Abstraction in Nonlinear Planning

Qiang Yang *
University of Waterloo
Canada

Josh D. Tenenberg †
University of Rochester
USA

Steven Woods‡
CSIRO Division of Information Technology
Australia

Abstract

We extend the hierarchical, precondition-elimination abstraction of ABSTRIPS to nonlinear, least-commitment planners such as TWEAK. Specifically, we show that the combined planning system, ABTWEAK, satisfies the *monotonic property*, whereby the existence of a lowest level solution Π implies the existence of a highest level solution that is structurally similar to Π. This property enables one to prune a considerable amount of the search space without loss of completeness. In addition, we develop a criteria for good abstraction hierarchies, and develop a novel, complete search strategy called *Left-Wedge* that is optimized for good abstraction hierarchies. We demonstrate the utility of both the monotonic property and the Left-Wedge strategy through a series of empirical tests.

Abbreviated Title: Same as the title.

*Computer Science Department. Waterloo, Ont. Canada, N2T 3G1. Email: qyang@watdragon.waterloo.edu.

†Computer Science Department. Rochester, New York, U.S.A., 14627 E-mail: josh@cs.rochester.edu

‡CSIRO Division of Information Technology, Centre for Spatial Information Systems, Canberra, ACT, Australia, 2601. Email: sgwoods@csis.dit.csiro.au

1 Introduction

Symbolic planning representations have been useful in AI because they enable planning agents that use them to make strong predictions about future states given the execution of a hypothetical sequence of actions (a *plan*). In addition, the associated formal methods allow researchers to make and verify strong formal claims, for instance, regarding soundness, completeness, and complexity.

However, one of the central problems with this formalist approach is that finding plans that will bring about *desired* states is very resource intensive, typically involving a heuristically guided traversal of large portions of the state space. Researchers have pursued several approaches to reduce this resource cost, two of the most prominent being nonlinear least-commitment planning, such as Chapman's TWEAK [1], and hierarchical abstraction through precondition elimination, such as Sacerdoti's ABSTRIPS [10]. In this paper, we describe our research in combining these two approaches. The first half of our paper demonstrates that the formal characteristics of each are preserved when combined, while the second part, describing a set of experiments and a discussion of search strategies, demonstrates that significant search efficiencies can often be obtained.

In nonlinear, least-commitment planning, at any time in the plan construction process, the temporal ordering and variable bindings of the operators need only be *partially* specified through the incremental posting of pairwise constraints. For instance, suppose that one has the problem of drilling a large hole in a block of wood and nailing a plate to this block. If the plate and the hole are spatially disjoint, as in figure 1, then there is no reason to initially order one operation before the other. A linear planning system, such as STRIPS or ABSTRIPS, because of its weak plan representation, would be required to impose a total ordering on these operations, such as to first attach the plate, and then drill the hole. However, constraints added by subsequent planning might impose an order contrary to that which was initially chosen. One such constraint would arise if one of the nails holding the plate protrudes into the space carved out by the drill press, as in figure 2. In this case, the planner would need to backtrack and reorder the plan steps, or add additional steps into the original plan. A nonlinear planner, however, would not have imposed an ordering on the operators if none was yet called for, and thus would not be required to backtrack when the potential clobbering

was uncovered. All that would be required would be to add an ordering constraint specifying that the drilling must precede the attaching.

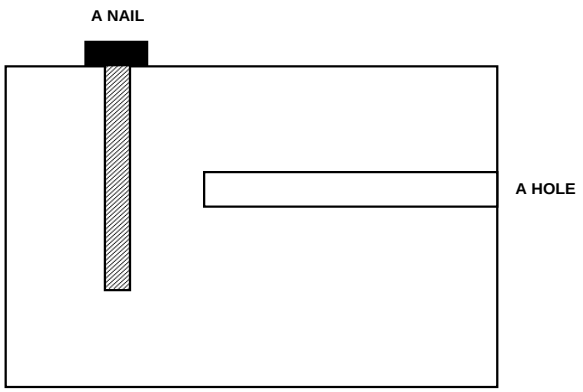

Figure 1: Spatially disjoint nail and hole.

