Abstraction in Nonlinear Planning

Qiang Yang [*]
University of Waterloo
Canada

Josh D. Tenenberg [†]
Indiana University at South Bend
USA

Steven Woods[‡]
Defense Research Establishment Valcartier
Canada

Abstract

Planning with abstraction is an effective method to reduce search. While various theories are emerging on how to construct good abstraction hierarchies, relatively little has been studied on how to control abstract search in order to make the best use of a given abstraction hierarchy. In this paper, we investigate how various domain-dependent and independent search heuristics could be employed to further enhance abstract planning. Our goal is to search for heuristics that are not only efficient but also complete. That is, if a solution exists then one can be found. We start by extending the hierarchical, precondition-elimination abstraction of ABSTRIPS to nonlinear, least-commitment planners such as TWEAK. Using the resulting system ABTWEAK as a test bed, we show how to improve the search efficiency using several different methods: by protecting a subset of abstract conditions achieved so far, by biasing search toward deeper levels in a hierarchy, and by appropriately exploiting domain-dependent heuristics. We also present empirical test results that demonstrate the efficiency of these methods.

Key words: Problem Solving/Planning, Abstraction, Plan Generation.

[*]Computer Science Department. Waterloo, Ont. Canada, N2L 3G1. Email: qyang@watdragon.uwaterloo.ca

[†]Department of Mathematics and Computer Science, P.O. Box 7111, South Bend, Indiana 46634 USA Email: josh@natasha.iusb.indiana.edu

[‡]Combat Intelligence Automation Group, 2459, Pie XI Blvd, North, P.O. Box 8800, Courcelette, Quebec G0A 1R0. Email: woods@jupiter.drev.dnd.ca

Contents

1 Introduction

Abstraction has been used as a method to reduce the computational complexity of classical planning. A large number of problem-solving systems, including GPS[Newell and Simon 1972], ABSTRIPS[Sacerdoti 1974], LAWLY[Siklossy and Dreussi 1973], NOAH[Sacerdoti 1977], NONLIN[Tate 1977], MOLGEN[Stefik 1981], SOAR[Unruh and Rosenbloom 1989], and SIPE[Wilkins 1984], have been developed based on the concept of abstraction. Two central issues must be studied to fully understand abstract planning. The first issue is how to construct an abstraction hierarchy automatically. Work related to this issue is aimed at uncovering the principles behind what constitutes "good" abstraction hierarchies. Various proposals have emerged to address this, including the work by Sacerdoti[Sacerdoti 1974], Knoblock[Knoblock 1991], Knoblock, Tenenberg and Yang[Knoblock, Tenenberg, and Yang 1991], Bacchus and Yang [Bacchus and Yang 1991, Bacchus and Yang 1992], and Christensen [Christensen 1990].

The second research issue on abstraction is how to make the best use of a given abstraction hierarchy; it is on this issue that we focus. We investigate how various domain-dependent and domain-independent search heuristics can be employed to further enhance abstract planning. Our goal is to search for heuristics that are not only efficient but also complete, thereby a solution can be found if a problem is solvable. Our contributions can be summarized as follows.

First, we present our implemented testbed for studying abstraction with nonlinear planning, by combining ABSTRIPS-style abstraction with TWEAK-style nonlinear planning. The result is an abstract, nonlinear planning system called ABTWEAK. Based on ABTWEAK, we argue that abstraction hierarchies provide a convenient and natural way to select subgoals to be *protected*. Under this strategy, instead of protecting every subgoal achieved so far as is done in some planning systems, we protect only an important subset of subgoals, where the importance criterion is provided by the abstraction hierarchy itself. We show that this protection method is more advantageous than arbitrary protection. Further, we show that for any ABSTRIPS-style abstraction hierarchy, protecting any set of subgoals determined by the abstraction hierarchy will not result in the loss of completeness.

Second, we show that protection itself cannot guarantee efficient planning even with a reasonably good abstraction hierarchy. An efficient abstract search strategy must be able to take advantage of the multi-layered structure to speed up search. We present *Left-Wedge*, a search heuristic that is both complete and efficient for abstract planning, and demonstrate its utility via empirical tests.

Finally, we show that although certain domain-dependent heuristics can improve search efficiency with abstraction, they often achieve this by sacrificing the completeness property. However, if these heuristics are designed in accordance with the structure of abstraction hierarchies, then completeness can be restored while still improving search efficiency.

The organization of the paper is as follows. In Section 2, we present a planning formalism based on the TWEAK language, and demonstrate our extension to ABTWEAK. Section 3 presents the monotonic goal protection heuristic, Section 4 discusses the LEFT-WEDGE heuristic for controlling abstract search, and Section 5 shows how domain-dependent heuristics can be effectively combined with abstract planning. An empirical evaluation of the heuristics is done in Section 6, and a comparison to related work is made in Section 7. Finally, conclusions are given in Section 8.

2 TWEAK and ABTWEAK

In this section, we provide a formal foundation for the rest of the paper, by reviewing the TWEAK plan language, and introducing the ABTWEAK plan language.

2.1 TWEAK

Chapman [Chapman 1987] provides a formalization of a least commitment, nonlinear planner, TWEAK, similar to but predating the recent work on the SNLP planner [McAllester-Rosenblitt, 1991]. TWEAK extends STRIPS[Fikes and Nilsson 1971] by allowing for

1. a partial temporal ordering on the operators in a plan,
2. partial constraints on the binding of variables (*codesignations*) of the operators.

A TWEAK plan thus represents a space of STRIPS plans: all totally ordered, fully ground plans that satisfy the ordering and codesignation constraints.

Formally, a TWEAK system is a pair $\Sigma = (L, O)$. L is a restricted language consisting of a finite number of predicate symbols, infinitely many constant and variable symbols, and negation. The set of *terms* of L is the constants unioned with the variables. The set of *atoms* is all expressions of the form

$$P(x_1, \ldots, x_n),$$

where P is an n-ary predicate and the x_i are terms. The *ground* atoms are the atoms where all terms are constants. The *literals* (also called *propositions*) include all atoms and their negations. Further, for any literal p, $\neg\neg p$ is equivalent to p. O is a set of operator templates (referred to simply as operators). Associated with each operator a is a set of precondition literals, P_a, and effect literals, E_a.

Chapman did not give a formal definition of a TWEAK plan [Chapman 1987]. Because this concept is very important in defining a number of others later in the paper, we formally define it as follows.

Definition 2.1 *A plan* Π *is a triple* $(\mathrm{Operators}_\Pi, \prec_\Pi, \mathrm{Co\&Nonco}_\Pi)$, *where*

- $\mathrm{Operators}_\Pi$ *is a set of operators, which are copies of operator templates, in which the template variables have new, unique names.*

- $\prec_\Pi$, *the temporal constraints, is a binary relation on* Operators$_\Pi$ *such that the transitive closure of* $\prec_\Pi$ *is a partial order (irreflexive, asymmetric, transitive),*

- Co&Nonco$_\Pi$, *the codesignation constraints, is a pair of binary relations on the terms in* L, *with* $\approx_\Pi$ *being the positive codesignations, and* $\not\approx_\Pi$ *being the non-codesignations.* $\approx_\Pi$ *is an equivalence relation.* Co&Nonco$_\Pi$ *is further constrained so that*

  Consistency: *if* $(x \approx_\Pi y)$, *then it is not the case that* $(x \not\approx_\Pi y)$, *for any terms* x *and* y, *and*

  Uniqueness of Names: *it is not the case that* $(c \approx_\Pi d)$, *for any 2 constants* c *and* d.

The plan subscripts to $\approx$, $\not\approx$, and $\prec$ will be dropped if the plan to which these relations refer is clear from context. In addition, we will use $\neg$ before an expression to mean "it is not the case that." For example, $\neg(a \approx b)$ is to be read "it is not the case that $(a \approx b)$." Note that $\neg(a \approx b)$ is *not* equivalent to $(a \not\approx b)$. Further, we extend $\approx$ and $\not\approx$ to literals in the standard way. That is, two literals codesignate if the predicates have the same number of arguments, and all corresponding arguments in the two literals codesignate. Likewise, two literals non-codesignate if the predicates have a different number of arguments, or if one or more of the corresponding arguments non-codesignate.

With the above definition, we can now restate several terminologies used in [Chapman 1987] formally. A *complete plan* Π is a plan where $\prec_\Pi$ is a linear ordering on Operators$_\Pi$, and Co&Nonco$_\Pi$ is such that every variable in every operator of Operators$_\Pi$ codesignates with some constant. A plan *completion* of Π refers to any complete plan Π' that satisfies the constraints of Π.

An operator α *asserts* literal p if there exists $q \in E_\alpha$ such that p and q codesignate, and *denies* p if its negation is asserted. A *state* is defined as a set of ground atoms in L. An input problem is taken to be a pair $\rho = (I, G)$, where I is a state (the *initial* state), and G is a set of propositions, (the *goal*).

For simplicity, the goal G can be represented by a special operator $\mathcal{G}$, where $P_\mathcal{G} = E_\mathcal{G} = G$. The initial state I can likewise be viewed as a special operator $\mathcal{I}$, with $P_\mathcal{I} = \emptyset$ and $E_\mathcal{I} = I$. These two operators will be an element of each plan Π, under the constraint that, for every other operator $\alpha \in$ Operators$_\Pi$, $(\mathcal{I} \prec_\Pi \alpha)$ and $(\alpha \prec_\Pi \mathcal{G})$.

A complete plan implicitly defines a sequence of states obtained by applying each fully instantiated operator in the sequence specified by the ordering relation. A complete plan is *correct* if for every operator, every precondition is satisfied in the state in which the operator is applied. A complete plan *solves* a problem if the plan is correct, and the goal is satisfied in the final state. A plan Π (not necessarily complete) is correct if every completion is correct, and Π solves a problem if every completion solves the problem. An equivalent definition of correctness, which we have adopted in our algorithm (see Appendix C), is Chapman's *Modal Truth Criterion (MTC)* [Chapman 1987], that has the additional advantage that it is polynomially checkable.

In the balance of the paper, we will apply Chapman's modal necessity operator ($\Box$) to propositions that are true, (or constraints that hold) in *every* completion of a given plan. Likewise, the possibility operator ($\Diamond$) denotes that a proposition (or constraint) is true in *some* completion of a given plan. Note that since plan completions are obtained by adding operators and constraints, a constraint (e.g., $a \prec b$) is necessarily true in Π if it is an element of one of the constraint sets of Π, and possibly true if its inverse (e.g., $b \prec a$) is not an element of one of the constraint sets.

2.2 AbTweak

In Abstrips, Sacerdoti developed an elegant means for generating abstract problem spaces by assigning criticality values (an integer between 0 and k, for some small k) to preconditions, and abstracting at level i by eliminating all preconditions having criticality less than i. This is formalized as follows.

A k level AbTweak system is a triple $\Sigma = (L, O, crit)$, where L and O are defined as for Tweak, and *crit* is a function mapping preconditions to non-negative integers:

$$crit : \bigcup_{\alpha \in O} P_\alpha \rightarrow \{0, 1, \ldots, k-1\}.$$

Intuitively, *crit* is an assignment of criticality values to each proposition appearing in the precondition of an operator.

Let α be an operator. We take ${}_iP_\alpha$ to be the set of preconditions of α which have criticality values of at least i:

$${}_iP_\alpha = \{p \mid p \in P_\alpha \ and \ crit(p) \geq i\}.$$

${}_i\alpha$ is operator a with preconditions ${}_iP_\alpha$ and effects E_α. Let the set of all such ${}_i\alpha$ be ${}_iO$. This defines a TWEAK system on each level i of abstraction:

$${}_i\Sigma = (L, \ {}_iO).$$

We extend this notation to plans. Given plan Π, ${}_i\Pi$ is plan Π where the operators are all drawn from ${}_iO$.

As with Abstrips, the strategy for planning with AbTweak is in a top-down manner – when a problem is input, planning proceeds first at the most abstract, least constrained level. This plan is then refined at the next lower level by inserting new operators to satisfy the re-introduced preconditions. Only after all of the preconditions are satisfied on the current level does the planner pass the plan to the level below.

Since the introduction of Abstrips, most work on abstraction has inherited the Abstrips-style top-down, length first search strategy, concentrating instead on automatically finding good abstraction hierarchies[Knoblock 1991, Christensen 1990, Bacchus and Yang 1992]. Little has been done on finding methods to reduce search based on the *structure* of a given abstraction hierarchy. As stated in Section 1, the focus of this paper is on how to make the best use of a *given* abstraction hierarchy. In the following sections, we explore how to improve the search efficiency using several different methods: by protecting a subset of abstract conditions achieved so far, by biasing search toward deeper levels in a hierarchy, and by appropriately exploiting domain-dependent heuristics.

