ($w(v) = 1$, for all $v \in V$), a graph $G_M = (V, E_M)$ with an edge cost function c, and a terminal set $\tau \subseteq V$.
- OBJECTIVE: Find a Steiner tree S for G_C, where $S = (V', E'_C)$, and a circuit P in G_M, such that each $v \in (V' - \tau)$ appears in the path P, which minimises the pair of objectives ($w(S)$,$c(P)$).

As an example, Figure 1 shows a connectivity graph and a mobility graph for a set of terminals $\tau = \{S_1, S_2, S_3\}$ and a set of candidate locations {A,B,C,D,E,F,K,S}. The minimal Steiner tree in the connectivity graph has the vertex set $\{S_1, S_2, S_3, A, D, K, F\}$. However, this is not an easy tree for the agent to create, since there is no short path between D and K. The Steiner tree $\{S_1, S_2, S_3, S, B, E, F, K, C\}$ requires two extra radios, but allows a shorter circuit. Which of these solutions should be selected will depend on the relative cost of the radio nodes compared to the path cost. High radio costs and low mobility costs will prefer the first tree, while high mobility costs will prefer the second tree.

The Shortest Cheapest Path SCP Algorithm

Our first approach assigns an ordering to the objectives. We first try to minimise the number of nodes required to connect all terminals. Given a minimal set of nodes, we then try to



(a) Connectivity Graph (b) Mobility Graph

Figure 1: Example connectivity graph and mobility graph.

find the cheapest circuit for the agent that visits all of those nodes.

Data: A graph $G_C = (V, E_C)$, a set of terminals $\tau \subset V$.
Result: A Steiner Tree $T = (V', E'_C)$ for τ in G_C.
begin
  $G_\tau(\tau, E_\tau, w_\tau) = metric\_closure(\tau, G_C)$;
  $T_\tau = Minimum\_Spanning\_Tree(G_\tau)$;
  $T \leftarrow 0$;
  for *each edge* $e_{uv} \in E(T_\tau)$ **do**
    if *node u and node v are not connected in T* **then**
      $P = shortest\_path\_replace(e_{uv})$;
      if *P contains less than two vertices in T* **then**
        Add P *to* T;
      end
      else
        Add to T all edges in P which are not already in the tree T, and do not create a cycle in T;
      end
      if *T is connected and includes all terminals* **then**
        break;
      end
    end
  end
  return T;
end

Algorithm 1: Steiner-MST Algorithm

Since the Minimum Steiner Tree in Graphs problem is NP-Hard, we use a heuristic algorithm, adapted from (Wu and Chao 2004) (Algorithm 1). First, we create a metric closure graph for the terminals, which is a complete graph over the terminal nodes, where each edge has a weight equal to the shortest path between the two endpoints in the original graph. We obtain the metric closure graph by repeated application of Dijkstra's algorithm. We then find a minimum spanning tree in that graph, using Kruskal's algorithm. We then consider each edge in that tree in turn and, if the end points are not yet connected in the new tree, select the corresponding shortest path from the original graph. If this shortest path contains no more than 1 vertex already added, we add all edges into the tree, with their required vertices; if it contains more than 1 vertex already added, then we add those edges that are not already in the graph, and their associated vertices if needed. The most expensive operation is the creation of the metric closure graph, and thus the entire algorithm has complexity $O(|\tau||V|^2)$.

For the problem of finding the shortest circuit (Algorithm 2), we take all the non-terminal nodes in the Steiner tree created above, and then for those vertices create a metric closure graph from the mobility graph G_M. We then apply (Lawler et al. 1985)'s Greedy-TSP heuristic which is based on Kruskal's algorithm - we sort the edges in increasing order of cost, and we then iteratively add the lowest cost edge which does not increase any vertex's degree to 3, and which does not create a cycle unless it completes the tour. The runtime is again dominated by the time of building the metric closure graph, i.e. $O(|V'||V|^2)$.

Data: A set of vertices V', a graph $G_M = (V, E_M, c)$
Result: a tour in G_M visiting all nodes in V'
begin
  $G'' = (V'', E'', w'') = metric\_closure(V', G_M)$;
  $P = Greedy\_TSP(G'')$;
  return [V'',P] ;
end

Algorithm 2: GreedyTour

Figure 2 shows the SCP algorithm being applied to the example of Figure 1. First we create the metric closure on the terminal nodes, then we create a spanning tree from that (Figure 2(a)). We then extract the corresponding Steiner tree from the connectivity graph (b). We extract the new locations (c), construct the metric closure for those vertices (d), and then create the tour (e), which in this case requires the agent to backtrack out from vertices D and K. This requires 4 new nodes in total, with a mobility cost of 36.

The Integrated Path IP Algorithm

The SCP algorithm prioritises the number of nodes over the mobility cost. Our second approach attempts to combine the two objectives, adding approximate mobility costs into the connectivity graph, and then searching for a minimal Steiner tree on this modified graph (Algorithm 3). First, for each edge in the connectivity graph we compute the shortest path between the vertices in the mobility graph using Dijkstra's algorithm, and add that as an edge cost to the connectivity graph. We then apply the SCP algorithm on this new problem, with the difference being in the step where we create the Steiner tree from the minimum spanning tree of the metric closure, since the edges in the connectivity graph now have non-uniform costs. The runtime is dominated by

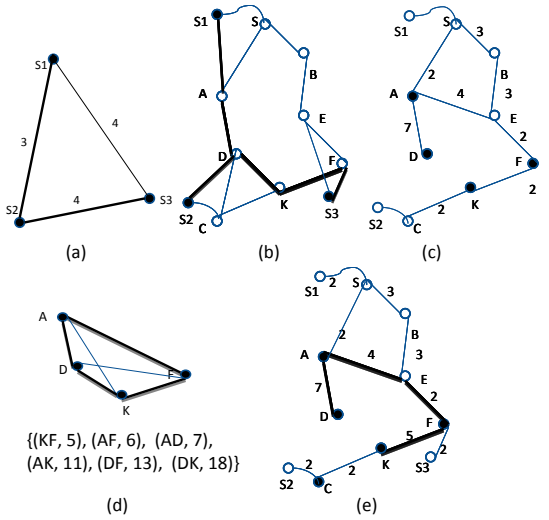


Figure 2: A sample execution of Shortest Cheapest Path

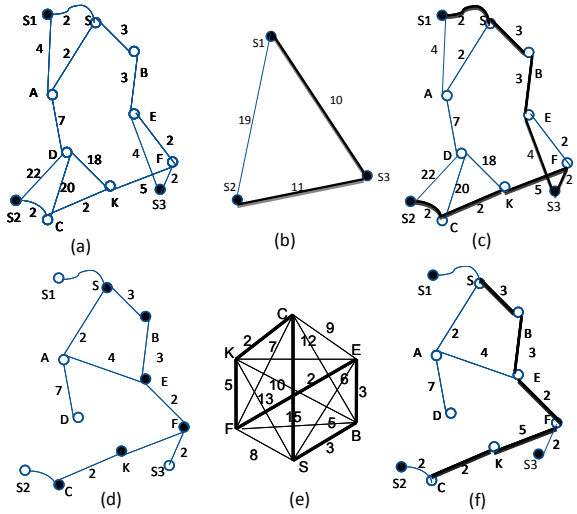


Figure 3: A sample execution of Integrated Path

the cost of building the initial weighted connectivity graph, $O(|E_c||V|^2)$.

Data: A graph $G_C = (V, E_C)$, graph $G_M = (V, E_M)$, an edge cost function c for G_M, a set of terminals $\tau \subset V$.
Result: Number of nodes placed and a tour in G_M visiting those nodes.
begin
  $G_C* = (V, E_C, w*) = w\_conn\_graph(G_C, G_M)$;
  $T = (V', E') = Steiner\_MST(G_C*)$;
  $P = Greedy\_TSP(V' - \tau, G_M)$;
  return [$V' - \tau$,P] ;
end

Algorithm 3: Integrated Path Algorithm

Figure 3 shows the IP algorithm being applied to the example of Figure 1. First we create the weighted connectivity graph (Figure 3(a)), then metric closure on the terminal nodes, followed by its spanning tree from (b). We then extract the corresponding Steiner tree from the weighted connectivity graph (c). We extract the new locations (d), construct the metric closure for those vertices (e), and then create the tour (f). This requires 6 new nodes in total, with a mobility cost of 30.

