Reducing Accidental Complexity in Planning Problems

Patrik Haslum
National ICT Australia
Patrik.Haslum@nicta.com.au

Abstract

Although even propositional STRIPS planning is a hard problem in general, many instances of the problem, including many of those commonly used as benchmarks, are easy. In spite of this, they are often hard to solve for domain-independent planners, because the encoding of the problem into a general problem specification formalism such as STRIPS hides structure that needs to be exploited to solve problems easily. We investigate the use of automatic problem transformations to reduce this "accidental" problem complexity. The main tool is abstraction: we identify a new, weaker, condition under which abstraction is "safe", in the sense that any solution to the abstracted problem can be refined to a concrete solution (in polynomial time, for most cases) and also show how different kinds of problem reformulations can be applied to create greater opportunities for such safe abstraction.

1 Introduction

Planning, even in the very restricted case of the propositional STRIPS formalism, is, in general, a hard problem (in fact, it is PSPACE complete; Bylander, 1991). But not all planning problems are hard: in fact, it has been shown that in many of the benchmark domains frequently used to evaluate planning systems, a plan can be found by simple procedures running in low-order polynomial time [Helmert, 2006b]. In spite of this, the problems are not easy for domain-independent planners. That is not to say that planners can not solve problems in these domains – many recent planners scale up fairly well – but what determines the difficulty of a particular planning problem for a given planner is often not the intrinsic complexity of the problem, but more the size and structure of the problem encoding. This discrepancy between the *essential* complexity of the problem being solved and the *accidental* complexity, i.e., the difficulties added to the problem as a result of the way the problem is formulated, the tools adopted for solving it, etc., is not unique to automated planning[1].

We investigate some transformations aimed at reducing the accidental complexity of planning problems, mainly by abstracting away parts of the problem that are easily solved. Abstraction has a long history in AI planning, and it has been shown that it can reduce problem-solving effort exponentially [Knoblock, 1994] (but also potentially increase the required effort [Bäckström and Jonsson, 1995]). Methods of abstracting, decomposing or "factoring" planning problems have attracted renewed interest recently [*e.g.* Sebastia *et al.*, 2006; Domshlak and Brafman, 2006].

We focus on "safe" abstraction, i.e., on finding conditions under which solutions to an abstracted problem are guaranteed to be refinable into a concrete solution. In most abstractions we consider, refinement of an abstract solution is a polynomial-time operation (though, in the worst case, the generated plan may be exponentially longer than the shortest plan). We also show examples of how "difficult" problem formulations can make safe abstraction impossible, and suggest some methods to (automatically) reformulate problems to make them more amenable to abstraction.

Due to space restrictions, proofs and detailed algorithm descriptions are omitted from the paper, but will be provided on request.

2 Representation

A domain-independent planner is – indeed must be – presented with an encoding of a planning problem in a problem specification formalism. Our starting point is the standard propositional STRIPS representation but, like several researchers have recently done, we also consider an alternative representation based on multi-valued, rather than propositional, state variables [Jonsson and Bäckström, 1998; Edelkamp and Helmert, 1999; Helmert, 2006b]. In this representation, a world state assigns to each of a set of variables, $\{V_1, \ldots, V_n\}$, a value from a finite domain D_i. Each action is described by a pre- and a postcondition, which are partial assignments over the set of variables. The postcondition is interpreted as an assignment: on execution of the action, the variables mentioned in the postcondition are changed as indicated and variables not mentioned keep their values.

[1]The terms essential and accidental complexity were coined in the area of software engineering, by Fred Brooks in 1986. In his paper, essential complexity refers to the complexity of the functionality of a program or algorithm, while the accidental complexity is the "complexity overhead" caused by inadequate tools or languages, making simple solutions difficult to state.