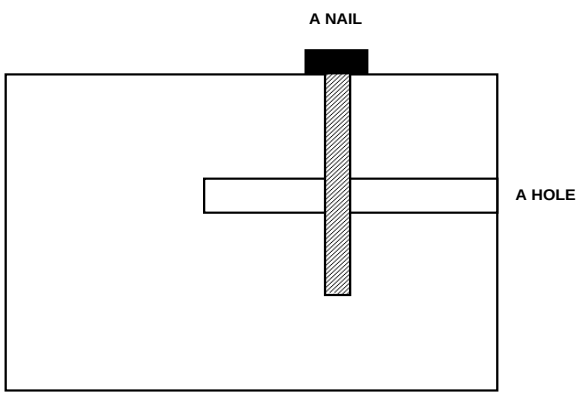

Figure 2: Interacting nail and hole.

Likewise, if any one of a number of drill presses will suffice for the hole drilling, there is no reason to choose one over the other, at least until subsequent constraints require it. Thus, least-commitment planners will not require an arbitrary choice of variable bindings. Thus, by deferring planning choices and incrementally collecting constraints, one performs less backtracking than would occur from the premature commitment to planning decisions.

A partial plan represented by pairwise ordering and codesignation constraints thus stands for a *space* of plans, namely, all totally ordered, fully

ground plans satisfying these constraints. An assumption underlying this representation is that only a single one of these plans will be chosen for execution. Thus, a formal means is required for determining that *each* plan completion, if executed, will solve the goal. Not only is this verification important for goal satisfaction, but it is crucial in plan construction, since one must verify for each operator that its preconditions will all be satisfied in the state in which it is applied, regardless of the ordering and variable instantiations of the predecessor operators. This formal requirement is met by Chapman in [1], and is termed the *Modal Truth Criterion* (MTC). The virtue of this criterion is that it can be checked in polynomial time as a function of the number of operators in a plan, even though there are potentially an exponential number of total orderings and an infinite number of codesignations that satisfy the given constraints. Demonstrating that the conditions of the Modal Truth Criterion are satisfied at higher levels of abstraction is one of the main technical results of this paper.

Hierarchical planning through precondition elimination is primarily concerned with the order in which goals and subgoals are solved. All atomic statements are given a *criticality* ranking, and all goals and subgoals having a particular criticality value are solved before any goal or subgoal having a lower criticality. The intuition has traditionally been that high criticality propositions are those most difficult to achieve, and low criticality propositions the least difficult. A criticality assignment thus generates a hierarchically ordered sequence of planning levels on an otherwise non-hierarchical planning system, each level ignoring propositions having lower criticality values. A plan is first obtained at the highest criticality level. Once a solution is obtained at a particular criticality level i, it is refined at level $i-1$ by inserting operators between the level i plan steps in order to satisfy their criticality $i-1$ preconditions. For example, someone traveling from Toronto to IJCAI91 in Sydney would likely first plan their air-travel before planning their intra-city travel. In terms of our formulation, this amounts to ignoring the preconditions that one must be at the airport in order to fly between cities. These concrete level preconditions are considered only after the entire abstract air-travel plan has been formulated. Problem-solving in this manner often promises an exponential amount of savings in computation (see [5, 6]).

A theoretical principle that underlies the abstraction/refinement search strategy states that for all problems having a lowest criticality level solution, there exists a highest level solution that refines through insertion in the

above-described manner down to the lowest level. This principle is described in [4], and is termed *Monotonic Refinement*. It actually states something even stronger: that the criticality i subgoals are *protected* from lower level insertions, and are thus never violated at lower levels. In essence, this principle states that one can structure search so that work performed at the higher levels is never undone at lower levels. What is surprising about this property is that it holds for *any* assignment of criticalities to propositions.

We define a planning system, ABTWEAK, which applies ABSTRIPS style abstraction to TWEAK, yielding a hierarchy of nonlinear planning levels. Our main theoretical results establish that 1) the correctness of plans as specified by the Modal Truth Criterion is preserved when going up abstraction levels, and 2) the Monotonic Property holds of ABTWEAK systems, regardless of criticality assignment. This latter property thus serves as a basis for an iterative refinement search strategy, analogous to the case for linear planners. Due to the fact that this property holds regardless of criticality, it is not a sufficient constraint to guarantee performance improvements under any criticality assignment as compared to other search strategies. However, it does suggest a criterion for *good* criticality assignments – those assignments which result in few protection violations. Good criticality assignments will tend to be those in which satisfaction of a subgoal will not rely upon the satisfaction of lower-valued subgoals.