Experiments

Both algorithms presented above are heuristic, and take different approaches to the multi-objective problem. Therefore, we evaluate them empirically on randomly generated graphs, to compare the quality of their solutions on both objectives, and also on their runtime. For the graphs, our aim is to represent a physical area rather than abstract random graphs, and so we overlay the graphs on a rectangular grid. Connectivity is based on the distance between two locations. To represent a landscape or a building interior, we add obstacles into the grid, which may hinder or forbid access. The mobility graph is then based on line-of-sight, with some limited ability to cross the obstacles.

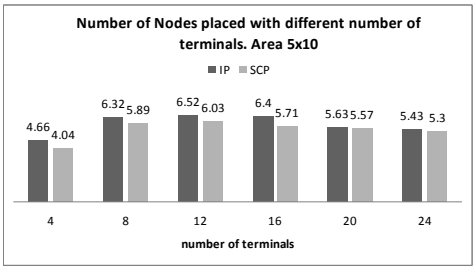
We generate graphs within a rectangular area consisting of $n \times m$ squares of size 10 units. Within this space, we place k mobility obstacles, where each obstacle is a random polygon contained within a randomly selected pair of neighbouring cells. For each square, we generate a random position within it; if that position is inside an obstacle, we discard it, otherwise we designate it as a candidate location. Each obstacle is given a random weight w between 0 and 1, representing the difficulty it creates for the agent to traverse it, and such that any obstacle with a weight greater than 0.2 is assumed to be not able to be traversed. We consider two different problem sizes: (i) a 5 $\times$ 10 grid, and thus a maximum of 50 candidate locations, and 15 obstacles, and (ii) a 10 $\times$ 10 grid, and thus a maximum of 100 candidate locations, and 28 obstacles.

We then create the connectivity graph by adding edges indicating that two candidate locations are within transmission range. For each pair of locations, if they are within 10 units apart, we add an edge with probability 0.85; if they are between 10 and 20 units apart, we add an edge with probability 0.2. For the mobility graph, we add an edge between any pair of locations which are less than 25 units apart and which can be connected by a straight line that does not cross an obstacle. The weight of the edge is simply the length of the connecting line. In order to create more dense graphs with varied mobility costs, we then add extra edges. For each pair of nodes separated by a distance of less than 45 we select it with a probability of 0.2; if the straight line between them traverses any obstacle with a weight greater than 0.2, we discard it, otherwise the cost of the edge is the distance plus 10*weight for each obstacle it crosses.

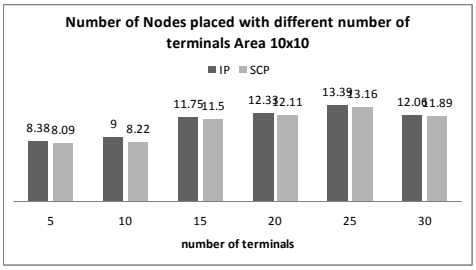
For each of the two problem sizes, for each data point, we generate 50 instances, and present the average solution cost (mobility cost, number of nodes needed) and runtime.

For each instance, we randomly select candidate locations as terminals.

Figure 4 shows the number of nodes placed by each algorithm, as we vary the number of terminals. As the number of terminals to be connected increases, the number of required nodes initially rises to a peak when 25% of the locations are terminals, and then starts to decline: as we add more terminals we require more nodes until the graph becomes sufficiently dense that we can start to re-use the nodes to connect multiple terminals. The IP algorithm requires more nodes, although on average no more than one extra node.



(a) Area 5x10

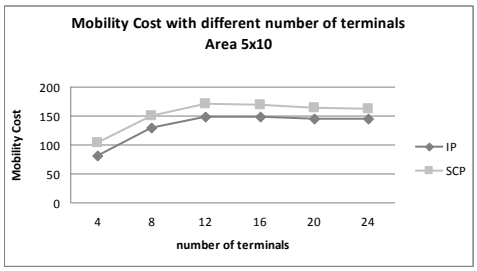


(b) Area 10x10.

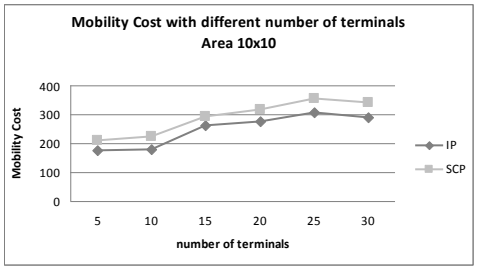
Figure 4: Number of nodes placed with varied number of terminals.

Figure 5 shows the mobility costs as we vary the number of terminals. Again, the costs rise as we increase the number of terminals to approximately 25% of the locations, and then start to decline. IP always produces solutions with lower mobility cost than SCP, ranging from a 10% to a 21% improvement. Figure 6 shows the runtime for the two algorithms. The SCP algorithm is significantly faster, with a speedup factor between 10 and 30.

To decide which algorithm should be selected will require greater knowledge of the relative cost of the new radio nodes versus the mobility costs. We should also take into account the runtime of the algorithm, since as the graphs grow in size, the runtime will become more significant: waiting for the algorithm to complete may remove any benefit gained from a shorter path. If we regard the mobility cost as the time to traverse the circuit, we can then consider the total time for restoring the network as being the runtime of the algorithm plus travelling time of the agent, plus any time



(a) Area 5x10

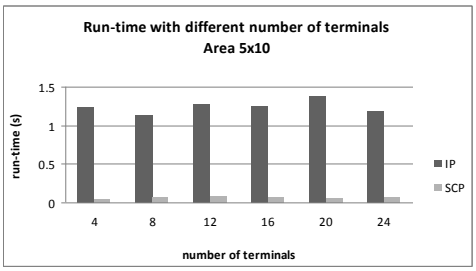


(b) Area 10x10.

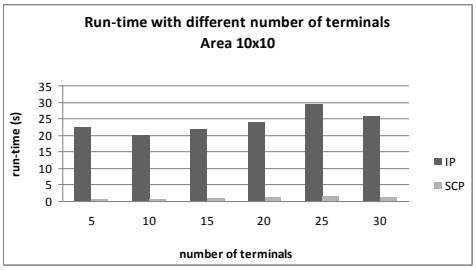
Figure 5: Mobility Costs with varied number of terminals.

required to place the nodes in position. If we assume that the agent is a small robot, then a typical speed might be 0.1ms^{-1}, while the time to place a node might be 30 seconds. Based on the results above, for a 25 terminal problem in the 10×10 grid, where 1 unit is 1m, the total time to repair the network will be 300/0.1 + 13.4*30 +21 = 3423 seconds for the IP algorithm, while it will be 350/0.1 + 13.16*30 + 1 = 3896 seconds for the SCP algorithm. In this case, the IP algorithm will reduce the total repair time by 473 seconds. This reduced time may be significant for network repair during an emergency. For faster moving agents (e.g. a motor vehicle over rough terrain), if we assume an average speed of 4ms^{-1}, the total repair time for IP is 300/4 + 13.4*30 + 21 = 498, while for SCP the total repair time is 350/4 + 3.16*30 + 1 = 483, and thus SCP is slightly faster. For a given problem size, as the speed of the agent increases, we are more likely to prefer SCP.