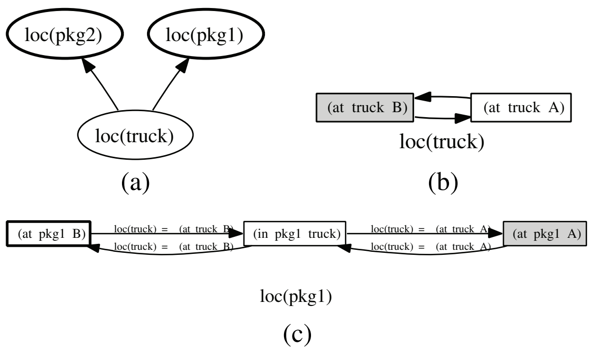


Figure 1: (a) Causal graph for a simple logistics problem; (b) and (c) domain transition graphs of two problem variables. Edges in the DTG in figure (c) are annotated with the preconditions of the corresponding actions on other variables.

The multi-valued state variable representation is expressively equivalent to propositional STRIPS and can be constructed from the STRIPS representation by making use of state invariants of a particular kind, namely sets of atoms with the property that exactly one atom in each set is true in every reachable world state. Several methods for automatically extracting such "exactly one" invariants from STRIPS encodings have been proposed [*e.g.* Edelkamp and Helmert, 1999; Scholz, 2004; Helmert, 2006b].

We also make use of two kinds of graphs that summarise some structural properties of a planning problem: The causal graph of a planning problem is a directed graph whose nodes correspond to problem variables (multi-valued or propositional) and that has an edge from V to V' iff some action that has an effect on V' has a precondition or effect on V. In particular, the causal graph contains a bidirected edge between any pair of variables such that there exists an action that changes both simultaneously. The domain transition graph (DTG) of a (multi-valued) variable V is a directed graph whose nodes correspond to the values in the domain of V and whose edges correspond to actions that change the value of the variable: the graph has an edge from v to v' if there exists some action whose precondition includes $V = v$ and whose postcondition includes $V = v'$.

To illustrate, consider a very simple transportation problem: two packages (`pkg1` and `pkg2`) can be loaded in and unloaded from a truck, which can move freely between some locations (`A`, `B`). The multi-valued variable representation of this problem has a variable `loc(truck)`, with values {`(at truck A)`, `(at truck B)`}, and a variable `loc(?p)` for each package `?p`, with values {`(at ?p A)`, `(at ?p B)`, `(in ?p truck)`}. Note that values in the variable domains correspond to atoms in the STRIPS encoding of the problem. The causal graph and domain transition graphs of two variables are shown in figure 1.

3 Abstraction

The idea of hierarchical abstraction in planning is to solve the problem first in an abstracted version, in which only the "critical" aspects of the problem are considered and the "details" are ignored. Because these "details" are not taken into account, the solution to the abstract problem may not be a valid plan. The next step is to find, in the non-abstracted version, a plan to "bridge the gap" between each step of the abstract plan. Solving the abstract problem is generally easier, as is filling in the missing details because the plan fragments needed to bridge consecutive steps in the abstract plan are typically short. This can be done recursively, with each "bridging plan" successively refined through a number of abstraction levels.

In an ideal abstraction hierarchy any solution at a higher level of abstraction can be refined into a solution at the next lower abstraction level by only inserting some actions ("fill in the details"), that is, without changing the solution as it appears at the more abstract level. This is the *downward refinement* property, as defined by Bacchus and Yang [1994]. When this property does not hold, the abstract solution found by a planner may not be refinable to a concrete solution, forcing backtracking to generate a new solution at the higher abstraction level.

Knoblock [1994] has proposed an efficient procedure for automatically generating abstraction hierarchies satisfying a weaker property, called *ordered monotonicity*: in such a hierarchy, there exists at least one abstract plan that can be refined into a concrete plan (if the problem is solvable). Knoblock's procedure constructs a graph over the atoms in the planning problem (essentially the causal graph) and computes the tree of strongly connected components of this graph which is then topologically sorted to yield a hierarchy. The procedure can equally be applied to the multi-valued variable representation of a problem, resulting in the same hierarchy. In the transportation problem described above, the resulting hierarchy considers at the most abstract level only the location of one package; at the next, the other package as well; and only at the concrete level, the location of the truck.

