From: AAAI Technical Report SS-93-03. Compilation copyright © 1993, AAAI (www.aaai.org). All rights reserved.

Forbidding Preconditions and Ordered Abstraction Hierarchies

Eugene Fink *
School of Computer Science
Carnegie Mellon University
Pittsbugrh, PA 15213
eugene@cs.cmu.edu

Qiang Yang *
Department of Computer Science
University of Waterloo
Waterloo, Ont., Canada N2L3G1
qyang@logos.waterloo.edu

Abstract

The use of abstraction in problem solving is an effective approach to reducing search, but finding good abstractions is a difficult problem. The first algorithm that completely automates the generation of abstraction hierarchies is Knoblock's ALPINE. The algorithm is based on the notion of *ordered abstraction hierarchies*, which formalizes an important intuition behind "good" hierarchies. In this paper we continue the work on formalizing the notion of ordered hierarchies. The paper shows that the hierarchies generated by ALPINE are over-constrained by restrictions that are not necessary for the ordered monotonicity property. We present *necessary and sufficient* conditions for an abstraction hierarchy to be ordered and describe an algorithm based on these conditions to improve the quality of abstraction hierarchies.

Introduction

Recently, there has been an increasing amount of interest in formalizing abstraction planning. Much work has stemmed from Sacerdoti's ABSTRIPS system [Sacerdoti, 1974]. A notable recent achievement is Knoblock's ALPINE system [Knoblock, 1991], which completely automates the formation of abstraction hierarchies. The hierarchies constructed by ALPINE satisfy the *ordered monotonicity property* (OMP) [Knoblock *et al.*, 1991], which states that while refining an abstract plan on a concrete level, no abstract-level literal may be changed. To a large extent, the property is successful in formalizing an intuition behind the use of abstraction: one wants to preserve the structure of an abstract plan while adding concrete-level operators. Experiments [Knoblock, 1991] have demonstrated that ALPINE gains an exponential amount of running time saving in many planning domains.

ALPINE generates an abstraction hierarchy by imposing ordering constraints on the criticalities of literals in a planning domain. A problem with ALPINE is that it often generates a hierarchy with too few levels. In some cases, the entire hierarchy collapses into a single level. The "collapsing problem" of ALPINE makes it incapable of handling many practical problems.

A careful examination of ALPINE reveals a source of this problem. The constraints used by Knoblock's algorithm for generating hierarchies are stronger than required for ensuring the OMP, and the "collapsing problem" is often caused by unnecessary constraints imposed by ALPINE on the criticalities of literals, which reduce the number of abstraction levels.

In this paper we present a *necessary and sufficient* condition of the OMP. The new condition depends on a special type of operator preconditions, called *forbidding preconditions*. If a forbidding precondition of some operator does not hold, no plan can achieve all preconditions of the operator, and the operator can never be applied. We show that if forbidding preconditions can be detected, then one can impose a weaker set of constraints onto the domain while still obtaining the OMP as ALPINE does.

*The first author is supported by Carnegie Mellon University. The second author is supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and ITRC.

Ordered hierarchies

We follow the terminology used in [Knoblock *et al.*, 1991]. A *planning domain* consists of a set of literals and a set of operators. Each *operator* α is defined by a set of *precondition literals* $Pre(\alpha)$ and *effect literals* $Eff(\alpha)$. A *state* of the world is a set of literals. A *plan* $\Pi = (\alpha_1, .., \alpha_n)$ is a sequence of operators, which can be applied to a state by executing each operator in order. A plan $\Pi = (\alpha_1, .., \alpha_n)$ is *correct* relative to an initial state if the preconditions of each operator are satisfied in the state in which the operator is applied, i.e. $(\forall i \in [1..n])\ Pre(\alpha_i) \subseteq S_{i-1}$.

A *planning problem* is a pair (S_0, S_g), where S_0 is an initial state, and S_g is a *goal state*. A plan Π is *correct* relative to the planning problem (S_0, S_g) if Π is correct relative to S_0 and the goal is satisfied in the final state: $S_g \subseteq S_n$.

To build an abstraction hierarchy, one can associate some natural number, called *criticality*, with every literal in the planning domain. While solving a problem at *the i-th level of abstraction*, we ignore all literals with the criticalities less than i. Abstraction planning is usually done in a top-down manner: first we find a solution at the highest level of abstraction. Then we refine it to account for successive levels of details.