3 Protection

In this section, we describe a search heuristic that takes advantage of the structure of an abstraction hierarchy. The idea of the heuristic is to protect only those goals that have already been achieved at a higher level of abstraction. This can be viewed as a middle ground between the control strategy of TWEAK, which protects no previously achieved goals, and SNLP, which protects all previously achieved goals.

We first formalize goal protection in planning, where a protected goal is one that has been achieved during the planning process and is neither undone nor reachieved by subsequent plan steps. Then we introduce the idea of monotonic goal protection: protecting only the abstract goals achieved at the higher levels. Finally we prove that the monotonic goal protection heuristic preserves a solution for every planning problem in the search space, for any abstraction hierarchy. Thus, the heuristic can be used by a planner without sacrificing completeness.

3.1 Establishment

Plan construction is a goal-directed process. Given a set of goals, a planner selects a goal, and attempts to find an operator that can achieve it. Because operators have preconditions that need to be satisfied, the preconditions themselves become new goals to achieve, and the planner iterates on these new goals. The need for goal protection arises because an operator for achieving one goal may inadvertently undo or reachieve an already achieved goal. When a goal is protected, no subsequent operators are allowed to be added to a plan that undo or reachieve this goal.

Although goal protection has a long history in planning[Sussman 1973, Tate 1977], an unresolved issue is what set of goals to protect. In this section we propose to protect only a subset of goals achieved so far. For any plan at a certain level of the hierarchy, this subset is defined by the conditions achieved at all levels *above* the current one. The intuition of the idea is clear: since an abstraction hierarchy defines what conditions are important and what conditions are not important, we would like to protect the important conditions already achieved, and still retain the flexibility of varying the unimportant ones. Below, we make the heuristic more precise, and prove that using the heuristic, ABTWEAK does not lose completeness.

In order to formalize goal protection, we must define what it means for a goal (or precondition) to be achieved at different points in the plan. In addition, we must specify precisely how a subsequent operator can undo or reachieve a goal, so as to be able to prevent such occurrences from happening. We do so by presenting the notion of an *establishment*, which specifies that one operator adds an effect satisfying the precondition of a subsequent operator. This is a restatement within the TWEAK representation of Sussman's *ontological structure* [Sussman 1973], Tate's *goal structure* [Tate 1977], and McAllester and Rosenblitt's *causal links* [McAllester-Rosenblitt, 1991].

Suppose that we have a correct plan produced by TWEAK. Then in this plan, every precondition p of every operator must have an operator before it that *asserts* p. More precisely,

Definition 3.1 *Let* Π *be a plan. Operator* α *establishes proposition* p *before operator* β *(*$\mathrm{Establishes}(\alpha, \beta, p)$*) if and only if*

1. $\Box(\alpha \prec \beta)$,
2. $\exists u \in E_\alpha . \Box(u \approx p)$, *and*
3. $\forall \alpha' \in \mathrm{Operators}_\Pi, \forall u' \in E_{\alpha'}$, *if* $(\alpha \prec \alpha' \prec \beta)$, *then* $\neg(u' \approx p)$.

The final condition ensures that α is the last such operator that asserts p.

Notice that in a TWEAK plan the operators are partially ordered. Thus, it is possible that a precondition may have several establishers. For example, consider a plan containing three operators, α_1, α_2 and β. Suppose that both α_1 and α_2 assert p, where p is a precondition of β. Furthermore, α_1 and α_2 both precede β while being unordered themselves. Then both α_1 and α_2 are establishers of p for β. If a precondition has more than one establisher, we call the set of all establishment relations in the plan the *establishment set*[1].

3.2 Threats and Goal Protection

Adapting the definition from [McAllester-Rosenblitt, 1991], we say that an operator c is a *possible threat* to $\mathrm{Establishes}(a, b, p)$ if c is an operator other than a or b that either possibly asserts (a *positive* threat) or possibly denies p (a *negative* threat) and c is possibly between a and b. Thus, given a possible threat, in some completion, c occurs between a and b and asserts or denies p. If c is necessarily between a and b, and either necessarily asserts or necessarily denies p, then c is a necessary threat. By definition, given a necessary threat, in every completion c occurs between a and b and asserts or denies p. Relating this to Chapman's terminology [Chapman 1987], a possible negative threat is also called a *clobber*.

Possible threats can be made *safe* [McAllester-Rosenblitt, 1991] by adding ordering constraints making c be either before a or after b (demotion and promotion, [Chapman 1987]), or by adding non-codesignation constraints making c not assert or deny p (separation). If

[1] A similar notion was defined by Kamphambati[Kambhampati 1992]

a plan is obtained from Π by adding new operators and/or constraints, then it is called a *descendent* plan of Π.

Definition 3.2 (Protection) *Precondition p of β is protected in plan Π if and only if in no descendent plan of Π* every *establishment in the establishment set of p for β is necessarily threatened by an operator.*

In other words, any descendent plan containing an establishment set of a protected goal in which every establishment is necessarily threatened (i.e., a protection *violation*), is pruned from the search tree. We prune these descendents since no existing establishment of the protected goal can be made safe by subsequent planning no matter how many operators and constraints are added.

Whenever a precondition for some operator already in a plan is established, we can view this as progress toward solving the goal. It is thus intuitively appealing to consider that every establishment achieved should be protected from all threats by subsequent planning effort. And in fact, this is the policy employed by McAllester and Rosenblitt in their Systematic Non-Linear Planning algorithm (SNLP) [McAllester-Rosenblitt, 1991]. One of the results of our research has been to challenge the correctness of this view that protecting all establishments is the most beneficial. This challenge is based on the fact that a protected establishment implicitly takes precedence over any subsequent establishments that are added, since these latter establishments are constrained to never violate the protected establishment. Thus, protecting an establishment is equivalent to asserting its greater importance relative to all subsequent establishments. It therefore makes little sense to protect those establishments that might be viewed as trivial aspects of the plan. Unfortunately, this is precisely what McAllester and Rosenblitt's algorithm permits: protection is based upon the *order* in which goals have been achieved, but this ordering is left unspecified, and hence may be arbitrary. ABTWEAK, on the other hand, protects only those goals that have been achieved at the higher levels of abstraction, based upon the criticality function that embeds ordering heuristics. We formalize this in the following section, and return to a comparison of ABTWEAK and SNLP when we discuss the results of experiments performed using both algorithms in Section 7.3.

3.3 Monotonic Protection

As with ABSTRIPS, ABTWEAK performs a length-first search. At each level i, a correct plan Π' at level $i+1$ is taken as input. The process of taking a plan at a higher level and transforming it into a correct one at the current level is called *refinement*. The operators in Π' are converted to their corresponding ones at level i. Then a search is performed to refine Π' to a correct solution at the current level, where the correctness criterion is based on Chapman's MTC[Chapman 1987]. The iteration starts at the highest level with an initial plan $(\mathcal{I}, \mathcal{G})$, and terminates when a correct plan is found. The algorithm can be found in Appendix C.

In AbTweak, we use the abstraction hierarchy as a selection criteria not only for which goals to solve next, but also for which goals to protect. Abstract level goals represent important domain conditions, and effort at the abstract level has already been expended to achieve them. Thus, during refinement, we will protect the abstract level goals.

Definition 3.3 (Monotonic Protection (MP)) *A planner enforces* monotonic protection *if, for every plan* Π *at level* i*, every level* i *precondition of an operator in* Π *is protected during refinement at every lower level.*

A refinement of an abstract plan obtained with the monotonic property enforced is called a *monotonic refinement* [Knoblock, Tenenberg, and Yang 1991]. We use the term monotonic since once an operator or constraint is added at a particular abstract level, it is never removed during refinement at less abstract levels.

Note that monotonic protection does not preserve *every* establishment relation constructed so far; rather, it is guaranteed that at least *one* of the abstract establishments for each precondition is protected. It also means that while planning, no establishment relations *on the current level* are protected.

In the rest of the paper, the term AbTweak *with MP* refers to the abstract planner AbTweak augmented with the addition of monotonic protection at every level. According to the definition, AbTweak with MP prunes all plans with monotonic violations. The question emerges as to whether the resulting planner is still complete. The following theorem answers the question. It states that AbTweak with MP does not lose the completeness of search, and that this property holds regardless of the specific abstraction hierarchy used.

Theorem 3.4 *If a solution plan exists, then a search path to one such plan also exists in the search space of* AbTweak *with MP.*

Proof Sketch: The strategy of our proof is to construct from a lowest level solution a highest level solution that refines using monotonic protection to the lowest level. First, we show that if a solution, Π_0, exists at the lowest level of abstraction, then this plan is also a solution at each higher level. We extend this result to show that the existence of a lowest level solution implies that the justified version of that solution is a correct plan at each higher level, yielding the plan sequence $\Pi_1, \ldots, \Pi_{k-1}$ (for a k-level hierarchy), where justification is a process of removing superfluous operators and constraints. This insures that for all solvable problems, abstract solutions exist, and that these abstract solutions are no longer than the lowest level solutions. Finally, we show that each Π_i is a monotonically protected plan of Π_{i+1}. Thus, if a lowest level solution exists, then there exists an abstract plan in the search space that can be successively refined using weak protection to yield the lowest level solution.

The complete proof is provided in Appendix D. □

Based on this theorem, AbTweak with MP can eventually find a solution if one exists. In the following section, we explore different search control methods for AbTweak with MP.

4 The LEFT-WEDGE Search Control Strategy

So far, we have identified a universal property, the monotonic property, for all abstraction hierarchies. This property guarantees that a sequence of monotonic refinements exists for any hierarchy. However, the way in which one goes about searching for such a sequence is not obvious. Many different search control strategies exist, each resulting from a different way of coordinating search in the abstract plan space and search during plan refinement. In the rest of the paper, we will investigate search control strategies that are both complete and efficient.

The monotonic property states that for solvable problems, there exist abstract solutions that monotonically refine to concrete solutions. Thus, during refinement, the planner need only expand plans which are monotonic refinements of some abstract solution. Unfortunately, not all abstract solutions will monotonically refine, and hence, the planner must search through the space of abstract plans for those that do. In order to ensure completeness, then, the planner must interleave search both length-wise, across alternatives at any given level of abstraction, and depth-wise, through the space of monotonic refinements of any given abstract solution.

An intuitively obvious choice of control is to use a strategy that is complete on each level of abstraction. This is especially appealing, since it is not difficult to specify complete control strategies for TWEAK, either using a complete state-space search procedure such as A^*, breadth-first search, or the procedure provided by Chapman [Chapman 1987]. Using this approach, if a plan is formed on abstraction level i, then it is passed down to the level below. At level $i-1$, all the conditions of criticalities no less than $i-1$ are planned for. The process continues, until either a correct plan is formed at the base level, or it is found that a plan cannot be made correct at a level. Then the planner backtracks to the level immediately above the current one, and tries to find an alternative solution.

The fact that each level is complete may lead one into believing that the above control structure is also complete. Unfortunately, this is not the case in general. The reason is that a complete search strategy for any given level is only guaranteed to find a single solution at that level. But this first solution might not be monotonically refinable. Incompleteness might result if either searching for a refinement never terminates, or the strategy does not search the space of alternative abstract solutions.

ABTWEAK's search strategy does not suffer from these drawbacks, but rather interleaves its effort between expanding *downwards* by refining abstract solutions to lower level ones, and *rightwards* by finding more solutions at each particular level of abstraction. The degree in which a search strategy tends to favor either dimension of growth is an important aspect governing search performance. In Figure 1 the abstract solution search space is shown. This figure shows the relationship between search within an abstract level and search across abstract levels. Each node in the figure represents a solution, that is, a plan, found within a particular abstraction level. It is possible that there may exist multiple solutions within a particular level of abstraction. The leftmost solutions shown in Figure 1 represent the first or "simplest" solutions found within that abstract level. Subsequent, more complex solutions found appear in a left to right fashion. In Figure 1, the solution nodes are labeled such that the abstraction ancestry of that solution can be seen. For example, at level $k-1$, the leftmost node is labeled "1/1", indicating that this is the first level $k-1$ solution found, and is descended from the first level k solution.

One search strategy employed in our experiments is breadth-first search in a search space in which each state corresponds to a plan. This strategy is described in the following section, and the full algorithm is given in Appendix C. During each iteration, a plan Π is selected in order to test its correctness according to the Modal Truth Criterion. If Π is correct at level i, then all operators in Π are replaced by their corresponding $i-1$-level operators (i.e., the level $i-1$ preconditions are added). Otherwise, the plan is modified according to TWEAK's plan modification procedures. The process terminates when a correct plan is found at level-0. The cost of each plan in the search tree is simply the total number of operators in the plan.