Developing criteria for good abstraction hierarchies is not sufficient for good performance: one must additionally use search strategies that exploit the control knowledge embedded in the abstraction hierarchy. Is breadth-first search, depth-first, or some other search strategy most appropriate? How will this search strategy impact the completeness of the planner? Unfortunately, this is the aspect of precondition-elimination abstraction that is the least understood. Thus, in exploring search, we were not able to generalize strategies useful in previous hierarchical planners.

With good abstraction hierarchies, early plan choices are less likely to be abandoned. This suggests that search proceed primarily in a depth-first manner. However, since the first abstract plan found might not be monotonically refinable, completeness requires that alternative abstract plans must at the same time be pursued. There is, then, a tradeoff between search through the space of alternative abstract plans, and search through the space of plan refinements. To address this tradeoff, we have developed a new search strategy, the *Left-Wedge*, which has elements of both depth- and breadth-first search.

Left-Wedge is a complete strategy, but it gives priority to those plans that are most refined. That is, it prefers to push deeper into the space of refinements, but expends some smaller percentage of its resources in considering abstract alternatives.

In assessing the practical utility of the monotonic property and the left-wedge search control strategy, we have conducted a series of empirical tests. Our results show that in general, an abstract planner using the monotonic property outperforms one without. We also show that with intuitively good abstraction hierarchies, the left-wedge search strategy improves search efficiency dramatically over a straightforward application of the breadth-first search.

We first present brief descriptions of TWEAK and ABSTRIPS, and then define ABTWEAK. We demonstrate that ABTWEAK has the monotonic property, and show how the application of this property affects search.

2 Nonlinear Planning: TWEAK

Chapman [1] provides a formalization of a least commitment, nonlinear planner, TWEAK. TWEAK extends STRIPS 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. One should note that TWEAK systems are somewhat constrained in their expressive power. In particular, actions cannot have *indirect* or *context-dependent* effects, and the domain of each variable is taken to be infinite, and identical to the domains of all other variables. Thus, there is no straightforward way of representing limited resources.

Chapman did not give a formal definition of a TWEAK plan [1]. 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* $(\text{Operators}_\Pi, \prec_\Pi, \text{Co\&Nonco}_\Pi)$*, where*

- Operators_Π *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_Π *such that the transitive closure of* $\prec_\Pi$ *is a partial order (irreflexive, asymmetric, transitive),*

- $\text{Co\&Nonco}_\Pi$*, 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.* $\text{Co\&Nonco}_\Pi$ *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*.*

  We extend $\approx$ *and* $\not\approx$ *to propositions:*

  1. $P(x_1, \ldots, x_m) \approx_\Pi Q(y_1, \ldots, y_n)$ *iff* $P = Q$*,* $m = n$*, and* $x_i \approx_\Pi y_i$ $1 \leq i \leq n$
  2. *For atoms* p *and* q*,* $\neg p \approx_\Pi \neg q$ *iff* $p \approx_\Pi q$*.*

  The plan subscripts to $\approx$*,* $\not\approx$*, and* $\prec$ *will be dropped if the plan to which these relations refer is clear from context.*

With the above definition, we can now restate several terminologies used in [1] formally. A *complete plan* Π is a plan where $\prec_\Pi$ is a linear ordering on Operators_Π, and $\text{Co\&Nonco}_\Pi$ is such that every variable in every operator of Operators_Π codesignates with some constant. A plan *completion* of Π refers to any complete plan Π' that satisfies the partial 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 example, consider the 3-disk Tower of Hanoi domain. The locations of the disks are represented by three predicates, `OnLarge` `OnMedium`and `OnSmall` each taking a single argument denoting the peg. The `IsPeg( )`predicate is used as a precondition in the operator definitions to ensure that every variable is instantiated to an existing peg. Thus, the language L of the TWEAK system consists of the symbols listed in Table 1.

Symbols

Predicate	`IsPeg()`,`OnLarge()`,`OnMedium()`,`OnSmall()`
Constant	`Peg1`,`Peg2`,`Peg3`, $c_1, c_2, \ldots$
Variables	$x_1, x_2, \ldots$

Table 1: Tower of Hanoi Domain Language.