If every "refinement problem", i.e., the subproblem solved when refining a plan down one level in the abstraction hierarchy, is known to be solvable, then clearly any abstract plan can be refined. This is the case in the example transportation problem: as the variable `loc(truck)`, representing the location of the truck, can change from any value to any other value, whatever the state of the other variables and without changing any of them, it is always possible to refine an abstract plan for moving the packages by inserting actions to move the truck to appropriate locations.

More generally, the downward refinement property can be ensured by abstracting only when some sufficient condition for the solvability of all (relevant) refinement problems is met. This idea, termed "safe abstraction" was recently introduced by Helmert [2006a], as an improvement of the Fast Downward planner[2]. To formalise safe abstraction, define the *free domain transition graph* of a variable V as the subgraph of the DTG containing only the edges due to actions that do not have any pre- or postcondition on any other variable. A value v' of V is *free reachable* from value v iff there is a path from v to v' in the free DTG of V. Helmert's condition for safe

[2]A similar technique is for numeric variables is described by Chen et al. [2006].

```
Variables:               Actions:

 b1: {0, 1}               inc1:
 b2: {0, 1}                pre:  b1 = 0
 b3: {0, 1}                post: b1 = 1

Init: b1 = 0,             inc2:
      b2 = 0,              pre:  b1 = 1, b2 = 0
      b3 = 0               post: b1 = 0, b2 = 1

Goal: b1 = 1,             inc3:
      b2 = 1,              pre:  b1 = 1, b2 = 1, b3 = 0
      b3 = 1               post: b1 = 0, b2 = 0, b3 = 1
```

Figure 2: A small instance of the binary counter planning problem.

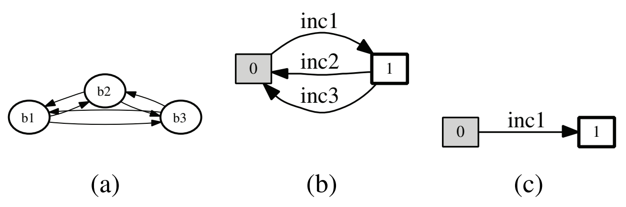


Figure 3: (a) The causal graph of the problem in figure 2. (b) The DTG of variable `b1`; (c) the free DTG of `b1`.

abstraction is that the free DTG forms a single, strongly connected component, i.e., that every value is free reachable from every other value. This condition, however, is unnecessarily strong: for downward refinement to be ensured, it is sufficient that any value of the variable that is required to refine an abstract plan is free reachable from every value that the variable may take on as a consequence of that plan. Let the *externally required* values of variable V be those values that appear in the precondition of some action that changes a variable other than V and let the *externally caused* values of V be those values that appear in the postcondition of some action that changes a variable other than V, plus the value of V in the initial state.

Theorem 1 *If all externally required values of V are strongly connected in the free DTG and free reachable from every externally caused value of V, and the goal value of V (if any) is free reachable from each externally required value, then abstracting V is safe (i.e., preserves downward refinement).*

Checking the safe abstraction condition and refining each step in an abstract plan are both polynomial in the size of the domain of the state variable, which corresponds to the number of atoms that make up the variable. The principle is straightforwardly transfered to components of more than one variable, but at exponentially increasing computational cost since the "domain transition graph" of such a component is the abstract state space of all variables in the component.

To illustrate the difference that the weaker condition makes, consider a planning problem involving a binary counter that can only be increased, in steps of one: the problem description, in multi-valued variable form, for a small instance is shown in figure 2. Variables `b1`, `b2`, `b3` represent the bits of the counter. Figure 3 shows the causal graph of the problem, and the DTG and free DTG of variable `b1`. The free DTG of variable `b1` is not strongly connected; however, the only externally required value is `1` (a precondition of `inc2` and `inc3`, and a goal) and the only externally caused value is `0`, and since `1` is indeed free reachable from `0`, variable `b1` is safely abstractable. Moreover, abstracting away `b1` makes the transition from `0` to `1` in the domain of `b2` free, and thus `b2` also safely abstractable in the same way; following the abstraction of `b2` also `b3`; and so on.