We also note that the number of terminals for a fixed size problem does not have a significant impact on the runtime, but that there is a significant difference as we increase the network size. Therefore, we perform another experiment in which we vary the density of nodes and edges. More specifically, we vary the maximum number of nodes able to be placed in a single grid square from 1 to 4. During generation, for each square, we randomly select the number of nodes to be placed, and then select their positions. As we increase this parameter, the number of edges increases significantly, and particularly so for the mobility graph. The results are shown in Figure 7, where the x-axis again shows different number

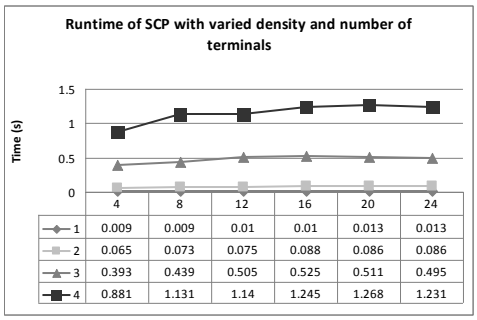


(a) Area 5x10



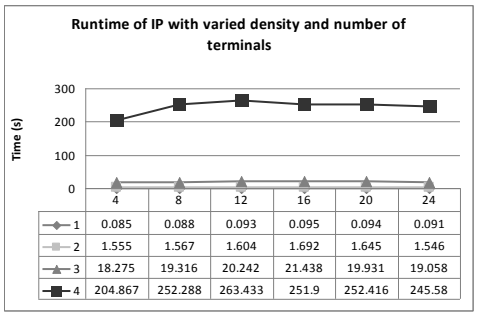
(b) Area 10x10.

Figure 6: Runtime with varied number of terminals.



	4	8	12	16	20	24
1	0.009	0.009	0.01	0.01	0.013	0.013
2	0.065	0.073	0.075	0.088	0.086	0.086
3	0.393	0.439	0.505	0.525	0.511	0.495
4	0.881	1.131	1.14	1.245	1.268	1.231

(a) SCP



	4	8	12	16	20	24
1	0.085	0.088	0.093	0.095	0.094	0.091
2	1.555	1.567	1.604	1.692	1.645	1.546
3	18.275	19.316	20.242	21.438	19.931	19.058
4	204.867	252.288	263.433	251.9	252.416	245.58

(b) IP

Figure 7: Runtime with varied density and number of terminals.

of terminals. For the SCP algorithm the runtime appears to scale with the square of the density parameter. For the IP algorithm, the runtime appears to grow exponentially. Note that the density parameter relates to the density of the nodes packed into the same geographic area, and so a higher value involves more nodes as well has a higher average degree for each node in the graph. The SCP algorithm complexity is the product of the square of the number of nodes and the size of the Steiner tree. As more nodes are added, the graph density increases, and we expect the Steiner tree to grow sublinearly. However, the complexity of the IP algorithm is the product of the number of edges and the square of the number of nodes, and so increasing the density parameter should have a more significant effect, as indicated by the increasing runtime. In Figure 8 we examine the results of density parameter = 4 in more detail. Although the IP algorithm saves about 20% of the mobility costs, it is two orders of magnitude slower in runtime, and requires slightly more nodes on average.

Table 1 gives a summary of the conditions under which we would prefer one algorithm over the other. For high node costs, dense graphs, or fast moving agents, we expect to prefer the SCP algorithm, while for cases where energy costs are significant, or where agents are relatively slow, then the IP algorithm will be preferred.

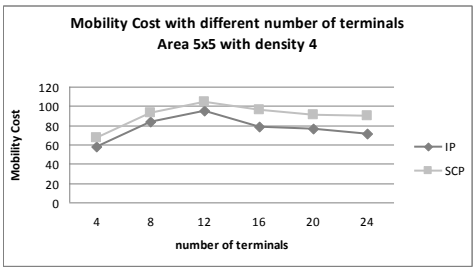
Algorithm	IP	SCP
Energy (Movement)	y	n
Total restoring time with low-medium speed	y	n
Total restoring time with high speed	y	y
Expensive node	n	y
Very dense network ($d \approx 4$)	n	y

Table 1: IP or SCP.

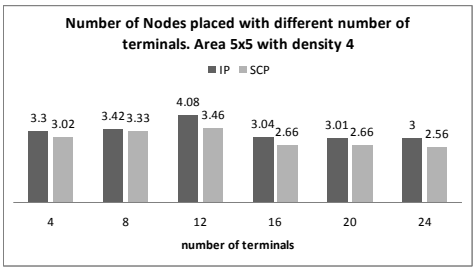
Related Work

The subject of network restoration for wireless sensor networks has been an active area of research. The different approaches can be classified as (i) deploying redundant nodes so as to be able to cope with a pre-determined number of failures, (ii) use of mobile (actor) nodes that can be moved into position in order to restore connectivity, (iii) dispatching mobile nodes in a pre-emptive manner to avoid failures in connectivity, and (iv) the deployment of additional nodes to restore connectivity after failures have occurred. The latter work is closest to our research but is different in two respects, firstly in that we optimise both the number of additional nodes as well as the path length needed for their deployment, and secondly in that we explicitly take into account the impact of obstacles that can alter both the available paths and the ability of nodes to communicate directly. We now provide a summary of the related papers.

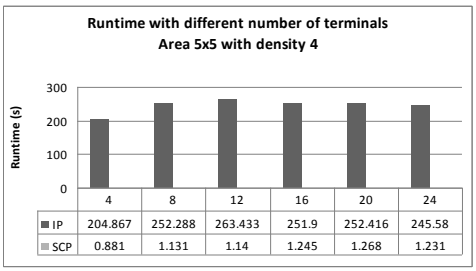
In (Khelifa et al. 2009),(Almasaeid and Kamal 2009) the goal is to deploy k-1 redundant nodes with the intention of



(a) Mobility Cost



(b) Placing Nodes



(c) Runtime

Figure 8: Runtime with density 4 and varied number of terminals.

achieving k-connectivity, for example by placing nodes at the intersection between the communication range of each pair of nodes. The number of additional nodes required by these approaches is prohibitive.

Several papers consider the use of mobile actor nodes in network restoration, e.g. (Abbasi, Akkaya, and Younis 2007),(Akkaya and Senel 2009),(Abbasi, Younis, and Baroudi 2010),(Akkaya et al. 2010),(Sir et al. 2011). The papers propose different strategies to choose the moving actors, for example, based on estimating the shortest moving distance and/or degree of connectivity. The solution space is quite limited, focused on dealing with a single failure and re-connecting just two networks at a time, and with an assumption that mobility is unimpeded by obstacles.

(Dai and Chan 2007) proactively deploys additional helper mobile nodes, controlling their trajectories in response to predicted network disconnection events. The work assumes that the mobile nodes are always fast enough to reach the desired destination in case of a predicted disconnection event, and that a full map of the physical terrain and radio environment is available. Details of how to determine the number of mobile nodes that are needed and the related path planning are not provided.

(Senel, Younis, and Akkaya 2009) and (Lee and Younis 2010) assume multiple simultaneous failures involving many failed nodes and a network that is partitioned into many segments. The approach is to re-connect those segments in a centralized manner with the main objective of using the smallest number of additional nodes. (Senel, Younis, and Akkaya 2009) uses a spider web approach to reconnect the segments while (Lee and Younis 2010) forms a connectivity chain from each segment toward a centre point and then seeks to optimize the number of additional nodes that are needed.

Conclusion

We have defined the new problem of simultaneous network connectivity restoration with constrained route planning, in the presence of obstacles. We formalise the problem as a multi-objective problem of minimising a Steiner tree in a connectivity graph and minimising a tour of the nodes in that tree in a mobility graph. We present two heuristic algorithms, Shortest Cheapest Path and Integrated Path. Shortest Cheapest Path prioritises node cost, and first finds the minimum number of nodes required to heal the network; it then finds the cheapest path which visits all the new node positions. Integrated Path combines the two objectives by adding weights into the connectivity graph to approximate the mobility cost of establishing each link, and then searches for the cheapest tree that connects all existing nodes. We conducted an empirical evaluation of the two algorithms on random connectivity and mobility graphs. The SCP algorithm tends to find graphs with fewer nodes, while the IP algorithm finds slightly larger solutions but with cheaper mobility costs. The SCP algorithm is significantly faster, particularly on dense graphs.