The `MoveLarge` operator is given in Table 2, with the full operator set listed in Appendix A. This operator is used for moving the large disk from peg x to peg y. An input problem in the Towers of Hanoi domain is to move all three disks from `Peg1` to `Peg3`. For this problem, the goal G is represented by the set

$$\{\texttt{OnLarge(Peg3)}, \texttt{OnMedium(Peg3)}, \texttt{OnSmall(Peg3)}\}$$

A plan for solving this problem is:

$$\langle \texttt{MoveSmall(Peg1,Peg3)}, \texttt{MoveMedium(Peg1,Peg2)}, \texttt{MoveSmall(Peg3,Peg2)}, \texttt{MoveLarge(Peg1,Peg2)}, \texttt{MoveSmall(Peg2,Peg1)}, \texttt{MoveMedium(Peg2,Peg3)}, \texttt{MoveSmall(Peg1,Peg3)}\rangle. \quad (1)$$

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

Table 2: Definition of the `MoveLarge` operator.

The temporal order is linear, indicated by the left-to-right and top-to-bottom order of the literals, and the replacement of the operator variables by constants indicates the codesignations. This same convention for writing the ordering and codesignation constraints will be used throughout.

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 $\alpha \in \text{Operators}_{\Pi}$, $(\mathcal{I} \prec_{\Pi} \alpha)$ and $(\alpha \prec_{\Pi} \mathcal{G})$.

In general a plan can have many different instantiations and many different total orderings consistent with the partial order. We will often want to talk about constraints that hold in all completions, or that hold in at least one completion, for which Chapman presents the possibility and necessity operators. In a partial plan, two terms p and q *necessarily* codesignate, denoted $\Box(p = q)$, if they codesignate under every completion, and *possibly* codesignate, $\Diamond(p = q)$, if they codesignate under some completion. Operator α necessarily precedes β, denoted $\Box(\alpha < \beta)$, if α precedes β under every completion, and α possibly precedes β, $\Diamond(\alpha < \beta)$, if α precedes β under some completion.

Given our previous definition of plans, we have the following equivalences:

$$\begin{aligned}
&\Box(\alpha < \beta) \iff (\alpha, \beta) \in \prec_\Pi \\
&\Diamond(\alpha < \beta) \iff \neg\Box\neg(\alpha < \beta) \iff (\alpha, \beta) \notin \prec_\Pi, \\
&\Box(p = q) \iff (p, q) \in \approx_\Pi, \\
&\Diamond(p = q) \iff \neg\Box\neg(p = q) \iff (p, q) \notin \not\approx_\Pi, \\
&\Box(p \neq q) \iff (p, q) \in \not\approx_\Pi, \\
&\Diamond(p \neq q) \iff \neg\Box\neg(p \neq q) \iff (p, q) \notin \approx_\Pi
\end{aligned}$$

The following definitions introduce simplifying notation.

$$\begin{aligned}
\Diamond(\alpha < \gamma < \beta) &\iff \Diamond(\alpha < c) \text{ and } \Diamond(\gamma < \beta), \\
\Box(\alpha < \gamma < \beta) &\iff \Box(\alpha < c) \text{ and } \Box(\gamma < \beta).
\end{aligned}$$

3 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:

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

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:

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

For example, an ABTWEAK system can be constructed for the Towers of Hanoi domain using a criticality assignment to literals as shown in Table 3. With this hierarchy, the operator `MoveLarge`(x_1,x_2), when represented on level 2 of the hierarchy, appears as:

$_2$`MoveLarge` (x_1 x_2)
Preconditions={`IsPeg`(x_1)
`IsPeg`(x_2)
`OnLarge`(x_1)}
Effects={¬ `OnLarge`(x_1)
`OnLarge`(x_2)}

Criticality	Predicate
2	`IsPeg()`, ¬ `OnLarge()`, `OnLarge()`
1	¬ `OnMedium()`, `OnMedium()`
0	¬ `OnSmall()`, `OnSmall()`

Table 3: A criticality assignment for Towers of Hanoi domain.

At the abstract level, the following plan Π solves the original TOH problem:

$$\text{Operators}_\Pi = \{\texttt{MoveLarge}(\texttt{x}_1,\texttt{x}_2),\texttt{MoveMedium}(\texttt{x}_3,\texttt{x}_4),\texttt{MoveSmall}(\texttt{x}_5,\texttt{x}_6)\}$$
$$\prec_\Pi = \{\}$$
$$\approx = \{(x_1,\texttt{Peg1}),(x_2,\texttt{Peg3}),(x_4,\texttt{Peg3}),(x_6,\texttt{Peg3})\}$$
$$\not\approx = \phi$$

There are 3 operators in this plan, one for getting each disk on `Peg3`. In addition, at the abstract level, there are no temporal constraints between the operators. Thus, any ordering will achieve the goal. Finally, the only instantiations required for solving the goal are that the first operator move the large disk from `Peg1` to `Peg3`, and that the other operators move their respective disks from some peg to `Peg3`.