Breadth-first search shows no preference for concrete level plans over abstract plans. However, all solutions are at the concrete level, and therefore, effort expended toward exploring alternatives at the abstract level is solely for the sake of completeness. This completeness, however, exacts a heavy computational cost, since computational effort expended in searching at the abstract level is at the expense of further effort to refine plans. We introduce an alternative strategy, which we call LEFT-WEDGE, that allows for plunging more deeply into the search space along the "leftmost" frontier, yet still remaining complete. The motivation for this strategy rests on our intuition that the intent of criticalities is to impose an order on the solution of subgoals. A well chosen abstraction hierarchy would be one in which the choices made at the abstract level serve as fixed constraints throughout the planning, *and never need to be retracted.* Thus, a solution strategy that exploits such a hierarchy would prefer expanding plan refinements over plan alternatives, (downwards to rightwards) under the assumption that correct initial choices of abstract plan steps will rarely require the refinement of abstract alternatives. Thus, for any two plans Π_1 and Π_2 in the search space such that Π_2 is more abstract than Π_1 (see Figure 1), for every plan expansion to Π_2, several more expansions are done for Π_1. Further, as in A^*, preference is given to expanding the shorter of two plans at the same level of abstraction. For example, in Figure 1, in breadth-first search, solution plan 2 at level k may be expanded with the same preference as plan 1/1 and plan 1/2 at level $k-1$. However, in the framework of LEFT-WEDGE, plan 2 at level k is expanded with the same preference as plans 1/1/1 through 1/1/4. In this way, the search space grows deeper much more quickly on the leftmost branches than the right, with the frontier taking on the characteristic left wedge shape for which the strategy is named.

A detailed description of ABTWEAK is given in Appendix C. It is assumed that each plan Π has a certain cost, $cost(\Pi)$. The search strategy used by ABTWEAK always selects a plan Π with minimal cost to refine next. The LEFT-WEDGE heuristic can be implemented by modifying the cost function as follows. For each plan Π at a certain level i, an additional value is subtracted from the cost function, where the value depends on the level of abstraction

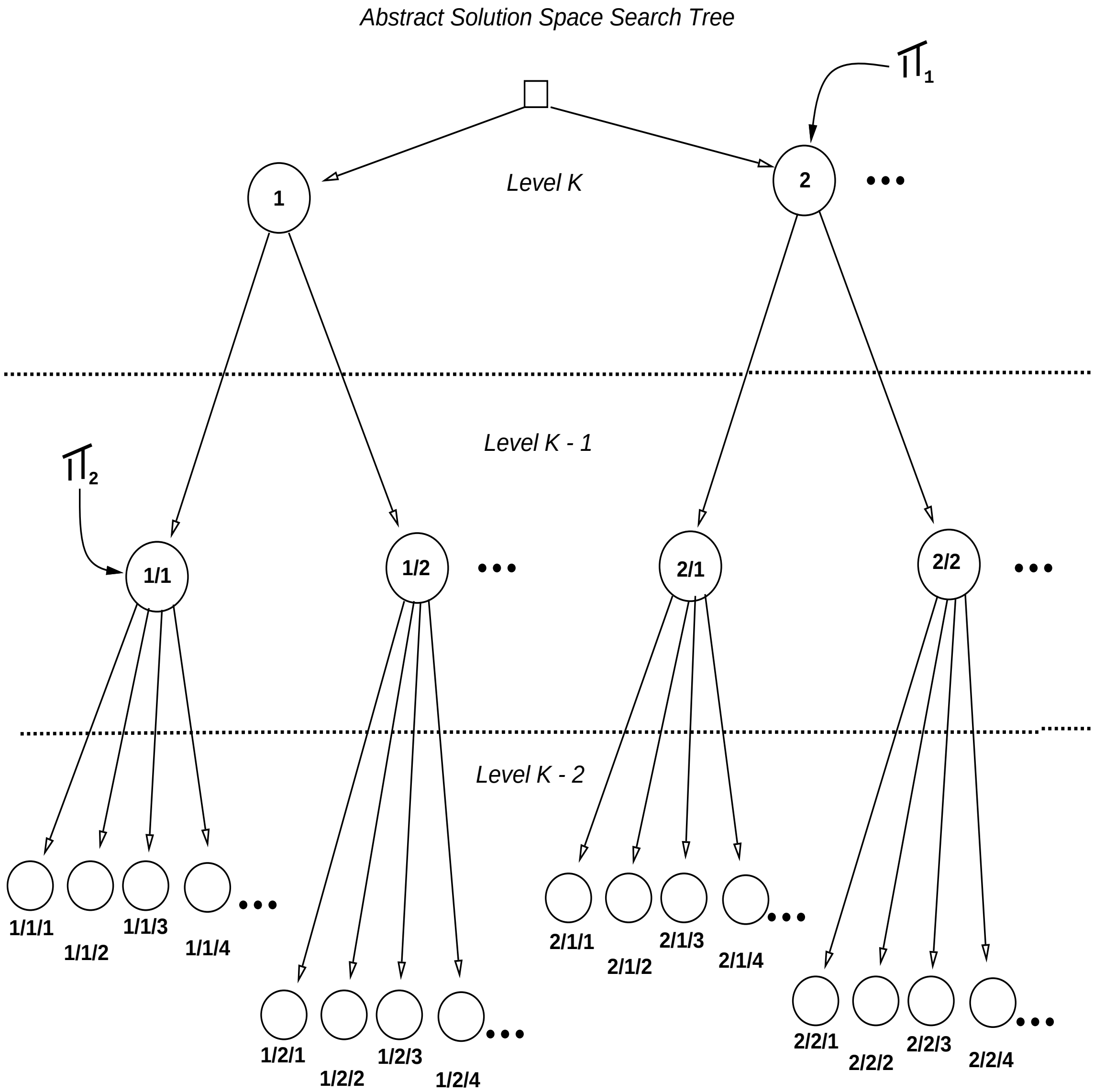

Figure 1: Representing the abstract solution space

i:

$$newcost(\Pi) = cost(\Pi) - lw(crit(\Pi))$$

The function $lw(i)$ is any monotonically decreasing function of i, such that $lw(k-1) = 0$, for a hierarchy with k levels of abstraction. Then search in ABTWEAK is guided by the *newcost* function; a plan with the minimal *newcost* value is always selected next for refinement. Experiments using this strategy, and comparisons with breadth-first search, are described in Section 6.

5 Combining Abstraction with A Domain-Dependent Heuristic

ABTWEAK is a domain-independent planner employing a weak search method. One might, however, wish to employ ABTWEAK in a domain for which there is additional heuristic information available. It is natural to want to incorporate such heuristics into the planner in order to improve its performance. An example is when the domain-dependent heuristic is encoded by making a distinction between *primary effects* and those effects which are not primary for each operator [Sacerdoti 1974, Minton 1990, Knoblock 1991]. For example, in the robot task planning domain to be described, the primary effect of pushing a box from one room to another is just that the box be in the destination room. Any additional effects, such as that the robot is also in destination room are considered secondary. This distinction indicates to the planner that to move the robot around, the push-box operator should *not* be used. It should be used only for moving boxes around, not for any side-effects that might result.

The application of primary effects corresponds to a special type of domain-dependent heuristic that can effectively reduce the branching factor of the search space, since operators are only considered as plan steps when the current subgoal is a primary effect. In fact, without the application of this heuristic, many trivial problems cannot be solved by ABTWEAK.

Unfortunately, as we learned in our experiments, under certain criticality assignments there can be antagonistic interactions between the new heuristic and the underlying length-first search inherent in abstraction which render the planner considerably *less* efficient or even incomplete. In the case with searching only with primary effects, under criticalities that assign low criticalities to primary effects, there might be problems solvable at the lowest level for which no abstract solution exists which is monotonically refinable. Search at the abstract level, therefore, would never terminate. Although we are able to provide restrictions on the criticality mappings that prevent this non-terminating search, this should serve as a caution to those who might want to add other heuristics to an ABTWEAK-style planner. We describe this interaction in more detail in the balance of this section, beginning with a description of the robot task-planning domain, which we will use to illustrate the problem, and which we will return to in Section 6.

In the robot task-planning domain there is a robot that can move between a number of connected rooms. Between any two rooms there may be a door, which can be open or closed. In addition, there are also a number of boxes, which the robot can push from one location to another. Figure 2 shows one configuration of the domain. A list of operators in this domain can be found in Appendix B. Both the representation of the domain, and the operator set in the domain are modified from [Sacerdoti 1974], in order to eliminate the need for axioms.

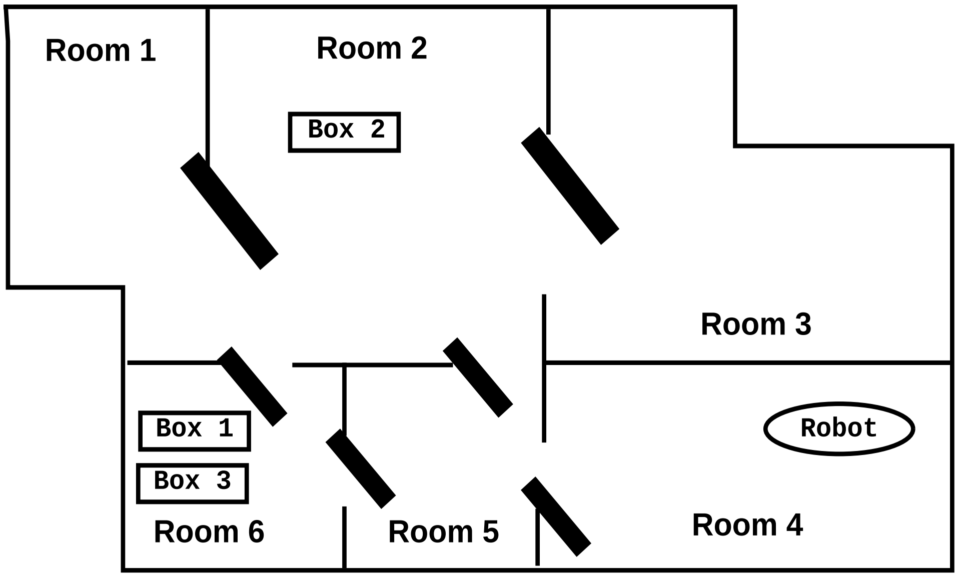

Figure 2: Robot Task Planning Domain.

This domain can be represented by the following predicates: `Box-Inroom`(b, r) representing that box b is inroom r, `Robot-Inroom`(r) representing that the robot is inroom r, `Box-At`(b, loc) representing that box b is at location loc, `Robot-At`(loc) representing that the robot is at location r. `Open`$(door)$ representing that the door, $door$, is open. In addition, there are also a number of predicates denoting the type of object its argument represents (e.g., `IsDoor`, `Pushable`).

For certain hierarchies, the application of the primary effect heuristic threatens the completeness of a planner using monotonic protection. To see the problem more clearly, consider a simplified version of the `push-thru-dr` and `go-thru-dr` operators in Table 1. Suppose that the initial state is as depicted in Figure 2, and the goal is `Box-Inroom`(Box3, Room4).

Now suppose that a hierarchy is built by placing all `Robot-Inroom` literals at a higher level than `Box-Inroom` literals. Then at the `Robot-Inroom` level of abstraction, an abstract plan is built by first inserting the `push-thru-dr` operator to achieve the goal, giving the following partially completed plan

$\mathcal{I}$, `push-thru-dr(Box3, Door5-4, Room5, Room4,` $\ldots$`)`, $\mathcal{G}$.

push-thru-dr (box, door-nm, from-room, to-room, ...)	
Preconditions	Box-Inroom(box,from-room), Robot-Inroom(from-room) ...
Primary-Effects	Box-Inroom(box,to-room),...
Side-Effects	Robot-Inroom(to-room),...

go-thru-dr (door-nm, from-room, to-room, ...)	
Preconditions	Robot-Inroom(from-room),...
Primary-Effects	Robot-Inroom(to-room), ...
Side-Effects	...

Table 1: Operator definition for the robot domain.

The only precondition of `push-thru-dr` visible at this level of abstraction is `Robot-Inroom(Room5)`. Using only primary effects to achieve this subgoal, the planner will complete the plan by adding more `go-thru-dr` operators. An example of a completed abstract plan is:

$\mathcal{I}$, `go-thru-dr(Door5-4, Room4, Room5, ...)`, `push-thru-dr(Box3, Door5-4, Room5, Room4, ...)`, $\mathcal{G}$.

However, this abstract plan cannot be monotonically refined at the `Box-Inroom` level, since any refinement would involve moving the robot into and out of Room 6 to fetch Box 3, and this would violate abstract establishment relations for the robot's position. One can check that no other abstract plans can be refined either. Thus, monotonic protection leads to incompleteness in this example.

Fortunately, there is one hierarchy in which both the upward-solution property and the monotonic property are satisfied. In this hierarchy, all the `Box-Inroom` preconditions are placed above the `Robot-Inroom` ones. Similarly, all the `Box-At` preconditions are above the `Robot-At` ones. One always first plans the location of the box before the location of the robot. In this case, the abstract plan

$\mathcal{I}$, `push-thru-dr(Box3, Door6-5, Room4, Room5, ...)`, `push-thru-dr(Box3, Door5-4, Room5, Room4, ...)`, $\mathcal{G}$.

can be monotonically refined to a solution.