Immediate future work will involve developing a hierarchical routing approach, which will allow us to handle dense mobility graphs, by initially merging adjacent location nodes into a supernode to reduce the complexity. Longer term work will focus on expanding the problem to include the simultaneous exploration of network connectivity and mobility constraints as we optimise the node selection and placement. This will require heuristic algorithms that tradeoff the cost of further exploration against the cost of placing nodes early. We will also consider distributed algorithms, allowing multiple agents and sensor nodes to collaborate to determine the damage to the network. Finally, we aim to tackle problems where the damage is continually changing, and thus the solutions need to be continually regenerated.

Acknowledgment

This work is part of the NEMBES project, which is funded by the Irish Higher Education Authority PRTLIIV programme.

References

Abbasi, A. A.; Akkaya, K.; and Younis, M. 2007. A distributed connectivity restoration algorithm in wireless sensor and actor networks. In *Proceedings of the 32nd IEEE Conference on Local Computer Networks*, 496–503. Washington, DC, USA: IEEE Computer Society.

Abbasi, A.; Younis, M.; and Baroudi, U. 2010. Restoring connectivity in wireless sensor-actor networks with minimal topology changes. In *Communications (ICC), 2010 IEEE International Conference on*, 1 –5.

Akkaya, K., and Senel, F. 2009. Detecting and connecting disjoint sub-networks in wireless sensor and actor networks. *Ad Hoc Netw.* 7:1330–1346.

Akkaya, K.; Senel, F.; Thimmapuram, A.; and Uludag, S. 2010. Distributed recovery from network partitioning in movable sensor/actor networks via controlled mobility. *IEEE Trans. Comput.* 59:258–271.

Almasaeid, H. M., and Kamal, A. E. 2009. On the minimum k-connectivity repair in wireless sensor networks. In *ICC*, 1–5. IEEE.

Dai, L., and Chan, V. 2007. Helper node trajectory control for connection assurance in proactive mobile wireless networks. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, 882 –887.

Khelifa, B.; Haffaf, H.; Merabti, M.; and Llewellyn-Jones, D. 2009. Monitoring connectivity in wireless sensor networks. In *ISCC*, 507–512. IEEE.

Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; and Shmoys, D. B. 1985. The traveling salesman problem.

Lee, S., and Younis, M. 2010. Recovery from multiple simultaneous failures in wireless sensor networks using minimum steiner tree. *J. Parallel Distrib. Comput.* 70:525–536.

Senel, F.; Younis, M.; and Akkaya, K. 2009. A robust relay node placement heuristic for structurally damaged wireless sensor networks. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, 633 –640.

Sir, M.; Senturk, I.; Sisikoglu, E.; and Akkaya, K. 2011. An optimization-based approach for connecting partitioned mobile sensor/actuator networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, 525 –530.

Wu, B. Y., and Chao, K.-M. 2004. *Spanning Trees and Optimization Problems*. USA: Chapman & Hall / CRC Press.

On Finding and Exploiting Mutual Exclusion in Domain Independent Planning

Filip Dvořák*, Daniel Toropila*†

*Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Prague 1, Czech Republic

†Charles University in Prague, Computer Science Center
Ovocný trh 5, 116 36 Prague 1, Czech Republic

{filip.dvorak, daniel.toropila}@mff.cuni.cz

Abstract

Planning is inherently hard; therefore it is desirable to derive as much information as we can from the structure of the planning problem and let the information be exploited by a planner. In this paper we focus on the relation of the mutual exclusion between atom pairs and pair-wise mutual exclusion in sets of atoms (mutex sets). We shall present our ideas on both the generation of the mutex sets and how we can use the generated mutex sets for enriching the problem formulation.

Introduction

The task of planning is to find a sequence of actions that transfer the world from some initial state to a state satisfying certain goal condition. Since planning is hard (Ero et al. 1995), it is beneficial to develop techniques for structural exploration of the planning problems. While some exploration techniques can be applied only during the search, others can be computed in advance and may even encourage reformulation of the problem into a form that directly encodes the information they brought. Very useful family of structural information are the problem invariants, among which we today find being the most helpful the landmarks (Hoffman et al. 2004) and mutexes.

In context of planning, mutual exclusion between two entities says that they interfere with each other and cannot occur together in some context. Given a planning problem, if two facts cannot occur at the same time point during the execution of any valid plan, we say they are (general) mutex. Unfortunately, deciding whether two facts are mutex is as hard as the planning itself.

Our approach can be devided into three steps. We use the Planning Graph structure (Blum and Furst 1997) for grounding and capturing the mutex relations, then we generate the sets of pair-wise mutually exclusive atoms and finally, we reformulate the planning problem into the SAS^+ formalism (Backstrom and Nebel 1995) using both greedy covering of the mutex sets and a new approach, over-covering of the mutex sets. In the following sections we describe these steps in detail.

Grounding

For the purpose of grounding, we have decided to adapt the planning graph structure. Planning graph originally takes a grounded planning problem as an input; hence we have relaxed the planning graph into a *lazy planning graph* that takes ungrounded problem and incrementally instantiates the operators, but only when the actions can actually occur in the plan.

Based on the latest International Planning Competition, we have observed that the current dominant approach to grounding is based on the delete-relaxation (Helmert 2006). With regard to the size of the grounded problem, the grounding through planning graph is always at least as good as grounding using delete relaxation (we use the delete relaxation, but constrain the grounding even further by using mutexes). We construct the planning graph using a standard algorithm given in (Ghallabet al. 2004).

We expect a significant improvement in the size of the grounded problem in the domains with *destructive actions*. We say that an action is destructive, if it transfers the world into a state from which the previous state becomes unreachable. An example of a destructive action can be an action eliminating a chess figure, which prevents the configuration of the chessboard from ever being the same. We say that a domain is destructive, if it contains some destructive actions. On the other hand, an example of a domain that is not destructive is the problem of solving the Rubik cube, where all actions can be reversed.

The planning graph captures the mutex relations between atoms for each layer. The most interesting for us is the final layer of the planning graph that contains all the general mutex relations between atoms it had discovered. We further use the gathered relations for building the mutex sets.

Mutex sets

By our observation, the dominant approach in the field of finding mutex sets is to use only the information contained in the domain of the problem, disregarding the actual instance of the problem (assuming the PDDL formulation). The planning graph however provides us with a richer set of relations that we can build the mutex sets from. For example, we can imagine a problem with one driver D, one car C and two locations loc1 and loc2. Using only the domain information, we find that {at(D,loc1), at(D,loc2)} and {at(C,loc1), at(C,loc2)} are mutex. However the planning graph also reveals the mutex {at(C,loc1), at(D,loc2)} and {at(C,loc2), at(D,loc1)}, since driver and car cannot be at different locations at once, because car must be driven by a driver (assuming that in the initial state car and driver are at the same location).

Given a set of mutex relations we can build a *mutex graph*, where vertices correspond to atoms and edges represent mutexes between the atoms. Finding all the sets of pair-wise mutex atoms is a variation of the NP-complete Maximal Clique problem. While for some small domains such as *pegsol* we can enumerate all the sets, for example the logistic domains tend to generate unfeasible number of mutex sets. To deal with the complexity we employ a probabilistic algorithm Cover (Richter et al. 2007) that we limit by time. Therefore instead of enumerating all the mutex sets we harvest only the most interesting ones.

Once we have a set of mutex sets, we move to encoding the information they provide using the SAS$^+$ formalism.

Translation

The key benefit of the SAS$^+$ formalism is capturing the information provided by the mutex sets. Instead of defining the state of the world as an assignment of truth values to all the atoms in the planning problem, we create a *state variable* for each mutex set and define the state of the world as an assignment of values to all the state variables. The value assigned to a state variable represents the fact that the corresponding atom is set to true and all other atoms represented by that state variable are false. The resulting state space of the planning problem is therefore significantly reduced (assuming there exist mutex sets larger than one).

Since we need to able to represent all the atoms of the original problem, the union of mutex sets needs to cover all the atoms. The usual expectation also is that every atom is covered by exactly one mutex set. Therefore given a set of mutex sets we need to find a subset that covers all the atoms, while the smaller the number of mutex sets we choose, the smaller state space we get. Alas, finding the minimal subset of mutex sets is a well-known NP-hard problem, Set Cover. Thus we employ a greedy algorithm that at each step chooses the set that contains the largest number of atoms that were not yet covered. The greedy algorithm is known to be the best-possible polynomial time approximation algorithm for the Set Cover problem.