4 Upward Solution Property

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.

Note the distinction between plan completion and plan refinement. Completion refers to specifying a totally ordered, fully ground plan that satisfies all of the ordering and codesignation constraints of the partial plan. Refinement refers to inserting plan steps at level i into a plan constructed at levels greater than i.

Implicit in the top-down search strategy, termed *length first search* by Sacerdoti [10], is the assumption that short plans to solve a given problem are guaranteed to exist at the abstract level which can be successively refined, and that search strategies exist to find such abstract plans. Our intent is to formally prove this property, and to show how it places some useful constraints on search. The intuition behind the proof is to show that if there exists a concrete level solution to a problem, then this solution will also solve the problem at each higher level of abstraction, since these higher levels do not place any new constraints on the problem. Further, since there are fewer preconditions at the higher levels, one can eliminate from this plan those operators whose purpose at lower levels is solely to satisfy, directly or indirectly, one of the eliminated preconditions.

In order to prove this, we first formalize what is meant by one operator directly or indirectly satisfying the preconditions of another, and then define correctness for partial plans. This new definition of correctness is used to verify that the shorter, abstract plans are correct and satisfy the goal. In the following definitions, all operators are taken to be relative to a plan Π.

4.1 Establishment

Definition 4.1 *Let* Π *be a plan. Operator* α *establishes proposition* p *before operator* β *(*$\text{Establishes}(\alpha, \beta, p, u)$*) if and only if*

1. $u \in E_\alpha$,

2. $\Box(\alpha < \beta)$,

3. $\Box(u = p)$, *and*

4. $\forall \alpha' \in \text{Operators}_\Pi, \forall u' \in E_{\alpha'}$, *if* $\Box(\alpha < \alpha' < \beta)$, *then* $\neg\Box(u' = p)$.

The final condition ensures that α is the last such operator that establishes p for β. This definition is aimed at formalizing and unifying many similar notions of *causal relationship* used in the planning literature, including *establishers* used in SIPE[11], *contributors* in NONLIN, validation structure in Priar[3], *causal links* in [7], *protection intervals* by Charniak and McDermott [2], and *triangle tables* used by Fikes and Nilssion[9].

Consider again the Towers of Hanoi example, where we assume that initially all three disks are on `Peg1`. The following plan solves the goal (`OnLarge(Peg3)`) at the lowest level.

$$\langle \texttt{MoveSmall(Peg1,Peg3)}, \texttt{MoveMedium(Peg1,Peg2)}, \texttt{MoveSmall(Peg3,Peg2)}, \texttt{MoveLarge(Peg1,Peg3)} \rangle \tag{2}$$

In this plan, the operator `MoveSmall(Peg1,Peg3)` establishes the precondition $\neg$`OnSmall(Peg1)` of operator `MoveMedium(Peg1,Peg2)`. Similarly, `MoveSmall(Peg3,Peg2)` is an establisher for `MoveLarge(Peg1,Peg3)`.

Informally, one operator α *clobbers* a proposition just prior to another operator β if α possibly precedes β and possibly denies this proposition. A *white knight* is another operator which necessarily re-establishes this clobbered proposition.

Definition 4.2 *γ is a clobberer of p before β, (*$\text{CB}(\gamma, \beta, p, q)$*) if and only if*

(1) $q \in E_\gamma$,
(2) $\Diamond(\neg q = p)$,
(3) $\Diamond(\gamma < \beta)$

Definition 4.3 *δ is a white knight of p before β, (*$\text{WK}(\delta, \beta, \gamma, p, q, r)$*), if and only if*

(1) $\text{CB}(\gamma, \beta, p, q)$,
(2) $r \in E_\delta$,
(3) $\Box(\gamma < \delta < \beta)$, *and*
(4) for every completion α of Π, if $(\neg q \approx_\alpha p)$ *then* $(r \approx_\alpha p)$.