Thus, applying domain-dependent heuristics such as the primary effects is a tricky matter in terms of the completeness of an abstract planner. The fact that the above hierarchy is complete even with the use of primary effects seems to be a fortunate coincidence. However, there is a deeper reason in this seemingly *ad hoc* engineering, in that a sufficient condition on the criticality assignments exists which guarantees the completeness of a hierarchical planner. The condition is that, for each operator, to assign primary effects a criticality at least as great as the criticality for the secondary effects.

Condition 5.1 *Let O be the operator set in a domain. $\forall\alpha \in O, e_1, e_2 \in E_\alpha$, if e_1 is a primary effect, and e_2 is not, then* $\text{crit}(e_1) \geq \text{crit}(e_2)$.

In our robot task planning domain, the **move-box** operator changes both the location of the robot, and the location of the box. However, only `Box-Inroom` is a primary effect. The above condition restricts that the criticality of `Box-Inroom` be no lower than the criticality of `Robot-Inroom`. Note also that the reverse hierarchy does not satisfy this condition.

This condition guarantees the following theorem:

Theorem 5.2 (Primary Effect Theorem) *Suppose that an abstraction hierarchy satisfies Condition 5.1. If* TWEAK *with primary effects can find a solution plan to a problem, then using the hierarchy,* ABTWEAK *with MP and primary effects can also find a solution to the problem.*

The theorem is sufficient for verifying the completeness of an abstract planner under a given abstraction hierarchy. For example, the hierarchy described in Table 9 can be shown to ensure that the abstract planner remains complete with the application of primary effects.

6 Experiments

Above we have described the monotonic property for search control within a level of abstraction, and LEFT-WEDGE as a control strategy for search across multiple levels of abstraction. While we are able to show that both methods guarantee completeness for ABTWEAK, it is difficult, if not impossible, to conduct a theoretical analysis of their effectiveness in search reduction. An alternative then, is to test ABTWEAK empirically.

Both ABTWEAK and TWEAK have been implemented in Allegro Common Lisp, on a Sun SparcStation II. A detailed explanation of the implementation can be found in [Woods 1991]. In the implementation, we have paid special attention in making sure that the two planners share key subroutines, so the comparison in their performances can be fair. We have also conducted experiments in two domains, the Towers of Hanoi domain, and a robot task planning domain from ABSTRIPS[Sacerdoti 1974]. The full operator descriptions for the Towers of Hanoi domain appear in Appendix A. Appendix B lists the operators and language used in the robot task planning domain.

In the following section, we discuss the results from each domain in turn. In doing so, we pay special attention to the following issues concerning search reduction:

1. Investigating the usefulness of enforcing the monotonic property under breadth-first and LEFT-WEDGE search.

2. Investigating the usefulness of the LEFT-WEDGE control strategy as compared to uninformed search strategies such as breadth-first search. Of special importance are the types of hierarchies with which a LEFT-WEDGE search is expected to gain a great amount of search reduction.

3. Studying how best to combine domain-dependent heuristics encoded in terms of primary effects, and abstraction as used in ABTWEAK. As we have pointed out earlier, an abstract planning system retains its efficiency as long as the criticality function assigns the primary effects of each operator to be at least as great as the non-primary effects of that operator. Thus, we would like to compare ABTWEAK using LEFT-WEDGE on such a criticality function, to both TWEAK and ABTWEAK using breadth-first search in the robot task-planning domain.

6.1 Testing the Towers of Hanoi Domain

For the first set of experiments, we have chosen to use the Towers of Hanoi (TOH) domain. This domain is particularly interesting for testing hierarchical problem solving for several reasons. First, there is a clear distinction of different degrees of "importance" among the domain conditions. In particular, the position of the large disk is the hardest to change since it would require that both the medium and the small disks be at appropriate positions first. Second, the problem is considered hard to solve by many non-hierarchical planners. This is in part due to the recursive nature of the domain, where a subgoal can back-chain on itself. Third, there is potential for a large number of goal interactions in the planning process, since many operators add and delete many subgoals. The problem also involves the achievement of conjunctive goals, many of which interact.

In the Towers of Hanoi, 3-disk domain, four predicates are used to describe the states. These are `IsPeg, OnSmall, OnMedium, OnLarge`. If a hierarchy is built based on assigning a distinct criticality value to each of the predicates, then 24 different hierarchies exist. Out of the 24 hierarchies, only one has been extensively tested in the past with linear, abstract planners [Knoblock 1991]. This well-tested hierarchy corresponds to assigning criticality values in the following way: $crit(\texttt{ISPEG}) = 3$, $crit(\texttt{OnLarge}) = 2$, $crit(\texttt{OnMedium}) = 1$, $crit(\texttt{OnSmall}) = 0$. In order to fully investigate the effects of different control strategies on search efficiency as a function of the hierarchy used, we have tested all possible permutations of the hierarchies. For ease of exposition, we use ILMS to represent the above hierarchy. Similarly, SMLI represents the hierarchy with the reverse order of criticality assignment.

Experimental results in this domain can be divided into two categories: those demonstrating the usefulness of the monotonic property in restricting search and those comparing the LEFT-WEDGE and breadth-first search strategies. Performance results, in the number of state expansions as well as CPU seconds for finding a solution, as a function of the hierarchy used, are shown in the summary Figures 9 to 12 in Appendix F.

6.1.1 Testing Monotonic Goal Protection

In Appendix F, we summarize the results of testing the usefulness of monotonic protection. In the appendix, Figure 9 shows the performance results of ABTWEAK using a breadth-first strategy with monotonic protection (MP), while Figure 10 shows those without MP.

Hierarchies	Expanded		CPU seconds	
	With MP	Without MP	With MP	Without MP
IMLS	166	550	60.4	213.6
IMSL	652	918	421.4	548.1
ILSM	765	1112	540.2	730.7
ISLM	1083	1771	929.8	1224.5

Table 2: Extracted data comparing AbTweak with and without MP using Breadth-first search.

Hierarchies	Expanded		CPU seconds	
	With MP	Without MP	With MP	Without MP
ILMS	471	471	252.2	218.4
LIMS	609	609	677.1	531.2
LMIS	1717	1717	1853.9	1668.8
LMSI	1894	1894	2690.8	2111.4

Table 3: Comparing AbTweak with and without MP using Breadth-first search. The data show cases where there are no monotonic violations.

Overall, breadth-first search with MP outperforms search without using MP, with respect to CPU time, in 15 out of 24 cases of the criticality permutation. Table 2 displays a few cases extracted from the two tables where the conclusion is clear. In terms of the total number of states expanded, using AbTweak with MP is no worse than AbTweak without MP in 21 out of 24 cases. The reason why AbTweak using MP often outperforms AbTweak without MP can be attributed to the fact that enforcing the monotonic protection amounts to the protection of all abstract establishments. As a consequence, during refinement no operators are added that violate the establishments. This reduces the branching factor of search.

On the other hand, there are also cases where applying MP significantly reduces search efficiency. The first class of such cases occurs with hierarchies for which no monotonic violation can occur (see Table 3). For example, with hierarchies where OnLarge is above OnMedium, and OnMedium is above OnSmall, no monotonic violation exists in the search space. Thus, protecting MP would waste an extra amount of CPU time because of the overhead of checking for violations. As a result, AbTweak with MP is more costly than AbTweak without MP. The second class of such situations occurs when intuitively "bad" hierarchies are used (see Table 4). For example, in two cases where OnSmall is above OnMedium and which in turn is above OnLarge, using MP slows down the search. This effect confirms a general

Hierarchies	Expanded		CPU seconds	
	With MP	Without MP	With MP	Without MP
ISML	over 5000	3142	over 9579.8	3333.5
SMLI	2519	2359	4987.9	2892.1

Table 4: Comparing ABTWEAK with and without MP, both using breadth-first search. The data show cases where "bad" hierarchies are used.

principle which applies not only to abstract planning, but to planning without abstraction as well: that protection of establishments only improves search efficiency when the difficult-to-achieve conditions are protected; otherwise protection will instead reduce search efficiency. For example, with the hierarchy ISML, using MP amounts to protecting all OnSmall conditions. However, it is the OnLarge conditions that are more difficult to achieve. Thus, with ISML the wrong conditions are protected, resulting in an increase in the amount of search required. In terms of search space, this phenomenon can be easily explained, as follows. Applying protection of establishments cuts off the branching factor of search, although it guarantees completeness of a planner by always making sure that at least one path is retained which leads to a goal. When protecting unimportant conditions such as OnSmall, many paths that can lead to goals with much shorter *solution length* in the search space are also cut off. As a result, while the branching factor is reduced, the depth of search is enlarged. The end result is that a planner has to search many more states to find a goal.

We are thus making the claim that a goal should only be protected from other goals that are considered less important, but should not be protected from goals that are at the same or higher levels of importance. Our earlier results in the Towers of Hanoi domain provide strong evidence for this position, since the abstraction hierarchies that performed most poorly were those that resulted in the greatest number of protection violations. These experiments demonstrate that a poor choice in subgoal ordering, when coupled with protection based upon this ordering, result in a large number of protection violations, and hence, a great deal of backtracking leading to poor performance.

6.1.2 The Placement of Object-type Predicates

Many predicates and operators in a particular domain apply only to certain objects, by their very nature. For example, in a blocks world domain in which unbound variables can either be a table or a block, an operator such as `Pickup` may only apply to blocks, but not tables. The way in which the application of these operators are constrained is through preconditions that identify the object-type of each variable in an operator representation. As another example, the `IsPeg` predicate in Towers of Hanoi is used to ensure that only the three pegs can be used to hold the disks, and that no pegs can be moved around. These object-type predicates are constraints on the possible bindings of variables during the search process. If we postpone

Hierarchies	Expanded	CPU seconds
IMLS	166	60.4
MILS	295	235.9
MLIS	313	304.8
MLSI	382	420.4

Table 5: Testing the effect of the placement of object-type predicates.

the constraint of these variables during planning, we often increase the branching factor by allowing operator instances in our search space which can never be satisfied, such as `Pickup( table )`. Thus, it is desirable to satisfy these object-type predicates *early* in the search. In other words, during abstract search, it is more desirable to assign higher criticality values to them.

Some of the test results showing the effect of object-type predicates on search are extracted from Figures 9 and 10, and are shown in Table 5. From the table, we see that when using the MP, the criticality assigned to object-type predicates such as `IsPeg` has a noticeable effect on search efficiency, giving evidence that placing `ISPeg` types of predicates at the highest level of abstraction reduces search. This further validates Sacerdoti's criticality heuristic in ABSTRIPS[Sacerdoti 1974] where these predicates are placed at the highest level, since there exist no plans that can ever change the truth value of any literal containing an object-type predicate. For example, if some object is not a peg in the initial state, it will never be a peg in any subsequent state.

6.1.3 Testing the LEFT-WEDGE Control Strategy

Figures 9 and 11 in Appendix F provide a comparison of ABTWEAK using breadth-first and LEFT-WEDGE, both with MP. It is evident from the two tables that search time and space is greatly reduced when using the LEFT-WEDGE strategy, for hierarchies with a small number of monotonic violations. In Table 6, we extract a portion of the test results to demonstrate this. However, no improvement, or even a decrease in performance is seen for certain other criticality assignments, notably IMSL and SMLI(see Table 7) This tells us that the LEFT-WEDGE strategy should be used only with good abstraction hierarchies. If one is not sure about the quality of a hierarchy, then a breadth-first strategy should instead be adopted.

When comparing LEFT-WEDGE with and without using MP (Figures 11 and 12), the results indicate that, in general, using MP with LEFT-WEDGE works well with good criticalities (those which generally result in few monotonic violations), and poorly with bad criticality assignments (those resulting in many monotonic violations). Again, test results are extracted and shown in Table 8 to demonstrate this fact. This result again confirms our conclusion above. The table shows that ABTWEAK with MP clearly outperforms ABTWEAK

Hierarchies	Expanded		CPU seconds	
	LEFT-WEDGE	Breadth-first	LEFT-WEDGE	Breadth-first
ILMS	57	471	38.2	252.2
IMLS	86	166	38.1	60.4
LIMS	56	609	79.1	677.1
MILS	94	295	81.3	235.9

Table 6: Comparing ABTWEAK with and without LEFT-WEDGE, both using MP.

Hierarchies	Expanded		CPU seconds	
	LEFT-WEDGE	Breadth-first	LEFT-WEDGE	Breadth-first
IMSL	3904	652	4971.6	421.4
SMLI	over 5000	2519	over 10773.0	4987.9

Table 7: Comparing ABTWEAK with MP using LEFT-WEDGE and without using LEFT-WEDGE search, under "bad" hierarchies.