While we expect that we shall be able to generate fewer state variables having a richer set of mutex sets, we do not stop here but try to encode more mutex information.

As a motivation example, we can imagine mutex sets {A,B,C}, {B,C,D} and {C,D,E}. A greedy covering would result in sets {A,B,C} and {D,E}, however the information given by {B,C,D} would be lost. Our idea is to relax the translation and allow the atoms to be covered by more than one state variable, which shall allow us to build more informed representations with regard to the mutual exclusion. We call this idea an *over-covering* of mutex sets. The practical effect should be an increase of information the problem representation carries, while the size of the problem shall increase only little, since every over-covered atom needs to be represented by multiple effects and preconditions in each action (for each state variable that is covering that atom).

Conclusions

We believe there is a space for improvement in ways how the mutex relations are used; hence we have proposed several ideas that may turn beneficial for the planners. Our next step is to evaluate the performance of planners using our translation.

Acknowledgement

This research is supported by the Grant Agency of Charles University as the project no. 306011 and 9710/2011.

References

Backstrom, C. and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence 11*, 625-655.

Blum, A. and Furst, M. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence vol 90*, 281-300.

Ero, K.; Nau, D. S. and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence, vol 76.*, 65-88.

Ghallab, M.; Nau, D. and Traverso, P. 2004. Automated Planning: Theory and Practice. Morgan Kaufmann Publishers.

Helmert, M. 2006. Solving Planning Tasks in Theory and Practice. Albert-Ludwigs Universitat Freiburg Doctoral thesis.

Hoffman, J.; Porteous, J. and Sebastia, L. 2004. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research 22*, 215-278.

Richter, S.; Helmert, M. and Gretton, C. 2007. A Stochastic Local Search Approach to Vertex Cover. *Proceedings of the 30th German Conference on Artificial Intelligence*, 412-426.

Towards Learning Operator Schema from Free Text

A.Fanan and T.L.McCluskey,
Department of Informatics,
School of Computing and Engineering,
University of Huddersfield

Abstract

In automated planning current research is focused on developing *domain-independent* planning engines. These require domain models, written in a standard input language such as PDDL to supply knowledge of the planning application and task, before they can be used. The main component of a domain model is the representation of actions in the form of lifted operator schema. The acquisition and engineering of these schema is an important area of research, as this process is recognised as being difficult and laborious even for planning experts.

A fruitful line of research is to investigate mechanisms to automatically learn planning domain models. Recent research has studied learning from structured or refined inputs supplied by a training agent (Cresswell, McCluskey, and West 2011; Zhuo et al. 2010; Wu, Yang, and Jiang 2005; McCluskey et al. 2010). An alternative method would be to allow planning agents to learn and develop the domain models by observation. One freely available source for learning actions is selected web text; here actions are represented as verbs in natural language. This project aims to investigate the possibility of extracting formal structures representing actions from free text. We intend to utilise large text corpuses available on-line from which to extract such action knowledge, and learn operator schema in a formal language that can be converted to PDDL.

A. Introduction

Over the last decade there has been some progress in producing tools to help in the acquisition of domain models for planning. These fall into three areas:

- *knowledge engineering tools for supporting experts:* this area covers user interface tools which help users in domain formulation, domain analysis, and planning simulation. GIPO (Simpson, Kitchin, and McCluskey 2007) provides a diagrammatic interface for the user in order to avoid the need to write operator schema explicitly. From user drawn state machines and annotations representing the domain, the system generates a formulation automatically, relieving the user of the burden of writing detailed parameters, logical expressions or predicates. GIPO is analogous in functionality to visualisation tools that perform code generation in software engineering environments. itSimple (Integrated Tools Software Interface for Modelling Planning Environment) (Vaquero et al. 2009) is a similar tool, but closer to the software engineering community than GIPO with its emphasis on the use of UML. Unlike GIPO, itSimple continues to be maintained and upgraded. The latest version of itSimple is 3.5 featuring an integrated environment with representation languages such as XML, Petri Nets and PDDL.
- *creating a planning domain model via translation:* There are situations where a formal model of a domain already exists, for instance in the areas of business modelling, workflow, or web services, and therefore the opportunity exists to create translation tools that when input with a model described in an application area specific language, output a full or partial planning domain model. The 2009 run of ICKEPS focused on this specific aspect of knowledge engineering technology, clearly showing ways that planning engines can be used as "black boxes" within environments that assemble the inputs to the planner automatically, and exploit the output plan in an application-dependent way (Bartak, Fratini, and McCluskey 2010).
- *tools that learn the domain model from engineered examples and partial domain models:* this includes tools that build models automatically from plan traces and background knowledge which may lead to reduction in timescales and effort over handcrafting. Several systems that learn and refine domain models from examples have been developed in recent years. For example, the systems ARMS (Action Relation Modelling System), LOCM (Learning Object-Centred Models) and LAMP (Learning Action Models from Plan traces) were all built specifically to support the learning of domain models from many examples of plan traces.

Our research is related to the last area, and we explore current systems in a little more detail. Wu et al (WU, YANG, and JIANG 2007) have presented the process of ARMS in the following phases: it converts

all action instances to their schema forms and finds the frequent predicate-action and action sets from the converted plans that share the same object types. It then transforms the frequent sets into a weighted representation to be input to a SAT solver, and synthesises operator schemas from the results. Systems such as ARMS need to input other information as well as example plan traces before learning can be effective. This can include predicates, initial, goal or intermediate state descriptions, target action names, or other domain information.

At the extreme of these systems is LOCM (Cresswell, McCluskey, and West 2011), an inductive tool which automatically induces a domain model from a set of training examples of plans without the need for any background information. The training input to LOCM (sets of valid action sequences) does have to satisfy certain constraints, however: each action is specified as a name followed by a sequence of affected objects, each instance of a named action has parameters in a consistent order as well as some assumptions on the inductive learning process. Quite strict assumptions are made of the output model also: objects are assumed to belong to a "sort" which, via a state machine, defines the identical behaviour of each object in the sort.

There are other systems that utilise other types of learning, but they too still need well engineered input. For example, LAWS (H.H.Zhuo, Q.Yang, R.Pan and L.Li 2011) learns using analogy: it inputs an existing domain model and uses it to synthesize a new domain model. It still requires other information, such as target action and predicate templates. Hence, common to all of these tools is the necessity for a trainer or teacher to prepare the input. These inputs in some cases have to be refined or engineered themselves before the tools can learn effectively. Invariably there are also many assumptions made on the form of the output model.

Research Programme

In contrast to the work above, we hypothesise that it is possible to learn the main part of a domain model, that is the operator schema, by observation alone, without the need for a trainer or specifically engineered background knowledge. Recently, in the broader area of AI, techniques are being developed to systemize the learning of action knowledge from free text such as extracting script or narrative event schemas (Chambers and Jurafsky 2009).

Closest to our proposal is the work of Sil and Yates: they have implemented a system called PREPOST that works in identifying the preconditions and the effects of actions and events (Sil, Huang, and Yates 2010). PREPOST is a text mining system that involves two distinct learned classifiers for both preconditions and postconditions. The PREPOST technique is dependent on the ability of the search engine to find a collection of documents that contains a specific word that should be a verb or an event. Therefore, PREPOST uses an English language progressive form of verbs and is used almost exclusively with events and action verbs. PREPOST uses a search engine to find a huge collection of documents for a selected word (X) by considering the pattern "is/are/were/was + X+ing", then uses techniques from the field of inductive learning to identify preconditions and effects. The system is supported by other techniques such as semantic role labelling and has been used with some success to identify predicates and event conditions in text. Recently, Sil and Yates have improved upon their previous work by presenting a system for extracting a full-formed STRIPS representation of actions (Sil and Yates 2011). This system has demonstrated that it has the ability to identify the preconditions of previously unseen actions using web documents that are automatically downloaded. Under certain constraints, the precision of the system in recognising preconditions is high.