For a given set of operators, different criticality assignments may result in different behavior in problem solving. *Ordered hierarchies*, introduced in [Knoblock *et al.*, 1991], satisfy the condition that every refinement of an abstract plan leaves the abstract plan structurally unchanged. More precisely,

Definition 1 (Ordered hierarchy)
(1) *A refinement of the* $(i+1)$*-st level plan* Π *is called* ordered *if none of the operators inserted into* Π *changes the value of any literal with the criticality value higher than* i.

(2) *An abstraction hierarchy is* ordered *if every justified refinement of every correct plan is an ordered refinement.*

For example, consider a planning domain with a robot that may travel within several rooms and turn on and off switches. We may build a two-level ordered hierarchy for this domain, where the status of switches is on the upper level and the position of the robot is on the lower level. Travelling between rooms does not change the status of switches, and thus a plan for changing the position of the robot cannot change anything on the upper level.

To guarantee the OMP, ALPINE imposes ordering constraints on literals in the domain based on the following syntactic restrictions:

Restrictions 1 and 2
Let O be the set of the operators in a domain. $\forall \alpha \in O$, $\forall l_1, l_2 \in Eff(\alpha)$, and $\forall l \in Pre(\alpha)$ such that l can be achieved by some operator,
(1) $crit(l_1) = crit(l_2)$, and
(2) $crit(l_1) \geq crit(l)$.

Thus, all effects of an operator have the same criticality, and their criticality is at least as great as the criticalities of operator's preconditions (except the preconditions that cannot be achieved by any operator).

Restriction 1 ensures that while adding an operator to achieve a literal l_1 on some level of abstraction, we change only literals on the same level, since the operator does not have effects on other levels. The intuition behind the second restriction is a little bit more complicated. After inserting some operator α to achieve l_1, we may need to achieve the preconditions of α. Restriction 2 ensures that the criticalities of its preconditions is no larger than the criticality of l_1, and thus we do not change anything on higher levels while achieving them.

It turns out that the second restriction is stronger than needed to guarantee the OMP. Thus, Restrictions 1 and 2 are only sufficient, but not necessary. To demonstrate this, we consider a simple robot world in which a robot can unlock and open a safe using two keys. The robot can unlock the safe if he has both keys, and the unlocked safe may be opened. If the safe is open, the robot can put the keys into the safe. The robot is limited in his ability to pick up a key dropped onto the floor: when he picks up a key, he loses the other one. This domain may be described with 5 literals:

have key 1	unlocked	keys in safe
have key 2	open	

and the negations of these literals. The formal description of the operators is shown in Figure 1.

It can be shown that ALPINE fails to generate a hierarchy for this domain: all literals collapse into a single level of abstraction. The constraints are shown in Figure 2, where arrows denote "$\leq$" relations between the criticalities of literals. The only

Pick Key 1	
Preconds	Effects
keys not in safe	have key 1 do not have key 2

Pick Key 2	
Preconds	Effects
keys not in safe	have key 2 do not have key 1

Unlock Safe	
Preconds	Effects
have key 1 have key 2	unlocked

Open Safe	
Preconds	Effects
unlocked	open

Put Keys	
Preconds	Effects
open have key 2 have key 1	keys in safe

Figure 1: Operators

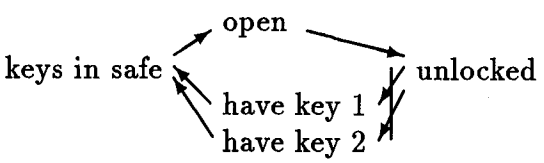

Figure 2: Constraints

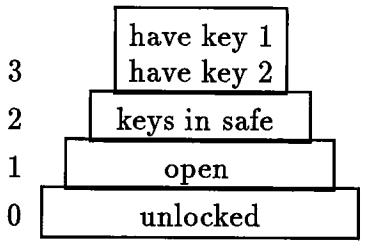

Figure 3: Ordered hierarchy

way to satisfy the constraints is to make the criticalities of all literals equal to each other, and thus the hierarchy collapses into a single level[1].