Hierarchies	Expanded		CPU seconds	
	With MP	Without MP	With MP	Without MP
IMLS	86	1009	38.1	1470.5
MLIS	73	989	126.4	2474.5
ISLM	over 5000	168	over 15491.3	214.9
ISMB	over 5000	963	over 12688.5	1459.0

Table 8: Comparing ABTWEAK with and without MP, both using LEFT-WEDGE.

without MP in hierarchies IMLS and MLIS. However, for hierarchies ISLM and ISML it appears that not using MP is considerably better. This result is hardly surprising, if one takes into account the depth-first nature of the LEFT-WEDGE strategy. For the hierarchies ISLM and ISML, the abstract versions of the concrete level solutions at the `OnSmall` level (level-2) correspond to the fourth alternative correct solution on that level.

A LEFT-WEDGE search with MP will commit to the first several abstract solutions at the `OnSmall` level, although none of these solutions can be refined to a final solution without monotonic violation. As a result, for such poorly chosen abstraction hierarchies, a strategy that does not protect the abstract goal achievement works best, since it is able to undo poor choices made early in the planning process without having to backtrack up abstraction levels.

6.1.4 Comparing TWEAK with ABTWEAK

The utility of abstract search will not be completely understood without also comparing it to search without abstraction. We have implemented the planner TWEAK, a description of which can be found in Appendix C. To ensure fairness in comparison, the two planners are implemented sharing all key subroutines such as state expansion and unification. In essence, one can view TWEAK as an ABTWEAK system with only a single level of abstraction. Thus, abstraction is neither used as a selection mechanism, since all subgoals are at the same level, nor is it used for protection, since there are no higher level establishments.

Figure 6.1.4 shows the result of the comparison in the Towers of Hanoi domain. In this test, ABTWEAK was run with MP and the LEFT-WEDGE control strategy, in the hierarchy ILMS. The figure contrasts TWEAK with ABTWEAK, in terms of the number of states expanded as a function of the solution lengths. The data in the figure are generated and averaged based on planning with a fixed initial state, and 26 different goal states in this domain. It is clear that ABTWEAK dramatically outperforms TWEAK when the solution length increases.

The same figure also compares the performance of the two planners, but using a poorly chosen criticality assignment, namely ISML. The result is that with this hierarchy, ABTWEAK using both MP and LEFT-WEDGE performs the worst. This result leads us to the conclusion that an arbitrary abstraction hierarchy is not necessarily good. To improve performance using abstraction, one has to be very careful in the choice of both the abstraction hierarchy and the search strategies guiding the abstract search. This result serves as a strong motivation for much of the current research in finding syntactic criteria for good abstraction hierarchies. Examples of such current work can be found in [Knoblock, Tenenberg, and Yang 1991] and [Knoblock 1991].

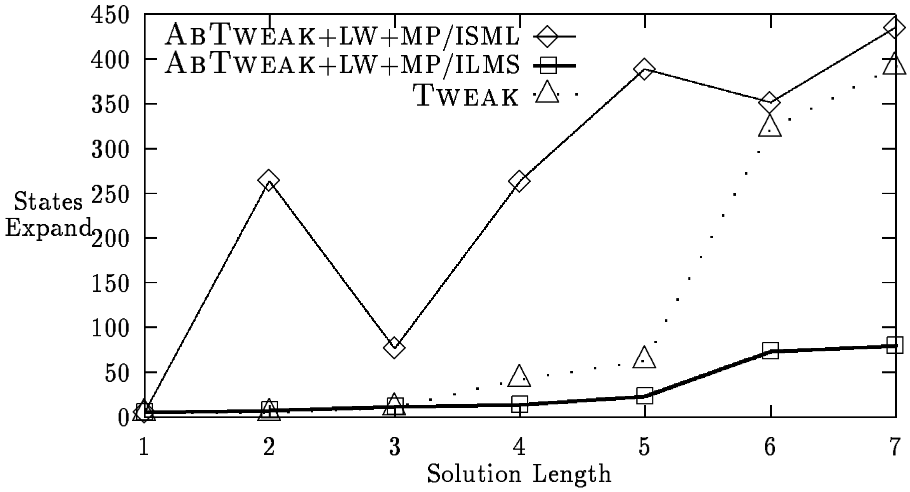

Figure 3: Comparing Tweak with AbTweak.

6.2 Robot Task Planning Domain

We have run 50 tests of AbTweak with the hierarchy in Table 9 using the primary effects heuristic. Without this heuristic, many simple problems were not solvable with the given time bounds by any of the planners that we tested. Five different planning problems of each length were solved using Tweak, AbTweak with breadth-first, and AbTweak with both MP and the Left-Wedge control strategy. Both planners in this domain used primary-effects as a domain-dependent heuristic to restrict the branching factor of search. Figure 6.2 shows the number of states expanded as a function of solution length. It is clear that AbTweak with MP and the Left-Wedge control strategy dramatically outperforms both Tweak and AbTweak with only the breadth-first control strategy.

7 Relation to Other Work

In this section, we discuss how our work is compared to other closely related efforts in planning and abstraction.

Criticality	Predicate
4	`Box-Inroom` and other sort-type predicates.
3	`Robot-Inroom`
2	`Box-At`
1	`Robot-At`
0	`Open`

Table 9: Criticality assignments for the Robot Task Planning Domain.

7.1 Relationship to Planners Based on Hierarchical Task Networks

We consider AbTweak as a clean theoretical and experimental tool to study abstraction and search. As such, it can be useful in justifying many abstract planning methods used in practical planners, as well as in predicting the effectiveness of various abstract search heuristics. To demonstrate the validity of this claim, in this section we compare AbTweak with planners based on hierarchical task networks (HTN). To focus our attention, we consider Sipe[Wilkins 1984], a representative HTN planning system. The main results of the comparison can be summarized as follows:

1. Sipe performs two types of goal protection, the protection of abstract achievements and the protection of certain conditions at each planning level. The former corresponds directly to the monotonic protection used in AbTweak. It follows form our analysis that this method of protection is complete.

2. To improve search efficiency, Sipe makes use of primary effects (called *main effects*) to prune the search space. Our analysis of the relationship between primary effects and abstraction hierarchies in AbTweak points out that, to maintain completeness and search efficiency, Sipe needs to place all primary effects at the same or higher levels of abstraction than the side effects of operators.

3. Finally, the Left-Wedge search control method used in AbTweak can be effectively applied to Sipe as well. Our analysis shows that with the application of Left-Wedge, Sipe is guaranteed to be find a solution if one exists. In addition, with a good abstraction hierarchy, Sipe can perform almost as efficiently as a depth-first search.

Below, we discuss each point in turn.

7.1.1 Mapping between Sipe and AbTweak

In this section, we draw a connection between Sipe and AbTweak. Sipe represents its actions as *schemas*. Each schema has a set of preconditions and effects, just like an AbTweak

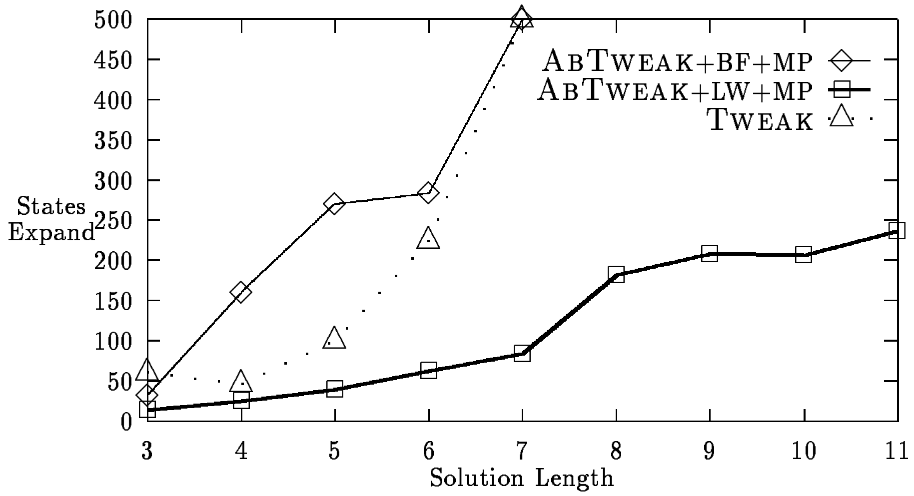

Figure 4: Comparing TWEAK with ABTWEAK in the robot task planning domain.

operator. Unlike ABTWEAK, each schema also specifies other information useful for planning. An important piece of information specifies how one action can be realized through many subactions. For example, a schema for fetching an object may consist of the subactions for sensing and picking up the object. The subactions are associated with the ordering constraints and the binding constraints imposed on variables. In addition, an action schema specifies the range of conditions in which they are to be protected during planning.

As pointed out by Wilkins, the search component of SIPE performs two types of planning activities. SIPE either expands an action node by replacing it by its subactions at a different level of detail, or it inserts subactions at the same level of detail. The former is an expansion across different *abstraction levels*, while the latter across different *planning levels*. As an example, consider planning to get a robot from one room to another. Suppose the planner already has a plan for crossing different rooms. If lower level conditions, such as changing the location of the robot within a room, are introduced into the plan, then the plan is expanded across abstraction levels. However, the process of inserting more operators to go from one location to the next in the same room is plan expansion at different planning levels.

Comparing SIPE to ABTWEAK, we see a striking similarity in their search behavior. Although ABTWEAK does not use action schemas to expand a plan, it implicitly accomplishes the process of expansion via planning from the first principles. In other words, the plan expansion process of SIPE at different planning levels is paralleled by the plan refinement process of ABTWEAK within a given abstraction level. Both systems work at different levels of abstraction, and compose plans in a top-down manner. Furthermore, both systems protect abstract level goal conditions during planning, although the protection is done under different names — In ABTWEAK an interval of protection is called a *establishment relation*, while in SIPE it is known as a *protect-until* slot.

Based on the above analysis of their similarities, we next discuss how various knowledge gained from ABTWEAK can be useful for SIPE.

7.1.2 Monotonic Protection

The action schemas in SIPE specify a set of *protect-until* slots. Associated with each slot is a condition to be protected during planning, and each slot specifies the range where the condition is to be maintained. As we noted above, it is not hard to see that the protect-until construct of SIPE is very similar to the *establishment relations* in ABTWEAK. Both systems use this information to control search. In particular, SIPE periodically applies a set of verification routines, known as *Critics*, to a plan. If any of the protected conditions is found to be violated by the Critics, then SIPE prunes that plan from the search space. In essence, this is exactly the same operation as the monotonic protection used in ABTWEAK.

Having established the precise correspondence between the methods used by both systems for protecting abstract conditions, it follows from the completeness of monotonic protection in ABTWEAK that the protection method used in SIPE is *complete*. In other words, SIPE will not lose any potential solution by performing its protection on abstract conditions. Although intuitive, this completeness result does not follow from the completeness property of other planners such as SNLP, which protects everything achieved so far, and to the best of our knowledge, it has never been formally established before for SIPE.

7.1.3 Primary Effects and Abstraction Hierarchies

To improve its search efficiency, SIPE specifies the main purpose of each action schema, in a manner similar to the primary effect heuristic used in ABTWEAK. In particular, each action schema has one or more main *purposes*, which are the primary effects of the action. During planning, the schema is inserted in a plan only for its stated purpose. All other effects are derived through a special causal reasoner and domain axioms; these being the *side* effects of the action. The goal of restricting the effects of actions in this way, is exactly the same as using primary effects in ABTWEAK: they are both for cutting down the branching factor of search, and reducing the total amount of time used in reasoning about side effects.

Our analysis in ABTWEAK has shown that using primary effects together with an abstraction hierarchy is a tricky matter. Unless all primary effects have criticality values no lower than their side effects, an abstraction planner is likely to perform worse than without using abstraction, or even become incomplete. The same conclusion should also apply to SIPE, by virtual of their similarity in search. This fact places a constraint on SIPE users; when the action schemas are defined, care should be taken to place all conditions stated as purposes of actions at an abstract level no lower than the side effects.

7.1.4 Left-Wedge Search

To maintain its heuristic adequacy, SIPE uses a depth-first search method for selecting the next plan to expand. A problem inherent in the depth-first search is its inability to always find a solution for a solvable problem. In other words, SIPE using a depth-first search may be incomplete. The LEFT-WEDGE method used in ABTWEAK addresses this incompleteness problem. Our theoretical and empirical analyses of LEFT-WEDGE in the context of ABTWEAK predict that if LEFT-WEDGE were used in SIPE, the incompleteness problem of SIPE could be overcome without the loss of efficiency gain from depth-first search.

7.2 The Ordered Monotonic Property

A closely related work to ABTWEAK is Knoblock's ALPINE system [Knoblock 1991], which is an extension of Sacerdoti's ABSTRIPS and Siklossy and Dreussi's LAWLY[Siklossy and Dreussi 1973]. ALPINE differs from ABTWEAK in two significant ways. First, ALPINE is a strictly *linear* planning system, and has no capability for nonlinear planning. Second, Knoblock's focus with ALPINE was in automating the generation of abstraction hierarchies, and not on how to make efficient use of a given hierarchy, as was our intent.