Note that the causal graph of the problem, shown in figure 3(a), forms a single strongly connected component. Thus, Knoblock's abstraction procedure applied to this problem yields only a single abstraction level, comprising the entire problem. The causal graph summarises all possible causal dependencies in a planning problem (recall that it has an edge between variables V and V' iff there *exists some action* that changes V' and depends on or changes V). The free DTG condition, on the other hand, focuses on what is possible to achieve independently of other variables.

4 Reformulation

Typically, there are several ways that a planning problem can be encoded in a specification formalism, such as STRIPS. It is widely recognised that different problem formulations may be more or less suited to a particular planning system or technique. Although the abstraction conditions discussed in the previous section rely on the causal and domain transition graphs, which are considered to be "structural" features of a problem, they too suffer from such encoding sensitivity. Ideally, a planner should be able to accept any problem formulation, and if necessary reformulate it in a way more suited to the planners needs. Sometimes, such reformulation is possible by exploiting structure in the problem description.

As an example, consider the well known Towers-of-Hanoi problem. This problem was used by Knoblock [1994] to demonstrate the construction of an effective abstraction hierarchy. Figure 4(a) shows the encoding used by Knoblock: it represents the position of each disc by the peg that the disc is on and has a different action for moving each disc. Figure 4(b) shows a different encoding of the domain, introduced by Bonet & Geffner [2001]. In this formulation, the position of each disc is either on a peg (with noting beneath it) or on a larger disc, and there is a set of additional atoms to represent that a disc or peg is "clear" (has nothing on it). The same action moves each disc.

Both encodings are correct, in the sense that they capture the essential constraints of the original problem, but they differ in their properties. Figure 5 shows the corresponding causal graphs (for the STRIPS representation of the two problem formulations). The causal graph for Knoblock's formulation consists of three strongly connected components (each contains atoms relating to the position of one disc), forming a tree, and thus a three-layered abstraction hierarchy. The causal graph for Bonet's & Geffner's formulation, however, has only a single connected component. Thus, Knoblock's procedure does not find any abstraction hierarchy in this formulation of the problem; neither does the safe abstraction condition defined above. For finding safe abstractions, a

```
Atoms:
 (on ?disc ?peg)     ?disc = large, medium, small
                     ?peg = peg1, peg2, peg3

Actions:
 (move-small ?p1 ?p2):
  pre: (on small ?p1)
  add: (on small ?p2)
  del: (on small ?p1)

 (move-medium ?p1 ?p2):
  pre: (on medium ?p1), (not (on small ?p1)), (not (on small ?p2)))
  add: (on medium ?p2)
  del: (on medium ?p1)

 (move-large ?p1 ?p2):
  pre: (on large ?p1), (not (on medium ?p1)), (not (on medium ?p2)),
       (not (on small ?p1)), (not (on small ?p2)))
  add: (on large ?p2)
  del: (on large ?p1)
```

(a)

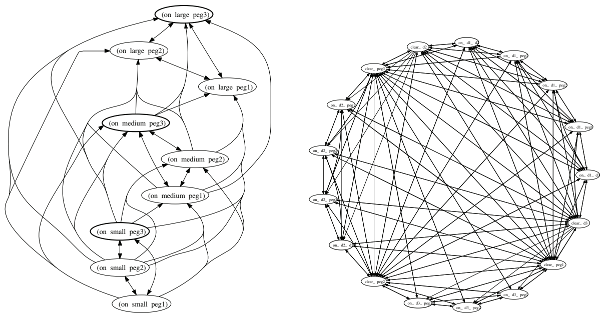
```
Atoms:
 (on ?disc ?pos)     ?disc = large, medium, small
                     ?pos = peg1, peg2, peg3, or any smaller disc
 (clear ?pos)

Actions:
 (move ?d ?p1 ?p2):
  pre: (can-go-on ?d ?p2), (on ?d ?p1), (clear ?d), (clear ?p2)
  add: (on ?d ?p2), (clear ?p1)
  del: (on ?d ?p1), (clear ?p2)
```

(b)

Figure 4: Two different STRIPS encodings of the Towers-of-Hanoi problem.

problem formulation with less "coupling" (fewer mutual dependencies) between variables is generally better. How can Bonet's & Geffner's STRIPS formulation be rephrased – automatically – so as to be if not identical to Knoblock's formulation, then at least as good for this purpose?