The research programme that we have embarked on aims to investigate the feasibility of utilising techniques in text mining and natural language processing to identify the characteristics of actions and events from free text, and extract pre- and post conditions within a formal language in a similar manner to PREPOST. We plan, however, to add to the former work constraints and ideas from domain model learning systems such as LAWS and ARMS, in order to leverage more knowledge to improve the system's precision. The use of analogical learning, for instance, which draws on a database of existing models, may be able to provide a more accurate action template for a new verb that is semantically close to one that has previously been encoded in the database.

The long term aim is to embed inside virtual agents the potential to acquire and maintain their own domain theories. This entails encouraging more work into the under-developed field of *learning planning domain models by observation.* Work to date has been aimed at rationally re-constructing the PREPOST system, and connecting such efforts with work on domain model learning and knowledge engineering carried out previously in the AI Planning community. A major research question is: *Is it feasible to utilise within an action learning algorithm a combination of:*

- *text mining tools,*
- *natural language semantic tools such as WordNet,*
- *assumptions and constraints about actions*

in order to induce accurate STRIPS models? In addition to investigating the case with free text, we are considering tackling this question in restricted domains where the vocabulary is limited or controlled, and the a set of actions is connected up, as might be the case in instruction manuals.

References

Bartak, R.; Fratini, S.; and McCluskey, L. 2010. The third competition on knowledge engineering for planning and scheduling. AI Magazine, Spring 2010.

Chambers, N., and Jurafsky, D. 2009. Unsupervised learning of narrative schemas and their participants. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2 - Volume 2*, ACL '09, 602–610. Stroudsburg, PA, USA: Association for Computational Linguistics.

Cresswell, S.; McCluskey, T.; and West, M. M. 2011. Acquiring planning domain models using LOCM. *Knowledge Engineering Review (To Appear)*.

H.H.Zhuo, Q.Yang, R.Pan and L.Li. 2011. Cross-Domain Action-Model Acquisition for Planning Via Web Search. In *Proceedings of ICAPS*.

McCluskey, T. L.; Cresswell, S. N.; Richardson, N. E.; and West, M. M. 2010. Action Knowledge Acquisition with Opmaker2. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*. Springer Berlin Heidelberg. 137–150.

Sil, A., and Yates, A. 2011. Extracting STRIPS representations of actions and events. In *Recent Advances in Natural Language Learning (RANLP)*.

Sil, A.; Huang, F.; and Yates, E. 2010. Extracting action and event semantics from web text. AAAI Fall Symposium on Commonsense Knowledge.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *The Knowledge Engineering Review* 22(1).

Vaquero, T.; Silva, J.; Ferreira, M.; and Tonidandel, F. 2009. From requirements and analysis to pddl in itsimple3. 0. *Artificial Intelligence*.

Wu, K.; Yang, Q.; and Jiang, Y. 2005. Arms: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.

WU, K.; YANG, Q.; and JIANG, Y. 2007. Arms: an automatic knowledge engineering tool for learning action models for ai planning. *Knowl. Eng. Rev.* 22:135–152.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence* 174(18):1540–1569.

Planning in Offensive Cyber Operations: a new domain?

Tim Grant

Faculty of Military Sciences, Netherlands Defence Academy (NLDA)
P.O. Box 90.002, 4800PA Breda, The Netherlands
tj.grant@nlda.nl

Abstract

Several countries have recently published new national cyber security policies that allow for the possibility of offensive action in response to a cyber attack. The available literature shows that cyber attacks involve planning. The purpose of this short paper is to identify the characteristics of planning in offensive cyber operations, asking whether this is a new domain.

Background

Governments, business, and individuals are dependent on access to cyberspace. However, cyber crime has grown exponentially over the past five to ten years. Online fraud is estimated to have netted £52 billion worldwide for criminals in 2007 (TSO, 2009). As the Stuxnet attack on Iran's uranium processing facilities showed, cyber attacks could now hold to ransom essential national infrastructure, such as energy, food, water, transport, communications, government, emergency services, health, and finance.
In response, several countries (e.g. USA, UK, Netherlands, France) have recently published new national cyber security policies. Typically, these policies call for:

- Cooperation between government departments, public and private organizations, and international partners.
- The establishment of a National Cyber Security Centre (NCSC) to monitor developments in cyberspace, to analyse trends, to disseminate information, and to coordinate the response to cyber incidents.

Until now, it appears as if the cyber attacker has free rein, while defenders must stoically endure the attack (Owens et al, 2009). For example, when the Internet banking portal of a major Dutch bank was subjected earlier this year to a distributed denial of service (DDoS) attack lasting 29 hours, the victims could only wait until the attackers had tired of their activities. The resulting loss of turnover was estimated at €8 million (Volkskrant, 2011). Hence, there are strong commercial arguments for acquiring the capability to shorten the duration of such attacks by counter-attacking or, better still, by pre-empting an impending attack ("pro-active defence"). Notably, the new policies do not exclude the NCSCs from doing so.
There are many difficulties associated with acquiring offensive capabilities, not least the legal issues (Ducheine & Voetelink, 2011). Assuming these are resolved, there is then the need for the NCSC to acquire the appropriate tools and to recruit and train personnel in their use. Above all, the NCSC will need to acquire the knowledge of how to perform a cyber attack within the prevailing legal constraints. Outside cyberspace, such capabilities are traditionally provided by military forces, making it likely that Ministries of Defence will play a similar role within NCSCs. While such capabilities may have been fielded by USA, Israel, China, Russia, and Germany, other countries (e.g. UK and Netherlands) have yet to acquire them.

Offensive Cyber Operations

The US Department of Defense's Dictionary of Military Terms and Abbreviations (JP1-02, 2011) defines cyberspace as "*a global domain within the information environment consisting of the interdependent network of information technology infrastructures, including the Internet, telecommunications networks, computer systems, and embedded processors and controllers*". Cyber operations are "*the employment of cyber capabilities where the primary purpose is to achieve objectives in or through cyberspace*", and are divided into computer network attack (CNA), defence (CND), and exploitation (CNE).
Offensive cyber operations focuses on CNA, which are actions taken "*through the use of computer networks to disrupt, deny, degrade, or destroy information resident in computers and computer networks, or the computers and networks themselves*". To do this, it is necessary to know where the information resides and how the computers and networks are configured. This prior knowledge comes from CNE, which are the "*enabling operations and intelligence collection capabilities conducted through the use of computer networks to gather data from target or adversary automated information systems or networks*".
The open scientific literature overwhelmingly concerns defensive measures. Lin (2009) and Denning and Denning (2010) have recently argued that offensive cyber operations deserve more extensive scientific discussion.

Planning a Cyber Attack

There is some literature describing the attack process. Authors differ on whether this consists of three phases (Jonsson & Olovsson, 1997) or nine (Grant et al, 2007), or some number in between. However, they all describe it as a repeatable, linear sequence of steps, i.e. a skeletal plan. Intelligence gathering (i.e. CNE) always precedes the attack proper (i.e. CNA). The skeletal plan must be detailed, e.g. to take account of the target network

topology, the specific hardware and software components found, their vulnerabilities, the available paths for gaining access to those vulnerabilities, and the arsenal of malware the attacker possesses (Owens et al, 2009). Hence, cyber attack requires a planning step between CNE and CNA.
Owens et al (2009) compare planning a cyber attack with planning traditional military operations, concluding that cyber is more complex. Firstly, they observe that there are additional uncertainties, including those that result from unanticipated interactions between the target and surrounding systems. Secondly, they note that planning for cyber attack involves a larger range of choices. Timescales can range from seconds (e.g. when interfering with the timing of a real-time control system) to years (e.g. when implanting "sleeper" functionality to be activated at some future date). Target systems may be dispersed around the globe or concentrated in a facility next door. Thirdly, the desired effects of a cyber attack are almost always indirect. For example, the desired effects of Stuxnet were physical (i.e. the self-destruction of Iranian uranium centrifuges), rather than cyber. Fourthly, the causal chain is tied into human perceptions and decisions, e.g. how the defenders perceive and respond to the cyber effects. Finally, complex execution plans have many ways to go wrong. Owens et al suggest that one way of coping with failure-prone plans is to monitor execution, to obtain feedback on intermediate results, and then to generate and apply "mid-course corrections" as necessary. This requires the attacker to interleave planning with execution in real time.