In ALPINE, all generated hierarchies satisfy the *ordered monotonicity* property (OM), which is defined in [Knoblock 1991, Knoblock, Tenenberg, and Yang 1991] roughly as

> *Every* refinement of an abstract plan leaves *all* high-level literals unchanged.

This implies that for any OM hierarchy, it is impossible to generate monotonic violations of abstract preconditions during refinement at lower levels. This property can be guaranteed by a set of syntactic conditions that relate the operator schemas to the literals in the domain language. The syntactic conditions can then be used in the design of an algorithm that generates abstraction hierarchies possessing the OM property [Knoblock, Tenenberg, and Yang 1991]. For example, in the Towers of Hanoi domain, the hierarchy ILMS is ordered. Experiments reported in [Knoblock 1991] demonstrate that in several domains, planning with the abstraction hierarchy generated by ALPINE clearly improves planning efficiency.

Range of Applicability

In comparing ALPINE's ordered monotonicity (OM) property and the monotonic property (MP) method used in ABTWEAK, we note that OM is much stronger than MP, and thus is satisfied by fewer domains. In fact, OM requires that a refinement leave intact *all* higher-level literals, even those that are not part of the abstract plan being refined. Furthermore, this restriction must hold for *every* refinement. In many cases, the OM property is so strong that it can only be satisfied by trivial hierarchies, i.e., the hierarchy often collapses to a single level. In contrast, MP can be applied to every hierarchy, whether they satisfy OM or not. In addition, MP as defined by ABTWEAK can affect higher-level literals, just as long as it does not affect the higher-level literals appearing in the particular abstract plan being refined.

Completeness

If a hierarchy satisfies OM, then a solution path exists on which no violation of high level conditions can occur. However, this does not imply that every search path in the abstract search space leads to a solution. If a search path does not lead to a solution, then backtracking has to be invoked in order to find the solution. However, as we pointed out in Section 4, refinement of an abstract solution may take forever without reaching either a dead end or a solution. Therefore, even if a hierarchy satisfies the OM property, performing a depth-first search across abstraction levels still leads to incompleteness. Likewise, both ABSTRIPS and LAWLY are likely to be incomplete because they also perform depth-first search across abstraction levels. One way to solve the incompleteness problem is to apply the LEFT-WEDGE search method used in ABTWEAK. In this way, completeness is retained and search can be made more efficient than a blind breadth-first method.

Experimental Validation

Our experimental results in the TOH domain provide additional validation that abstraction hierarchies satisfying OM yield performance improvements over abstraction hierarchies not satisfying OM. These results were obtained by counting the total number of monotonic protection violations for each abstraction hierarchy in the TOH domain, and comparing this with the amount of CPU time required by that hierarchy to solve the test problems. Figure 5 and Figure 6 display the CPU times as a function of the number of monotonic violations in each experiment using LEFT-WEDGE search. The resulting correlation coefficient for CPU time and number of violations is 0.8. We can thus see a general rule emerging from the results in the figure, that the fewer the number of monotonic violations, the better the performance in search with an abstraction hierarchy. In particular, ILMS and IMLS are both the best hierarchies.

Hence, a good abstraction hierarchy is one with few monotonic violations. In particular, for the hierarchy ILMS, which satisfies the OM property, no monotonic violations will ever occur during plan refinement[Knoblock, Tenenberg, and Yang 1991]. These results provide additional support for Knoblock's claim that OM generates good abstraction hierarchies.

7.3 Goal Protection in SNLP

We now consider ABTWEAK's role as a goal-protection planner. We start by reviewing the different protection methods used in the past. In classical planning, goal protection has had an important role. Sussman's HACKER system [Sussman 1973] is among the first planning systems which protect subgoals. Subsequently, Waldinger's planner [Waldinger 1977] and Warren's WARPLAN[Warren 1974] both adopt an incremental planning approach, by

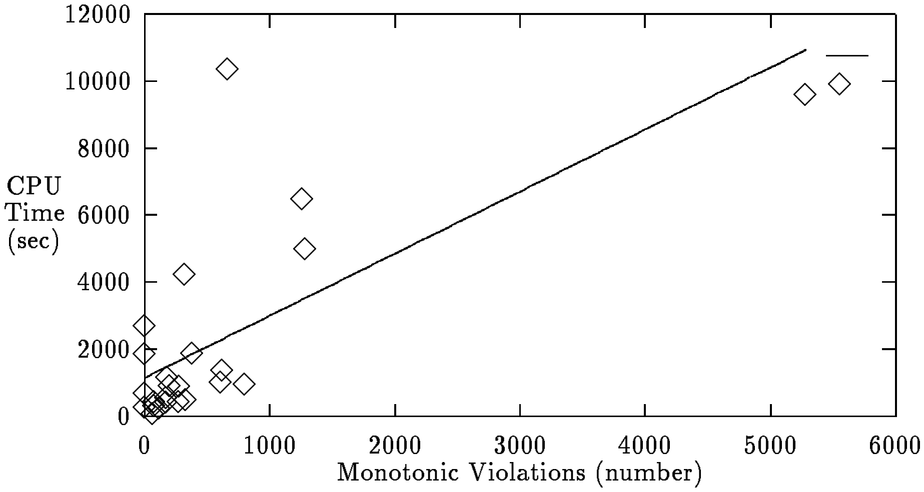

Figure 5: Violations versus CPU time, including regression fit curve. Data are obtained using the breadth-first strategy with MP.

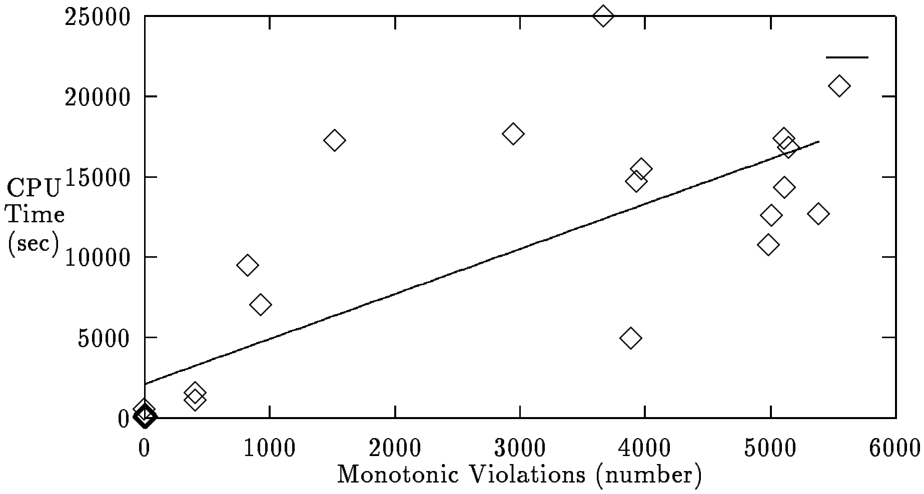

Figure 6: Violations versus CPU time, including regression fit curve. Data are obtained using the left-wedge strategy with MP.

first ordering a conjunctive set of goals, and then achieving the goals one at a time in sequence. When solving the next subgoal g_{i+1} in a sequence of subgoals $g_1, g_2, \ldots, g_n$, the set of establishment relations in the plan for solving $g_1, g_2, \ldots, g_i$ are recorded and preserved. No new operator inserted into the plan is allowed to change any of the recorded relations. The goal protection method is more fully developed with planning systems such as NOAH[Sacerdoti 1977], NONLIN[Tate 1977] and SIPE[Wilkins 1984], which introduce constraint posting as methods of resolving a conflict with goal achievement. Following the development of ABTWEAK, [Yang, Tenenberg, and Woods 1991, Woods 1991] McAllester and Rosenblitt [McAllester-Rosenblitt, 1991] developed a planning algorithm for a nonlinear planner that uses goal protection as the central search principle.

SNLP's protection policy differs from ABTWEAK's in two ways. First, an establishment is protected in SNLP by protecting against *possible* threats, as opposed to ABTWEAK, which protects against necessary threats. Second, in SNLP, *every* established precondition is protected during subsequent planning, as opposed to ABTWEAK, which protects only abstract level establishments[2]. ABTWEAK's weaker protection strategy avoids over-committing to the achievement of goals that are considered less important than the rest, thus achieving a level of flexibility not attained by previous planners that employ protection which, when coupled with good abstraction hierarchies, leads to improved performance.

To back up our claim, we have experimentally compared the performance of TWEAK, ABTWEAK, SNLP, and an abstract version of SNLP which we call ABSNLP. ABSNLP is our own hybrid invention, and is identical to SNLP in regards to its protection policy (protect everything achieved so far), but uses an abstraction hierarchy for goal selection. The reason for defining ABSNLP is to attempt to determine if performance differences between SNLP and ABTWEAK are as a result of differences in goal selection policies, or goal protection policies. These four planners are summarized according to their selection and protection characteristics in Table 10.

Planner	Selection	Protection
TWEAK	No abstraction	No
ABTWEAK	abstraction	Abstract establishments
SNLP	No abstraction	All establishments
ABSNLP	abstraction	All establishments

Table 10: Comparison of Planners by Selection and Protection Policy

For the experiment, each planner was run on the same 54 problems in the three-disk Tower of Hanoi domain. The results are averaged over the different solution lengths, which

[2]This is also one of the major differences between ABTWEAK and Kamphambati's multi-contributor partial-order planner[Kambhampati 1992].

range from 0 to 7. Figures 7 and 8 show the average CPU time and average number of nodes expanded for the different size problems[3]. Note that the y-axes of the graphs are scaled logarithmically. The CPU time graph provides a performance evaluation of the planners in terms of time complexity, while the expanded nodes graph indicates the amount of back-tracking that occurred during search. The abstraction hierarchy used is ILMS. As shown in the graphs, the complete goal protection advocated by McAllester and Rosenblitt performed much worse than no goal protection, and the abstract form of goal protection, implemented in ABTWEAK, provided moderate improvements in both CPU time and nodes expanded. Also note that ABSNLP performs worse than ABTWEAK in most cases, even though both are using the same abstraction hierarchy.

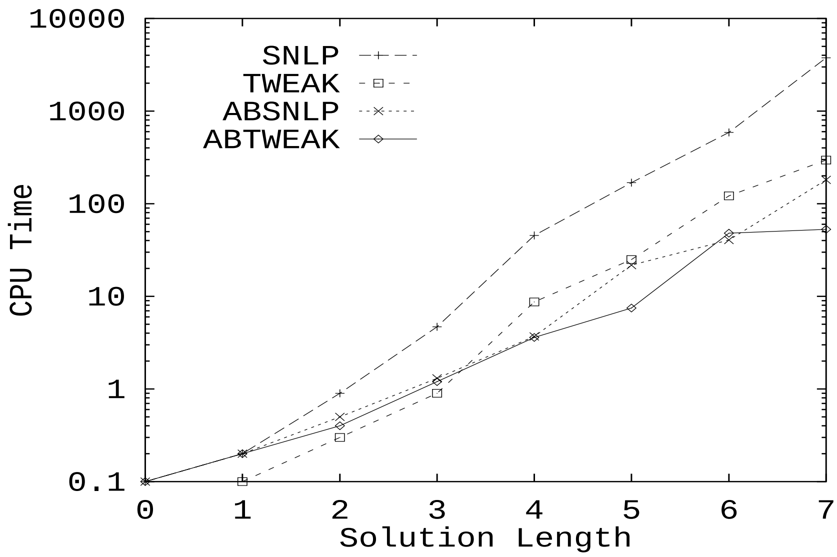

Figure 7: CPU time comparison of four planners.

The reason why SNLP performs worse than TWEAK may appear counter-intuitive, but from the point of view of goal protection it has a simple explanation. In the Tower of Hanoi, if the location of the smallest disk is protected, then it would be harder or even impossible to move any other disks. Thus, a premature protection of the movement of the smallest disk causes more backtracking than not protecting anything at all. As a consequence, more search is required for SNLP than for TWEAK.

The difference in performance between ABSNLP and ABTWEAK highlights the difference between different methods of using an abstraction hierarchy. ABTWEAK, which only protects abstract establishments, performs better in most cases than ABSNLP, which protects all establishments in the current level as well, and uses the stronger form of protection detailed above. This implies that it is not only important for a planner to order the achievement of

[3]We thank Craig Knoblock for his help in processing the experimental data.