The atom `(clear ?d)` in the second encoding is actually equivalent to the statement $\neg\exists x$`(on` x `?d)`, i.e., the atom acts as an "abbreviation" for a more complex condition. This equivalence is simply a restatement of an "exactly one" invariant (the same kind used to construct the multi-valued variable representation of the problem). For example, {`(clear large)`, `(on medium large)`, `(on small large)`} is such an invariant; thus, `(clear large)` is true exactly when $\neg$`(on medium large)` $\wedge$ $\neg$`(on small large)` holds, and consequently can be replaced by the latter in action preconditions and in the goal[3]. Performing such substitutions it is possible to eliminate all `clear` atoms from action preconditions, and once these atoms do not appear as preconditions (or goals) they can be removed from the problem entirely.



(a) (b)

Figure 5: Causal graphs of the two formulations of the Tower-of-Hanoi problem shown in figure 4.

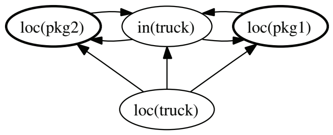


Figure 6: Causal graph for the capacity restricted transportation problem.

In the present example, however, this is not enough to make the problem hierarchically abstractable. To obtain such a formulation, it is necessary to invent new propositions abbreviating certain conditions, and substitute these into action preconditions and effects. For example the relevant conditions for the `medium` disc are `(on medium ?peg)` $\vee$ `((on medium large)`$\wedge$`(on large ?peg))`, for each `?peg`.

Eliminating atoms based on "exactly one" invariants is a simple operation, easily automated. However, each invariant gives rise to several possible – mutually exclusive – substitutions, and chosing which – if any – should be applied to arrive at a better problem formulation is not so easy, and automating the introduction of new abbreviating atoms even more difficult. Also, while the substitution is polynomial in time, it may introduce complex action preconditions, which may require exponential space to compile away.

5 Composition

Let us return to the simple transportation problem in section 2, but add a capacity restriction on the truck: it can carry at most one package at a time. This is naturally encoded with a new variable, `in(truck)`, with values {`(empty truck)`, `(in ?p truck)`}, for each package `?p`[4]. The causal graph of the new problem is shown in figure 6.

The variable `loc(truck)` is safely abstractable as before, but the variables representing the locations of the packages are now dependent, via the `in(truck)` variable, so the abstracted problem does no longer nicely decompose. The variable `in(truck)` could be eliminated by reformulation as described in the previous section, using the invariant fact that `(empty truck)` is equivalent to $\neg$`(in pkg1 truck)` $\wedge$ $\neg$`(in pkg2 truck)`, but this does not remove the coupling between variables `loc(pkg1)` and `loc(pkg2)`.

However, in this problem, unlike the Towers-of-Hanoi problem above, the coupling between variables `loc(?p)` and `in(truck)` is "transient". Assuming the goal is to move package `?p` to some location (not to leave it in the truck), the abstract plan consists of two steps: loading `?p` at

[3]Alternatively, `clear` could be recast as a *derived predicate*, using axioms as defined in PDDL2.2.

[4]The same kind of restriction, encoded in essentially the same way, is found in, e.g., the Gripper and Satellite domains.

its initial location and unloading it at the goal location (recall that the location of the truck has been abstracted away). Although the actions in this plan change both variables, the plan as a whole does not: `in(truck) = (empty truck)` holds both at the start and end of it, and only `loc(?p)` changes permanently. Moreover, the intermediate state visited by this abstract plan is "uninteresting", in the sense that the only actions that can be taken in this state that could not also be taken in the state before or after the plan are the one that completes the plan and one that just undoes the effects of the first. Thus, the two actions in the plan can be replaced by a single action, with the compound pre- and postconditions of the sequence, without affecting the solvability of the problem, and performing this replacement for all such sequences breaks the coupling between the two variables.