Observe however that once the robot has dropped one of the keys, he can never hold both of them again, and therefore he can never open the safe. In other words, if one of the two preconditions of the operator *Unlock Safe*, either "*have key 1*" or "*have key 2*", does not hold, the planner may never achieve both of them. So an intelligent planning system should treat these preconditions as unachievable. Since Restriction 2 must hold only for achievable preconditions, we may remove constraints imposed on the preconditions of the operator *Unlock Safe*. These constraints are crossed with a vertical line in Figure 2. Similarly, we do not need to impose constraints on the preconditions "*have key 1*" and "*have key 2*" of the operator *Put Keys*. One may verify that without these two constraints we obtain the four-level ordered hierarchy shown in Figure 3.

[1]One may verify that the hierarchy collapses into a single level even if we use *problem-specific* ordered restrictions [Knoblock *et al.*, 1991].

Necessary and sufficient conditions

To formalize the observation made in the example, we introduce the notion of *forbidding* preconditions. Informally, a precondition l of an operator α is forbidding if, once l does not hold, we can never achieve the preconditions of α, and thus we can never apply α. In the above example, the literals "*have key 1*" and "*have key 2*" are forbidding preconditions of the operator *Unlock Safe*.

Definition 2 *A precondition l of an operator α is a* forbidding precondition *if for any state S_0 in which l does not hold, no correct plan with the initial state S_0 may contain α.*

Note that the concept of forbidding preconditions is different from that of *filter* preconditions used in NONLIN [Tate, 1977]. Filter preconditions are used to distinguish the different context-dependent effects of operators. The major difference between the two is that a filter precondition can be made true and false several times in a plan, while the preconditions of an operator containing a forbidding precondition will *never* become true after the forbidding precondition is made false. The usefulness of forbidding preconditions is not affected by how a planner is implemented, while filter preconditions have been shown to be ineffective for backwards-chaining planners [Collins and Pryor, 1992].

Now we state the modified restrictions on the criticality values of literals, suggested in our example.

Restrictions 1 and 2′
Let O be the set of the operators in a domain. $\forall \alpha_1 \in O$, $\forall l_1, l_2 \in \mathit{Eff}(\alpha_1)$, and $\forall l \in \mathit{Pre}(\alpha_1)$,

1. $\mathit{crit}(l_1) = \mathit{crit}(l_2)$, and

2′. if l is not a forbidding precondition of α, then $\mathit{crit}(l_1) \geq \mathit{crit}(l)$.

Our first restriction is the same as Restriction 1 of the ALPINE system. Let us consider the difference between Restriction 2, used by ALPINE, and our Restriction 2′. ALPINE's restriction imposes constraints onto all *achievable* preconditions of an operator, i.e. onto the preconditions that may be achieved by applying some other operators. Our restriction constrains the criticalities of non-forbidding preconditions. Observe that if a precondition cannot be achieved by any operator, it is certainly forbidding. Thus, Restriction 2′ constrains the criticalities of fewer preconditions than the corresponding restriction of ALPINE. Since our restrictions impose less constraints, they may be used to generate finer-grained abstraction hierarchies. Restrictions 1 and 2′ are also *necessary* conditions of the OMP of a hierarchy, and thus they cannot be further relaxed.

Theorem 1 *An abstraction hierarchy is ordered* if and only if *it satisfies Restrictions 1 and 2′.*

A proof of the theorem may be found in [Fink, 1992].

Observe that Restrictions 1 and 2′ impose the same type of ordering constraints as ALPINE's restrictions, and thus they may be used by ALPINE to generate hierarchies. However, we first have to determine which preconditions are forbidding.

1. paint all preconditions of α *black*
2. **for** every *black* precondition l of α **do**
   for every operator α_1 **do**
   if α_1 achieves l and does not negate any *black* precondition of α
   then paint l *white*
3. **if** some precondition is painted *white* by Step 2 **then** go back to Step 2

Table 1: Finding forbidding preconditions of α

Finding forbidding preconditions

Consider again our robot example. Once the robot dropped one of the keys, he can never hold both of them again: if the robot picks *key 1*, he drops *key 2*, and if he picks *key 2*, he loses *key 1*. This example suggests the following intuition of forbidding preconditions of an operator: preconditions $l_1, .., l_k$ are forbidding if every operator α_1 that achieves one of them *negates* some other of these preconditions. An operator α_1 *negates* a literal l_i if l_i never holds after the execution of α_1. This happens if either

(1) the negation of l_i, $\neg l_i$, is an effect of α_1, or

(2) $\neg l_i$ is a precondition of α_1, and α_1 does *not* achieve l_i.