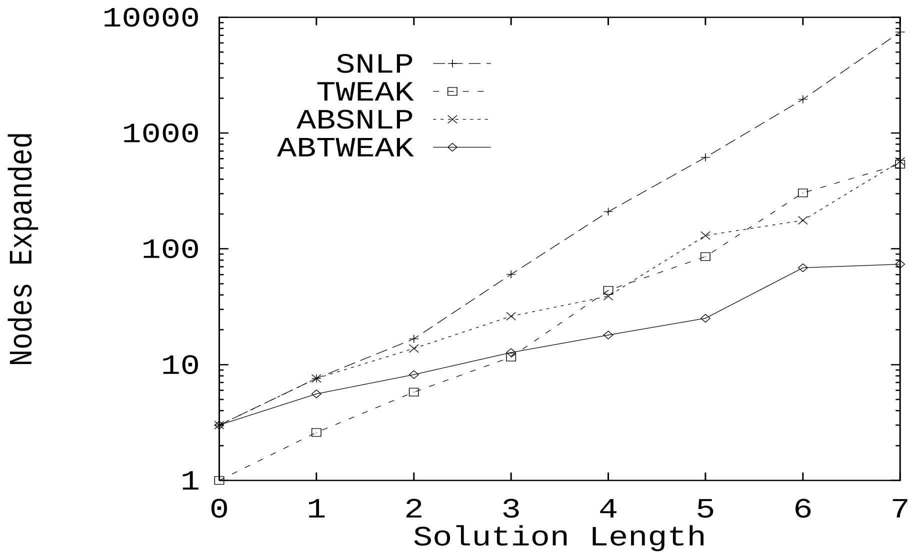

Figure 8: Nodes comparison of four planners.

goals, but it is also more advantageous to selectively protect the goal establishments so far.

8 Conclusions

This research is aimed at uncovering abstract planning heuristics that can improve search efficiency while maintaining completeness. In particular, we have used AbTweak as a test bed to study three heuristics for abstract search, including the use of monotonic goal protections in planning, the Left-Wedge heuristic, and methods of using primary effects in abstract planning. We have also empirically evaluated all methods. Our conclusions can be summarized below.

1. Using monotonic protection in abstract planning is often more advantageous than without, especially when difficult-to-achieve conditions are placed higher up in a hierarchy. This can be seen from the experiments with different hierarchies in the Towers of Hanoi domain. On the other hand, if a bad hierarchy is used, then using monotonic protection can reduce search efficiency.

2. We have demonstrated that a simplistic application of a control strategy for a single-level problem solver to each level of the abstraction hierarchy will not in general provide completeness for the multiple-level system. Completeness can be obtained by searching simultaneously in the space of alternative abstract plans (rightwards in the search tree), and in the space of refinements (downwards in the search tree). Preferring refinements over alternatives is the basis for the Left-Wedge strategy, which our experiments in

both the Towers of Hanoi domain and the robot domain show can optimize performance over a straightforward breadth-first strategy.

3. Certain domain-dependent heuristics, such as the use of primary effects in goal-achievement, can jeopardize the completeness of a hierarchy. However, a sufficient condition exists under which the completeness is preserved. The condition requires that all primary effects of an operator have criticalities at least as large as the other effects. In the robot domain, ABTWEAK with the hierarchy satisfying this constraint and using the LEFT-WEDGE control has shown a large amount of search reduction over search without abstraction.

Acknowledgement

We thank Craig Knoblock for many useful comments. This work was supported in part by research grants to Qiang Yang, from the Natural Sciences and Engineering Research Council of Canada, and by grants to Josh D. Tenenberg in part from ONR research grant no. N00014-90-J-1811, Air Force - Rome Air Development Center research contract no. F30602-91-C-0010, and Air Force research grant no. AFOSR-91-0108. Steven Woods would like to thank both the Commonwealth Scientific and Industrial Research Organization (CSIRO) Australia and the Defense Research Establishment Valcartier (DREV) Canada for providing valuable time to work on revisions of this paper.

References

[Bacchus and Yang 1991] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the 12th IJCAI*, pages 286–292, Sydney, Australia, August 1991.

[Bacchus and Yang 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the 10th AAAI*, pages 369–374, San Jose, CA, 1992.

[Chapman 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Christensen 1990] Jens Christensen. A hierarchical planner that creates its own hierarchies. In *Proceedings of the 8th AAAI*, pages 1004–1009, 1990.

[Fikes and Nilsson 1971] Richard Fikes and Nils Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Kambhampati 1992] Subbarao Kambhampati. Characterizing multi-contributor causal structures for planning. In *Proceedings of the First International Conference on AI Planning Systems*, 1992.

[Knoblock, Tenenberg, and Yang 1991] Craig Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the 9th AAAI*, Anaheim, CA, 1991.

[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.

[McAllester-Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th AAAI*, Anaheim, CA, 1991.

[Minton 1990] Steve Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[Newell and Simon 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice Hall, Enlewood Cliffs, NJ, 1972.

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

[Sacerdoti 1977] Earl Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, 1977.

[Siklossy and Dreussi 1973] L. Siklossy and J. Dreussi. An efficient robot planner which generates its own procedures. In *Proceedings of the 3rd IJCAI*, pages 423–430, 1973.

[Stefik 1981] Mark Stefik. Planning with constraints. *Artificial Intelligence*, 16(2):111–140, 1981.

[Sussman 1973] G.A. Sussman. *A Computational Model of Skill Acquisition.* M.I.T. AI Lab Memo no. AI-TR-297, 1973.

[Tate 1977] Austin Tate. Generating project networks. In *Proceedings of the 5th IJCAI*, pages 888–893, 1977.

[Unruh and Rosenbloom 1989] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of the 11th IJCAI*, pages 681–687, Detroit, MI, 1989.

[Waldinger 1977] R. Waldinger. Achieving several goals simultaneously. In *Machine Intelligence 8.* Elcock and Michie (eds.), Ellis Horwood, 1977.

[Warren 1974] David H.D. Warren. Warplan: A system for generating plans. Memo No. 76, Department of Computational Logic, University of Edinburgh, 1974.

[Wilkins 1984] David Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22, 1984.

[Woods 1991] Steven G. Woods. An implementation and evaluation of a hierarchical nonlinear planner. Master's thesis, Computer Science Department, University of Waterloo, 1991.

[Yang, Tenenberg, and Woods 1991] Qiang Yang, Josh Tenenberg, and Steve Woods. Abstraction in nonlinear planning. University of Waterloo Technical Report CS 91-65, 1991.

A Operators in the 3-disk Towers of Hanoi Domain

MoveLarge (x y)
Preconditions={IsPeg(x)
IsPeg(y)
¬ OnMedium(x)
¬ OnMedium(y)
¬ OnSmall(x)
¬ OnSmall(y)
OnLarge(x)}
Effects={¬ OnLarge(x)
(OnLarge y)}

MoveMedium (x y)
Preconditions={IsPeg(x)
IsPeg(y)
¬ OnSmall(x)
¬ OnSmall(y)
OnMedium(x)}
Effects={¬ OnMedium(x)
OnMedium(y)}

MoveSmall (x y)
Preconditions={IsPeg(x)
IsPeg(y)
OnSmall(x)}
Effects={¬ OnSmall(x)
OnSmall(y)}

B Operators in the Robot Task Planning Domain.

This appendix lists the operators used in the robot task planning domain. Primary effects of operators are marked by "*."

B.1 Operators for going between rooms

To push a box through a door between 2 rooms.

push-thru-dr (box door-nm from-room to-room door-loc-from door-loc-to robot)
Preconditions={Is-Door(door-nm from-room to-room door-loc-from door-loc-to)
Pushable(box)
Box-Inroom(box from-room)
Robot-Inroom(from-room)
Box-At(box door-loc-from)
Robot-At(door-loc-from)
Open(door-nm) }
Effects={¬ Robot-Inroom(from-room)
Robot-Inroom(to-room)
¬ Box-Inroom(box from-room)
Box-Inroom(box to-room)*
Robot-At(door-loc-to)
Box-At(box door-loc-to)*
¬ Robot-At(door-loc-from)
¬ Box-At(box door-loc-from) }

To go through door from room2 to room1.

go-thru-dr (door-nm from-room to-room door-loc-from door-loc-to)
Preconditions={Is-Door(door-nm from-room to-room door-loc-from door-loc-to)
Robot-Inroom(from-room)
Robot-At(door-loc-from)
Open(door-nm) }
Effects={Robot-At(door-loc-to)*
¬ Robot-At(door-loc-from)
¬ Robot-Inroom(from-room)
Robot-Inroom(to-room)*}

B.2 Operators for going within a room

Operator for going to a location in a room.

goto-room-loc (from to room)
Preconditions={Location-Inroom(to room)
Location-Inroom(from room)
Robot-Inroom(room)
Robot-At(from) }
Effects={¬ Robot-At(from)
Robot-At(to)*}

Operator for pushing box between locations within one room.

push-box (box room box-from-loc box-to-loc robot)
Preconditions={Pushable(box)
Location-Inroom(box-to-loc room)
Location-Inroom(box-from-loc room)
Box-Inroom(box room)
Robot-Inroom(room)
Box-At(box box-from-loc)
Robot-At(box-from-loc) }
Effects={¬ Robot-At(box-from-loc)
¬ Box-At(box box-from-loc)
Robot-At(box-to-loc)
Box-At(box box-to-loc)*}

B.3 Operators for opening and closing doors

To Open a door.

Open (door-nm from-room to-room door-loc-from door-loc-to)
Preconditions={Is-Door(door-nm from-room to-room door-loc-from door-loc-to)
¬ Open(door-nm)
Robot-At(door-loc-from) }
Effects={Open(door-nm)*}

To close a door.

close (door-nm from-room to-room door-loc-from door-loc-to)
Preconditions={Is-Door(door-nm from-room to-room door-loc-from door-loc-to)
Open(door-nm)
Robot-At(door-loc-from) }
Effects={¬ Open(door-nm)*}

C AbTweak Algorithm

C.1 Data Structures and Subroutines

1. OPEN — A priority queue of plans on the frontier of the search tree. The list is sorted in ascending order of the plans' costs, $Cost(\Pi)$.

2. MTC(Π) — A predicate on plans, which is true of Π exactly when Π is necessarily correct.

3. Successors(Π) — A function mapping each plan to a set of successor plans.

C.2 AbTweak

Algorithm AbTweak **(initial, goal):**

$OPEN \leftarrow$ Initial-Plan,
{where Initial-Plan is a plan with two operators, initial and goal).}
Loop
    If $OPEN$ is empty, **Then** exit with failure.
    Else, let $\Pi = First(OPEN)$, and $OPEN \leftarrow Remove(\Pi, OPEN)$.
    Endif
    If $crit(\Pi) = 0$ and $MTC(\Pi) =$ True, **Then** return Π, and exit with success.
    Else, **If** $MTC(\Pi) =$ True, **Then**
        {the plan Π is correct at an abstract level},
        $crit(\Pi) \leftarrow crit(\Pi) - 1$,
        $OPEN \leftarrow Insert(\{\Pi\}, OPEN)$,
        Else,
        {Successor Generation: Plan Π must contain at least one
        precondition that does not necessarily hold.}
        $OPEN \leftarrow Insert(Successors(\Pi), OPEN - First(OPEN))$.
        Endif
    Endif
Endloop

C.3 Successor Generation

Subroutine Successor (Π)
{Comment: The global variable MP is True whenever the Monotonic Protection is used in AbTweak.}

$succ := \emptyset$
$successors := \emptyset$
Find a precondition *precond* of an operator *User* in plan Π,
such that *precond* is not necessarily true
If MP = True and *precond* is established for *User* at a level higher than $crit(\Pi)$ **Then**
  For each of the abstract establishment relations of the form $Est_i = \text{Establishes}(\alpha_i, User, precond)$,
    that have been clobbered at the current level,
      $succ := \{(\Pi, Est_i)\} \bigcup succ.$
  Endfor
Else
    Let *Old* be the set of operators in Π whose effects possibly
    establish *precond* for *User*, and let *New* be the set of new
    operators taken from the operator schemas of the domain, that have
    effects which possibly codesignate with *precond*.

    For each operator α in $Old \bigcup New$ **Do**
      (1) Add temporal and codesignation constraints to a copy Π' of Π
      so that for some effect e_α of α, the relation
      $Est = \text{Establishes}(\alpha, User, precond)$ holds
      (2) $succ := succ \bigcup \{(\Pi', Est)\}$
Endif
{ **Declobber** }
For each pair (Π',$Est = \text{Establishes}(\alpha, User, precond)$) in *succ*, **Do**
    If *Est* is clobbered **Then**
      For each clobberer C of *Est*, with a clobbering effect e_C, **Do**
        (1) impose the constraint $C \prec \alpha$, onto a copy Π_1 of Π',
        (2) impose the constraint $User \prec C$, onto a copy Π_2 of Π',
        (3) impose the constraint $e_C \not\approx \neg precond$, onto a copy Π_2 of Π'.
        (4) **For** each $\Pi_i, i = 1, 2, 3$, if the constraints in Π_i are consistent, then
          $successors := successors \bigcup \{\Pi_i\}$.
        Endfor
        {Each copy is a new successor in the search space.}
      Endfor
    Else $successors := successors \bigcup \{\Pi'\}$
    Endif
Endfor

If MP = True **Then** {Monotonic Protection }
  For every plan Π' in *successors*, **Do**
    If there is some precondition p of an operator β,
    with an establishment set S in Π', such that
      For every establishment relation Establishes$(\alpha, \beta, p) \in S$
        there is an operator γ such that
    {Note: This condition defines monotonic violation.}
      (1) $(\alpha \prec \gamma \prec \beta)$,
      (2) For some effect e_γ of γ,
          either $(e_\gamma \approx p)$ or
          $(e_\gamma \approx \neg p)$.
      Endfor
    Then *successors* := *successors* $- \{\Pi'\}$
    Endif
  Endfor
Endif
Return *successors*

C.4 Tweak Implementation

Tweak can be implemented by making the following modifications to the AbTweak routines:

1. $crit(\Pi) = 0$, for all Π,
2. In the successor generation part, remove the two monotonic protection components.