Theorem 2 *Let c be a condition on two or more variables, let A be the set of all actions whose effect includes c*[5]*, and let B be the set of actions whose precondition includes c. If c does not hold in the initial state; c is either inconsistent with or disjoint from the effects of every action not in A; the postcondition of every action in A is either inconsistent with or disjoint from the goal condition; and every action not in B whose precondition is consistent with c is commutative with every action in $A \cup B$, then replacing the actions in $A \cup B$ with one composite action for each executable sequence $a, b_1, \ldots, b_k$, where $k \geqslant 1$, $a \in A$ and each $b_i \in B$ – excluding sequences with no effect – is safe (i.e., preserves solution existence).*

If the safe sequencing condition holds for every effect c involving some pair of variables V_1 and V_2, and if the effects of each of the new composite actions change at most one of V_1 and V_2, this replacement removes the causal coupling between the two variables due to simultaneous change. There may still be causal dependencies between them due to some action changing one having a precondition on the other: if such dependencies exist in both directions, nothing is of course gained by making the replacement.

Collapsing a sequence of actions into a single action has also been proposed by Junghanns & Schaeffer [2001], in the context of solving the Sokoban puzzle, in the form of what they call "tunnel macros". These sequences correspond to moves in a tunnel, where there is only one "productive" direction of movement, and also satisfy the criterion of having "uninteresting" intermediate states (the condition c in theorem 2 can be thought of as defining a "tunnel" in the state space of the planning problem). However, tunnel macros in Sokoban serve a different purpose, viz. to reduce search depth, and are not restricted to actions that cause coupling between variables. Likewise, Botea et al. [2005] show that adding macro actions to a planning problem can improve the efficiency of search (and, in some cases, heuristic accuracy; interestingly, so can removing macros [Haslum and Jonsson, 2000]). Yet, the procedure they propose for generating macros may be adaptable to our purpose as well.

[5]So that if A is non-empty there is a causal coupling due to simultaneous change of the variables mentioned by c.

Domain	Simplification
Gripper	Solved.
Logistics	Solved.
Movie	Solved.
Mystery	None.
Mprime	None.
Grid	Minor ($\sim$ 50% less atoms).
Blocksworld (4 ops)	Minor (0 – 20% less atoms)
Elevator (STRIPS)	Solved.
FreeCell	None.
Depots	Minor (1 – 10% less atoms).
DriverLog	Minor (0 – 25% less atoms; small instances solved).
Rovers	Significant (60 – 90% less atoms).
Satellite	Solved.
Airport	Some (40 – 60% less atoms).
Pipesworld	None.
Promela	None.
PSR (small)	Varied (0 – 50% less atoms).

Figure 7: Simplification obtained by current techniques on planning problems from different domains.

6 Conclusions

Abstraction and decomposition (or "factoring") can be very powerful tools for reducing the accidental complexity caused by encoding a planning problem in a general problem specification formalism, such as STRIPS. We have shown how the applicability of these methods can be widened, by formulating a weaker condition for safe abstraction and by applying different reformulations to make problems more amenable to abstraction.

Implementation & Current Results

The techniques described above, together with the action set reduction method proposed by Haslum & Jonsson [2000] and standard reachability and relevance analysis[6], have been implemented, though with some limitations: safe abstraction of more than one variable is limited by a bound on the size of the composite DTG, reformulation is restricted to atom eliminations with a simple replacing condition, and composition to sequences of length two. The strategy is to try each technique in turn, iterating until no more simplifications are made. In spite of the restrictions, some operations are still too time-consuming, and therefore disabled, in some domains. Also, different methods for finding invariants in some cases result in different multi-valued variable encodings, some of which are more suited to simplification than others. Thus, the system is not completely automatic.