We can use this observation to find (some of the) forbidding preconditions of operators.

The algorithm for finding forbidding preconditions of α is shown in Table 1. We use two colors, *black* and *white*, to paint the preconditions of α. Initially all preconditions are *black*. If we discover that some *black* precondition may be achieved without negating any other *black* precondition, we paint it *white*. Upon the execution of the algorithm, the preconditions of α are painted such a way that any operator achieving some *black* precondition of α always negates some other *black* precondition. Thus, once some *black* precondition does not hold, we cannot achieve all black preconditions of α, and we may never apply α. Therefore, all *black* preconditions are forbidding.

For example, if we apply this algorithm to the operator *Put Keys*, the precondition "*open*" will be painted *white*, since it may be achieved by the operator *Open Safe* without negating any other preconditions. On the other hand, the preconditions "*have key 1*" and "*have key 2*" will remain black, because every operator that achieves one of them

always negates the other. The algorithm conclude that these two preconditions are forbidding.

The described algorithm is based on a *sufficient* condition for finding forbidding preconditions: all preconditions that remain *black* are certainly forbidding, but some *white* preconditions may be forbidding as well. We may use another method to find some non-forbidding preconditions of α. For each *white* precondition l of α, we use a planning algorithm to find a plan that achieves all preconditions of α from any initial state in which l does not hold. If we find a state S in which l does not hold and a plan Π that achieves the preconditions of α starting from S, then l is *not* a forbidding precondition of α. If we limit the search depth of planning algorithm by some constant, then the running time of this method is *polynomial.*

For example, suppose we apply such an algorithm to the precondition "*open*" of the operator *Put Keys*. The algorithm will find that the preconditions of *Put Keys* may be achieved from the state S ={*not open, unlocked, have key 1, have key 2*}, by executing a one-operator plan Π =(*Open Safe*). Since the literal "*open*" does *not* hold in S, the algorithm concludes that it is *not* a forbidding precondition of *Put Keys*.

After we apply the two algorithms described above, there may still be some preconditions about which we do not know whether they are forbidding. In [Fink, 1992] we present an inductive learning algorithm that finds forbidding preconditions among them while solving planning problems. Initially the algorithm imposes constraints only on the preconditions that are definitely not forbidding. If some of the remaining preconditions are not forbidding either, it may fail to impose some necessary constraints, which would lead to problems during abstraction planning. However, when such problems occur, they reveal the lack of constraints, and the necessary additional constraints are imposed.

Conclusion

This paper describes an extension of the ALPINE algorithm for automatically generating abstraction hierarchies. Using the notion of forbidding preconditions, we are able to generate ordered hierarchies where ALPINE fails. The paper extends the theory of abstraction planning by presenting necessary and sufficient conditions of the OMP for abstraction hierarchies. The notion of forbidding preconditions is also useful for non-hierarchical planning, since the knowledge of such preconditions allows us to reduce the branching factor of search by detecting unachievable subgoals during planning process.

References

[Bacchus and Yang, 1991] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the International Joint Conference on Artificial Intelligence*, p 286–292, 1991.

[Collins and Pryor, 1992] Gregg Collins and Louise Pryor. Achieving the functionality of filter conditions in a partial order planner. In *Proceedings of the Tenth National Conference of Artificial Intelligence*, pages 375-380, 1992

[Fink, 1992] Eugene Fink. *Justified plans and ordered hierarchies.* Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Canada, 1992. Tech. Report CS-92-42.

[Knoblock, 1991] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-91-120.

[Knoblock *et al.*, 1991] Craig Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference of Artificial Intelligence*, pages 692–697, 1991.

[Sacerdoti, 1974] Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5, pages 115–135, 1974.

[Siklossy and Dreussi, 1973] L.Siklossy and J. Dreussi. An Efficient Robot Planner which generates its own procedures. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pages 423-430. Morgan Kaufmann, 1973.

[Tate, 1977] Austin Tate. Generating Project Networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888-893, 1977.