D Proof of the Completeness of MP

D.1 Ascending Preserves Correctness

An abstract version of a correct plan can be obtained by removing preconditions of operators while holding the plan fixed. Intuitively, this abstraction process does not affect the necessary truth of any condition, since it does not add any new operators into the plan, or delete any constraints from the plan. Thus, after removing a precondition of an operator in Π, the resulting plan Π' is still correct. This is stated in the following lemma.

Lemma D.1 *Let Π be a plan, and $\alpha \neq \mathcal{G}$ be an operator in Π. Let α' be α with precondition p removed from P_α. Let Π' be Π with α replaced by α'. If Π is correct, then Π' is also correct.*

¿From Lemma D.1, the following theorem can be easily proven by induction on the abstraction levels.

Theorem D.2 *If* Π *is a correct plan for goals* G *at level 0, then* ${}_i\Pi$ *is a correct plan for solving* G *on each higher level* $i = 1, 2, \ldots, k-1$.

D.2 Justification

Abstracting a correct plan Π to a higher level involves removing the preconditions of operators that have low criticality values. The resulting abstracted plan can thus contain operators and constraints that were in the plan at lower levels solely to establish one of the removed preconditions. All such operators and constraints are superfluous to the abstract plan, and can thus be safely removed without affecting the correctness of the abstract plan. A plan thus obtained is called a *justified* plan, and denoted by $\mathtt{Jus}(\Pi)$. For example, in the robot task planning domain, suppose we have a plan to push a box from one room to another. If the door open precondition for the open door operator has a low criticality value, then removing this precondition at the abstract level would cause the open door operator to have no purpose in the abstract plan, and thus it can be safely removed. Notice that for a correct plan Π, there can be more than one justified plan $\mathtt{Jus}(\Pi)$.

Justification provides a specification of those plan steps which are required in order to solve the goal, either directly or indirectly. From its definition, we trivially have the following lemma.

Lemma D.3 *If* Π *is a correct plan that solves goal* G*, then* $\mathtt{Jus}(\Pi)$ *also is a correct plan that solves* G.

We can now establish the *Upward Solution Property*:

Theorem D.4 *If* Π *is a correct plan that solves* G *at the base level, then the justified version of* ${}_i\Pi$*,* $\mathtt{Jus}({}_i\Pi)$*, is also a correct plan that solves* G *on the* i^{th} *level,* $0 \leq i \leq k-1$.

Proof: ¿From Theorem D.2, if Π is a correct plan that solves G at the base level, then on each level i, $1 \leq i \leq k-1$, ${}_i\Pi$ is a correct plan for solving G. From Lemma D.3, $\mathtt{Jus}({}_i\Pi)$ is also a correct plan that solves G on level i. □

D.3 Monotonic Refinement

In this section, we want to show the following property: for any hierarchy, if a solution exists to a planning problem, then a search path exists in ABTWEAK's search space on which every plan preserves the abstract plan structure on that path. This property, which we will call the monotonic property, follows directly from the upward solution property, and is *universal* in the sense that it does not depend on any particular abstraction hierarchy. We gave a formal definition for ABSTRIPS-type abstraction systems in [Knoblock, Tenenberg, and Yang 1991]. Here we define it for nonlinear, least-commitment planning abstraction systems.

Definition D.5 *Let* Π' *be an abstract plan that solves* ρ *at level* i*,* $i > 0$. Π' monotonically refines *to the level* $i-1$ *plan* Π *if and only if*

1. Π *solves* ρ *at level* $i-1$*, and*

2. $\texttt{Jus}({}_i\Pi) = \Pi'$.

In other words, a plan Π is a monotonic refinement of Π', if Π preserves the set of all operators, the partial order, and the codesignation and non-codesignation relations of the plan Π'.

Definition D.6 *A k-level* ABTWEAK *system is monotonic, if and only if, for every problem* ρ *solvable at the concrete* (0^{th}) *level, there exists a sequence of plans* $\Pi_{k-1}, \ldots, \Pi_0$ *such that* Π_{k-1} *solves* ρ *at level* $k-1$*, and for* $0 < i < k$*,* Π_i *monotonically refines to* Π_{i-1}*.*

Now we show that *every* criticality assignment gives rise to a monotonic hierarchy.

Theorem D.7 *Every* ABTWEAK *system of k levels, for any k, is monotonic.*

Proof: This will be proven by contradiction for a 2 level system, where extending the result to k levels follows from a trivial induction. Let ρ be a problem solved at the concrete level. Let Π be a correct base level plan that solves ρ. By the Upward Solution Property, there exists an abstract plan $\Pi' = \texttt{Jus}({}_1\Pi)$ solving ρ at the abstract level. By Definition D.5, Π is a monotonic refinement of Π'. □

There is a direct relationship between monotonic protection defined in Section 3 and monotonic refinement.

Lemma D.8 *Let* Π' *be a correct plan at level i and let* Π *be a monotonic refinement of* Π'*. Then* Π *is monotonically protected with respect to* Π'*.*

Proof: We prove this lemma by contradiction. Consider the plan ${}_i\Pi$. Π' is obtained by removing superfluous operators and constraints form ${}_i\Pi$. Let β be an operator in Π' and p be a precondition of β. If all establishers of p are removed in the process of justifying ${}_i\Pi$, then no establisher for p will exist in Π'. Thus, by MTC, Π' is incorrect. This contradicts our assumption that Π' is a correct plan at level i. □

We can now prove Theorem 3.4. Suppose that a solution plan exists at the lowest level of abstraction. Then by Theorem D.7, a solution plan Π_i exists at every abstraction level i, $i = 0, 1, \ldots, n-1$, such that each Π_i can be monotonically refined to a plan Π_{i-1} at a level below. ¿From lemma D.8 every plan Π_i is also monotonically protected with respect to Π_{i-1}. Thus, the search path from Π_{n-1} to Π_0 is one in which every plan is monotonically protected with respect to all previous ones. This search path also exists in the search space of ABTWEAK with MP.

E Proof of Primary Effects Theorem

Proof: We provide a sketch of the proof, as the majority of the proof is similar to that of the completeness of MP.

First, consider a two level system. We want to show that if a solution exists in which every operator achieves a precondition or goal using a primary effect, then the abstract version of the plan also has the property if Condition 5.1 holds. Let Π be a plan in which every operator achieves a precondition or goal using a primary effect. In its abstract version ${}_i\Pi$ we can perform justification, and then remove all operators that only achieve a precondition using one of its side effects. Let the resulting plan be Π'. We want to show that Π' is also correct.

Suppose Π' is incorrect. This must be due to the removal of an operator which only achieves a precondition q as a side effect. Let this operator be α. Since in Π the operator α achieves other preconditions using both primary and side effects, it must be the case that one of the preconditions p that is removed during the abstraction process is one of α's primary effects. Since p is removed while q is not, it must be true that $crit(p) < crit(q)$. However, this contradicts with Condition 5.1.

Therefore, Π' must be a correct plan. The proof that Π' monotonically refines to Π is similar to that shown in lemma D.3, page 44. $\Box$

F Experimental Results

Hierarchies	Expanded	Generated:	MP Pruned:	CPU Seconds:
ILMS	471	794	0	252.2
IMLS	166	233	65	60.4
IMSL	652	1037	270	421.4
ILSM	765	1205	172	540.2
ISLM	1083	1433	800	929.8
ISML	over 5000	6157	5282	9579.8
LIMS	609	1004	0	677.1
MILS	295	428	117	235.9
MISL	698	1107	274	868.1
LISM	907	1419	177	1137.4
SILM	522	715	329	480.2
SIML	844	1148	609	993.6
LMIS	1717	3661	0	1853.9
MLIS	313	597	77	304.8
MSIL	1339	2537	378	1862.7
LSIM	3339	6563	321	4207.1
SLIM	389	695	175	395.7
SMIL	989	1571	617	1350.2
LMSI	1894	4613	0	2690.8
MLSI	382	851	77	420.4
MSLI	3263	8065	1261	6460.7
LSMI	over 5000	12496	665	10339.2
SLMI	640	1388	200	892.3
SMLI	2519	6795	1286	4987.9

Figure 9: Monotonic Protection, Breadth-First

Hierarchies	Expanded	Generated:	MP Pruned:	CPU Seconds:
ILMS	471	794	0	218.4
IMLS	550	934	0	213.6
IMSL	918	1790	0	548.1
ILSM	1112	1973	0	730.7
ISLM	1771	3142	0	1224.5
ISML	3142	6171	0	3333.5
LIMS	609	1004	0	531.2
MILS	700	1215	0	500.2
MISL	964	1864	0	948.6
LISM	1257	2197	0	1321.1
SILM	1578	2899	0	1442.7
SIML	3249	6388	0	3589.5
LMIS	1717	3661	0	1668.8
MLIS	1181	2736	0	1020.8
MSIL	1605	3398	0	1728.3
LSIM	3700	7509	0	4355.0
SLIM	2249	5014	0	2363.6
SMIL	1449	3124	0	1514.4
LMSI	1894	4613	0	2111.4
MLSI	1217	3060	0	1255.3
MSLI	3874	10782	0	5783.0
LSMI	over 5000	13029	0	6849.5
SLMI	2940	7619	0	3821.4
SMLI	2359	6445	0	2892.1

Figure 10: Breadth-First only

Hierarchies	Expanded	Generated:	MP Pruned:	CPU Seconds:
ILMS	57	99	0	38.2
IMLS	86	129	16	38.1
IMSL	3904	4776	3891	4971.6
ILSM	608	808	408	1102.0
ISLM	over 5000	6268	3979	15491.3
ISML	over 5000	5841	5391	12688.5
LIMS	56	98	0	79.1
MILS	94	138	21	81.3
MISL	over 5000	6115	5015	12604.9
LISM	607	807	408	1599.1
SILM	4094	5018	2954	17645.9
SIML	4992	5780	5119	14341.2
LMIS	56	101	0	87.9
MLIS	73	122	9	126.4
MSIL	over 5000	6011	5116	17385.6
LSIM	1587	2116	931	7030.2
SLIM	over 5000	6592	3670	24998.1
SMIL	over 5000	6161	5151	16803.1
LMSI	250	636	0	540.5
MLSI	170	363	9	255.1
MSLI	over 5000	13780	3935	14721.4
LSMI	3142	7587	831	9513.8
SLMI	over 5000	12050	1526	17247.3
SMLI	over 5000	15215	4991	10773.0

Figure 11: Monotonic Protection, Left-Wedge

Hierarchies	Expanded	Generated:	MP Pruned:	CPU Seconds:
ILMS	57	99	0	30.5
IMLS	1009	1811	0	1470.5
IMSL	over 5000	9675	0	6670.5
ILSM	828	1471	0	1260.3
ISLM	168	284	0	214.9
ISML	963	1771	0	1459.0
LIMS	56	98	0	55.0
MILS	1008	1810	0	2278.8
MISL	over 5000	9675	0	9322.0
LISM	827	1470	0	1327.0
SILM	167	283	0	200.3
SIML	962	1770	0	1628.7
LMIS	56	101	0	84.7
MLIS	989	1785	0	2474.5
MSIL	over 5000	9691	0	10777.2
LSIM	1915	3500	0	4815.7
SLIM	148	258	0	195.7
SMIL	943	1745	0	1764.5
LMSI	250	636	0	402.7
MLSI	379	828	0	458.1
MSLI	over 5000	14416	0	9294.4
LSMI	4737	12387	0	9627.0
SLMI	over 5000	13633	0	10813.7
SMLI	over 5000	14571	0	10189.5

Figure 12: Left-Wedge only

List of Symbols

Symbol	**Meaning**
α, β, γ	Operators
Π	Plan
$\approx$	codesignation constraint
$\not\approx$	non-codesignation constraint
$crit$	criticality value
P_α	preconditions of α
${}_iP_\alpha$	preconditions with criticalities higher than i
$\neg$	logical negation
MP	monotonic goal protection

List of Tables

List of Figures