Table 7 summarises the (best achieved) result of attempting to simplify a collection of planning domains, comprising most of the domains from the 1st, 2nd, 3rd and 4th planning competitions. Many of these encode very simple problems, and indeed in many of them (those marked "solved" in the table) repeated abstraction removes the entire problem, so that

[6]The relevance analysis uses ideas similar to Scholz's [2004] path reduction technique, which is stronger than the standard method.

these problems are solved without search, except for the simple refinement searches carried out in the DTG of individual variables (the Logistics and Elevator domains are solved in this sense also with Helmert's safe abstraction method). For remaining domains, simplification is roughly measured by the reduction in problem size and a subjective estimate of the "complexity" of the part that remains (for example, in the Grid domain, only some irrelevant objects are removed; though this cuts down size significantly, it's viewed only as a "minor" simplification). How problem simplification affects the performance of different planners is a far more complicated question (for example, for $\frac{2}{20}$ Depots problems the – very minor – simplification drastically improves the accuracy of FF's heuristic, resulting in significant speedup, while for one problem time increases and for remaining problems it has no effect).

Open Problems

Most interesting to note among the results, however, are the failures, that is, domains that model essentially simple problems, but which the current suite of techniques fail to simplify, such as the Blocksworld, Schedule and PSR domains. What methods are needed to effectively deal with these domains? One observation is that our current techniques make very little use of knowledge about the particular problem instance. In the Blocksworld domain, for example, every variable potentially interacts with every other variable, making the problem difficult to abstract or decompose, but the interactions that actually need to be considered in the solution of any given problem instance are typically few. Much use has been made of the causal graph to analyse planning problems, but it is, as noted above, only a very coarse summary of potential interactions. The weaker abstraction condition yields a more fine grained view but still takes into account many possible, not actually relevant, interactions.

The current implementation works on a grounded (propositional or multi-valued) problem representation, which makes some steps computationally expensive. Applying these and other techniques in a first order setting, and to domain formulations using ADL constructs, is an important future development. Also, as noted above, the choice of transformations to apply is not fully automatic. This is a hard problem, where we see a potential for learning to play an important role.

Acknowledgements

Hector Geffner, Adi Botea, Malte Helmert and the IJCAI reviewers have all contributed ideas, advice, discussions and helpfull comments on various drafts of this paper. NICTA is funded through the Australian governements *backing Australias ability* initiative, in part through the Australian research council.

References

[Bacchus and Yang, 1994] F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43 – 100, 1994.

[Bäckström and Jonsson, 1995] C. Bäckström and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1599 – 1605, 1995.

[Bonet and Geffner, 2001] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5 – 33, 2001.

[Botea *et al.*, 2005] A. Botea, M. Enzenberger, M. Mueller, and J. Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of AI Research*, 24:581 – 621, 2005.

[Bylander, 1991] T. Bylander. Complexity results for planning. In *Proc. 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 274 – 279, 1991.

[Chen *et al.*, 2006] Y. Chen, B.W. Wah, and C. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of AI Research*, 26:323 – 369, 2006.

[Domshlak and Brafman, 2006] C. Domshlak and R. Brafman. Factored planning: How, when and when not. In *Proc. 21st National Conference on Artificial Intelligence (AAAI'06)*, 2006.

[Edelkamp and Helmert, 1999] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. 5th European Conference on Planning (ECP'99)*, pages 135 – 147, 1999.

[Haslum and Jonsson, 2000] Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In *Proc. 5:th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 150 – 158, 2000.

[Helmert, 2006a] M. Helmert. Fast (diagonally) downward. In *5th International Planning Competition Booklet*, 2006. Available at `http://zeus.ing.unibs.it/ipc-5/`.

[Helmert, 2006b] Malte Helmert. *Solving Planning Tasks in Theory and Practice*. PhD thesis, Albert-Ludwigs Universität Freiburg, 2006.

[Jonsson and Bäckström, 1998] P. Jonsson and C. Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125 – 176, 1998.

[Junghanns and Schaeffer, 2001] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219 – 251, 2001.

[Knoblock, 1994] C.A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243 – 302, 1994.

[Scholz, 2004] U. Scholz. *Reducing Planning Problems by Path Reduction*. PhD thesis, Technischen Universität Darmstadt, 2004.

[Sebastia *et al.*, 2006] L. Sebastia, E. Onaindia, and E. Marzal. Decomposition of planning problems. *Artificial Intelligence Communications*, 19(1):49 – 81, 2006.