Implications for AI Planning

Is planning in offensive cyber operations a new domain? In the AI literature, cyber attack plans have been represented as trees (Schneier, 1999) or graphs (Shyner, 2004). These representations help analyse attack scenarios by enumerating the target system's vulnerabilities and by capturing the relationships between the system's configuration and its vulnerabilities. Nodes in the tree or graph represent possible system states during the course of the attack, with the root node representing the attacker's goal. Edges represent the attacker's actions.
Attack graphs suffer from state space explosion, limiting their scaleability. Several approaches have been explored to solve this problem, including using planning techniques to reduce the attack graph (Ghosh & Ghosh, 2010).
Boddy et al (2005) used classical planning techniques to identify potential vulnerabilities and countermeasures. Some of the attack scenarios that their system generated were new to the analyst. Their planning domain has been used as a benchmark problem in the International Planning Competition. Roberts et al (2011) extended Boddy et al's domain to represent user behavior, arguing that the user may well pose a greater security risk than an adversary.
Superficially, the attack planning domain in the existing AI literature appears to be the same as planning in offensive cyber operations. However, this is misleading because the existing domains are defensive, being aimed at identifying vulnerabilities and countermeasures in the target system. They do not fully represent the attacker in the wider environment. For example, existing domains fail to model collateral damage resulting from the attacker's incomplete collection of intelligence or lack of anticipation of the interactions between the target system and its surroundings. They do not allow for a wide range of timescales or for desired effects that are non-cyber or indirect. Above all, the existing domains do not adequately close the feedback loop between attackers and defenders, where an action by the attacker leads to a response by the defender that requires the attacker to change his/her plan.
We conclude that offensive cyber operations is a new planning domain.
Our next step is to formalise the domain, drawing where possible on existing planning domains. The resulting planning operator set can then be evaluated with an eye towards application in the Dutch NCSC.

References

Boddy, M., Gohde, J., Haigh, T., & Harp, S. 2005. *Course of Action Generation for Cyber Security using Classical Planning*. Procs, ICAPS'05.

Denning, P.J. & Denning, D.E. 2010. *Discussing Cyber Attack*. CACM, 53, 9, 29-31.

Ducheine, P.A.L. & Voetelink, J.E.D. 2011. *Cyberoperaties: naar een juridische raamwerk*. Militaire Spectator, 180, 6, 273-286.

Ghosh, N. & Ghosh, S. 2010. *A planner-based approach to generate and analyze minimal attack graph*. International Journal of Applied Intelligence.

Grant, T.J., Venter, H.S., & Eloff, J.H.P. 2007. *Simulating Adversarial Interactions between Intruders and System Administrators using OODA-RR*. Procs, SAICSIT'07.

Jonsson, E. & Olovsson, T. 1997. *A quantitative model of the security intrusion process based on attacker behavior*. IEEE Trans on Software Engineering, 23, 4, 235-245.

JP1-02. 2011. *DoD Dictionary of Military and Associated Terms*. Washington DC: US Department of Defense.

Lin, H. 2009. *Lifting the Veil on Cyber Offense*. IEEE Security & Privacy, 7, 4, 15-21.

Owens, W.A., Dam, K.W., & Lin, H.S. (eds.) 2009. *Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities*. Washington DC: The Academies Press.

Roberts, M., Howe, A., Ray, I., Urbanska, M., Byrne, Z.S., & Weidert, J.M. 2011. *Personalized Vulnerability Analysis through Automated Planning*. Procs, AAAI'11.

Schneier, B. 1999. *Attack trees*. Dr. Dobb's Journal.

Shyner, O.M. 2004. *Scenario Graphs and Attack Graphs*. PhD thesis, CMU-CS-04-122, Pittsburgh: Carnegie Mellon University.

TSO. 2009. *Cyber Security Strategy of the United Kingdom*. Cm 7642, Norwich, UK: The Stationery Office.

Volkskrant. 2011. *Webwinkels claimen schade door aanval op Rabobank*. Amsterdam, Netherlands: De Persgroep Publishing (4 May 2011).

On Comparing Planning Domain Models

S. Shoeeb and T.L.McCluskey,
Department of Informatics,
School of Computing and Engineering,
University of Huddersfield

Abstract

Whereas research into the characteristics and properties of AI planning algorithms is over-whelming, a similar science of domain model comparison and analysis is underdeveloped. It has long been acknowledged that there can be a range of different encodings for the same domain, whatever coding language used, but the question of which is the "best" encoding is an open one, and partly dependent on the requirements of the planning application itself. There is a growing need to measure and compare domain models, however, in particular to evaluate learning methods. In this paper we motivate the research by considering the challenges in evaluating domain model learning algorithms. We describe an ongoing doctoral research project which is exploring model classifications for comparing domain models.

Introduction

The importance of the knowledge formulation process to the success of planning applications, and to make planning engines more accessible and open to community use, is now well established. The task of creating the domain models is accepted as difficult and time consuming, requiring an engineering process. This has increased the need to investigate the properties of the end product of this process - the planning domain model. It has long been acknowledged that there can be a range of different encodings for the same domain, whatever coding language used, but the question of which is the "best" encoding is an open one, and partly dependent on the requirements of the planning application itself.

The advances made in automated domain model creation, in particular, have focused effort on such questions as what makes an accurate or efficient domain model. In this area, domain models may be learned from examples (Cresswell, McCluskey, and West 2011; Wu, Yang, and Jiang 2005), learned from one example plus a partial domain model (McCluskey, Richardson, and Simpson 2002; McCluskey et al. 2010) or generated from a formal model using a translation process (Ferrer 2011; Bartak, Fratini, and McCluskey 2010).

Domain analysis techniques can of course be used to inform on the quality of a domain model, and in particular whether the model is self-consistent. They are well developed for AI planning, and can be used to check certain desirable static features such as operator reversibility: that is whether at any state, an operator's application can be reversed by another operator to result in the original state (Wickler 2011). Another use is to analyse the likely efficiency of a model when paired with a state of the art state progression planner. We may be interested in knowing the "landscape" of the key heuristic used in planning, such as investigated in Hoffman's recent research (Hoffmann 2011).

The steady increase in the development of domain model learning tools and the need to extract a healthy domain model from such tools requires the development of measurement techniques to evaluate a domain model. One way to do this is to quantify certain features that can easily be calculated and compared: the number of objects, the number of action instances, the degree of arity of operators names, the degree of arity of predicates and the average number of preconditions and postconditions in each operator.

In our research, we consider that the quality of automatically produced domain models needs to be evaluated in order to provide validation and comparison between competing automated approaches. This is of key significance for researchers attempting to evaluate their research in a rigorous fashion. Of course the quality of a model can be measured in a dynamic way, that is by running the models with a planner, and checking performance and output plans. This is a reasonable approach, but has a major flaw: performance is of course dependent on the planner as well as the model, and the results may be due to a planner rather than a model. Ideally, we would also like to measure and compare domain model in a planner-independent way. In this paper we motivate the reader as to why comparison techniques are important, and make some initial definitions of equivalence between domain models.

Motivating Examples

LOCM is an operator schema learning system (Cresswell, McCluskey, and West 2011) which automatically induces a set of operator schema from sets of example plan scripts. LOCM assumes that each plan is a sequence of ordered actions, and each action is composed

of a name and a list of objects. The system differs from related systems in that it requires no other knowledge to be provided (i.e no predicate structures, or initial and intermediate states, are required for LOCM to function). LOCM works well under the assumption that objects in a domain can be modelled as going through transitions in a object-centered state machine, and from inducing these state machines LOCM produces PDDL domain models.

One method of evaluation of LOCM is as follows: using existing hand crafted models, generate sets of plan traces. Feed these plan traces into LOCM and use it to output a domain model, then compare this output model with the original hand crafted.

The first challenge in this form of evaluation is to create a mapping between parts of the two models (the original and the learned). LOCM learns predicates, so that the 'anonymous' predicates in the learned model have to be matched up with those in the original model. A consistent mapping has to be made from the names in the learned model to the (meaningful) names in the original. In some domains, LOCM may generate domain models that, under such a name mapping, behaviourally seem equivalent to the hand-crafted model.

For example, consider the case when LOCM attempts to learn the "tyre domain" model using the original hand crafted model from which to generate plan script examples. Under the correct name mapping, a task (goal and initial state) in the notation of the original domain model can be mapped to the learned model notation; given to a planner, the output produced will be the same as if the original model was used. But consider the predicate definitions in the fragment below, which shows part of the hand-crafted model concerning "jack" related predicates, and the corresponding predicates in the induced model:

Hand Crafted:

```
(jack_in_use ?jack1 - jack ?hub1 - hub)
(have_jack ?jack1 - jack)
(jack_in ?jack1 - jack ?boot1 - boot)
```

Induced:

```
(jack_state0 ?v1 - jack ?v2 - hub)
(jack_state1 ?v1 - jack ?v2 - hub)
(jack_state2 ?v1 - jack ?v2 - hub)
(jack_state3 ?v1 - jack)
(jack_state4 ?v1 - jack ?v2 - boot)
```

The induced model contains 3 variations of "jack X in use on hub Y" predicate. Each variation records a different state of the hub assembly: when it is jacked up with the hub free, or with a loose wheel attached, or with a wheel fastened on. In this case, the learned model is certainly "close" to the original — behaviourally the two models appear the same, though the induced model is more fine grained; it may even be argued the induced one is better. The problem is, how can we compare and measure systematically such differences, and hence evaluate the learning mechanism?

LAMP (Zhuo et al. 2010) is a successor of the ARMS algorithm (Wu, Yang, and Jiang 2005), and designed to learn complex structured domain models containing quantifiers and logical implications. LAMP requires as input a set of observed plan traces, a list of actions composed of names and parameters, and a list of predicates with their corresponding parameters. It is aimed at learning in a mixed initiative fashion: the authors acknowledge that a model learned by LAMP needs to be refined by experts in order to produce a final domain model.

In the evaluation of LAMP, the authors use a very simple metric: they define the error rates of the their algorithm as the number of different predicates in either the preconditions or the effects of an operator schema, as a proportion of the total number of predicates. In the LOCM tyre example, this would result in a non-zero error rate, since there are 2 extra predicates different in the induced model (under some reasonable name mapping), and these appear in several of the operator schema.

If for example, it was required to show that with more examples, LAMP would be more accurate, then it would be necessary to show that one model was "nearer" the hand crafted version than another. Even with predefined predicates, it could be argued that the evaluation of LAMP is problematic in that is it not obvious how "near" the models it produces are using such syntactic distinctions. As the example in LOCM shows, this measure does not seem appropriate in the situation where the system learns predicates. We need measures that obey certain rules — for example a measure that is monotonic in the sense that as the error rate was lower, the domain created was "nearer" the original domain.

As well as comparing domain models for evaluation, we may want to compare or evaluate domain models as an aid to learning a new model. In the LAWS system (H.H.Zhuo, Q.Yang, R.Pan and L.Li 2011), the idea is to use a pre defined domain model from which to learn a model of a similar domain, in much the same way as a human might re-use an old model and adapt it to a new domain. Here again the concept of comparing models for their similarity is an important one, as the source domain has to be close to the target domain in some sense.

Model Comparison Definitions

The research we are carrying out is exploring properties and classifications to use in comparisons of domain models. We are utilising the wealth of past research on domain (model) analysis, and basing our ideas on our earlier investigative work (McCluskey 2003; McCluskey, Richardson, and Simpson 2002).

Strong Equivalence

We define **strong equivalence** where two domain models are strongly equivalent iff they are logically identical up to naming. To establish such an equivalence, a 1-1 mapping must exist which maps all the names in one domain to another (names of predicates, variables, operator schema, types). After a mapping has been performed, it is assumed that ordering of declarations of types, predicates and actions does not matter. "Logically Identical" here means that for each action A1 in one domain that maps to an action A2 in the other domain, A1.pre is logically equivalent to A2.pre, and A1.effects is logically equivalent to A2.effects. A good example is a translation of names in a domain model from English to French. The resulting model with names in the French language is strongly equivalent to the one in English, and vice-versa. We could perform further changes to the domain model by changing the order of declaration of types and predicate, or changing the logical conditions in the domain model using properties of logical connectives etc. In these cases the two domains would remain strongly equivalent.

Weak Equivalence

We can define **weak equivalence** for models analogous to "weak equivalence" in grammars — meaning two grammars generate the same language. The grammars may not be structurally the same, but they share the same behaviour. Thus, we say two domain models A and B are weakly equivalent iff there is a domain model B' which is strongly equivalent to B, such that:

- any task (I,G) that can be posed in A can be posed in B', and vice-versa
- for any task (I,G), any complete and corrent plan that can solve it using model A is a correct and correct plan according to B', and vice-versa.

Weak equivalence in domain models therefore means that the functional behaviour of the domain is the same for both models – the same tasks can be formulated, and the same solutions can be generated. In the LOCM example, the two domains are not weakly equivalent as the mapping between predicates in not 1-1. In this case, any tasks posed in the original can be solved in the induced model, but not the other way round.

Summary

In this paper we have highlighted the need to research into sound methods for comparing and measuring domain models. We have used two compelling examples of systems that learn domain models to illustrate the problem, and have started to develop a framework for model comparison.

The research questions we are investigating include (a) what are useful classifications of similarity of domain models? (b) can we produce support tools for checking similarity? (c) what are good metrics for domain models, and are they useful for comparing domain models?

References

Bartak, R.; Fratini, S.; and McCluskey, L. 2010. The third competition on knowledge engineering for planning and scheduling. AI Magazine, Spring 2010.

Cresswell, S.; McCluskey, T.; and West, M. M. 2011. Acquiring planning domain models using LOCM. *Knowledge Engineering Review (To Appear)*.

Ferrer, A. G. 2011. *Knowledge Engineering Techniques for the Translation of Process Models into Temporal Hierarchical Planning and Scheduling Domains*. Ph.D. Dissertation, Universidad de Granada.

H.H.Zhuo, Q.Yang, R.Pan and L.Li. 2011. Cross-Domain Action-Model Acquisition for Planning Via Web Search. In *Proceedings of ICAPS*.

Hoffmann, J. 2011. Analyzing search topology without running any search: on the connection between causal graphs and h+. *J. Artif. Int. Res.* 41:155–229.

McCluskey, T. L.; Cresswell, S. N.; Richardson, N. E.; and West, M. M. 2010. Action Knowledge Acquisition with Opmaker2. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*. Springer Berlin Heidelberg. 137–150.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *The Sixth International Conference on Artificial Intelligence Planning Systems*.

McCluskey, T. L. 2003. PDDL: A Language with a Purpose? In *Proc. PDDL Workshop, ICAPS, Trento, Italy*.

Wickler, G. 2011. Using planning domain features to facilitate knowledge engineering. In *Proc. KEPS Workshop, ICAPS*.

Wu, K.; Yang, Q.; and Jiang, Y. 2005. Arms: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence* 174(18):1540–1569.

Copyright Notice

Disclaimer

The University of Huddersfield will take reasonable care in preparing the information presented on this system. However the security, fitness for purpose and the accuracy of such information is not implied and cannot be guaranteed. Anyone who uses this information (and this system), does so at their own risk andshall be deemed to have indemnified the University of Huddersfield from any injuries or damages arisingfrom such use.

The University of Huddersfield reserves the right to immediately remove any information on this system that is the subject of legal objection, once notification of legal objection has been received.

Computing and Library Services reserve the right to immediately remove any information presented on this system that is considered to be putting the University of Huddersfield or its services at risk.