4.2 Correctness: The Modal Truth Criterion

A *complete* plan is correct if every precondition of every operator is satisfied in the state in which the operator is applied. By our earlier definitions, this condition holds whenever every precondition has some establisher and there is no subsequent clobberer. A TWEAK plan is correct if every completion is correct. However, given the exponential number of total orderings and infinite number of constant instantiations, it is impossible to check each completion separately for correctness. We would instead like a criterion that decides the truth of a proposition at a given point in a plan, and which can be translated into a polynomial time algorithm.

Fortunately, Chapman [1] provides a concise statement of the criterion, which he calls the *Modal Truth Criterion* (MTC), and which, for each precondition, can be computed in time $\mathbf{O}(n^3)$, n being the number of operators in the plan. A problem with his definition, however, is that it is stated in terms of *situations*, which are not well-defined in a partial plan. For that reason, we provide a modified version of the MTC, defined solely in terms of operators in a plan. Intuitively, it states that a proposition p is true at a point in a plan if and only if p has an establisher, and for every clobberer of p, there exists a white knight.

Criterion 4.1 *Proposition p is necessarily true just before operator b in plan Π if and only if*

1. *there exists operator $\alpha \in \text{Operators}_\Pi$ and proposition u such that* $\text{Establishes}(\alpha, \beta, p, u)$, *and*

2. *for every operator $\gamma \in \text{Operators}_\Pi$ and $q \in E_\gamma$ such that* $\text{CB}(\gamma, \beta, p, q)$, *there exists $\delta \in \text{Operators}_\Pi$ and $r \in E_\delta$ such that* $\text{WK}(\delta, \beta, \gamma, p, q, r)$.

4.3 Ascending Preserves Correctness

As defined above, a plan Π is correct if and only if every precondition of every operator is necessarily true just before it is applied. Therefore, removing a precondition of an operator while holding the plan fixed does not affect the necessary truth of any condition. Thus, after removing a precondition of an operator in Π, the resulting plan Π' is still correct. This is stated in the following lemma, whose proof follows trivially from the definition of plan correctness:

Lemma 4.4 *Let Π be a plan, and $\alpha \neq \mathcal{G}$ be an operator in Π. Let α' be α with a precondition p removed from P_α. Let Π' be Π with α replaced by α'. If Π is correct, then Π' is also correct.*

Recall that ${}_i\alpha$ is operator α with preconditions ${}_iP_\alpha$ (those having criticality at least i), and effects E_α. Let Π be a plan on the base level. Then for $i = 1, 2, \ldots, k-1$, ${}_i\Pi$ is formed by replacing every occurrence of α (except $\mathcal{I}$ and $\mathcal{G}$) in Π by ${}_i\alpha$.

From Lemma 4.4, the following theorem can be easily proven by induction on the abstraction levels.

Theorem 4.5 *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 $i = 1, 2, \ldots, k-1$.*

For example, it is not hard to verify that Plan 1 in the Towers of Hanoi domain is correct, and that it is also correct when abstracted at each higher level.

This demonstrates that a plan correct at the concrete level is correct at the abstract level. But, since the abstract level no longer contains some of the concrete preconditions, the plan steps that were added solely to satisfy these eliminated preconditions are no longer required. For instance, consider a plan for getting a box from one room into an adjacent room, in which the robot picks up the box, goes to the door, sets the box down, opens the door, picks up the box, and goes through the doorway. Suppose that the status of the door – whether it is open or closed – is ignored at the abstract level. In this case, since opening the door is no longer considered as a precondition, the intermediate steps of setting down and re-picking up the box are no longer necessary; their sole purpose was to free the agent's hands for the door opening. Thus, the abstract level plan is simpler than the concrete level plan. In the next section, we provide a definition of justification, whereby all constraints and operators from a lower level not justified at the abstract level can be safely removed while preserving the correctness of the plan.

4.3.1 Justification

Justification provides a precise specification of those plan steps which are required in order to solve the goal, either directly or indirectly. By definition, we take the initial and goal operators to be justified. Other operators

and constraints are justified if either 1) they establish a precondition of a justified operator, 2) they are a white knight for a justified operator, or 3) they enforce a codesignation or temporal ordering without which a justified operator would be clobbered.

Definition 4.6 *Let* Π *be a plan, and* $\mathcal{I}, \mathcal{G}$ *be the special operators for the initial and goal states. Then in plan* Π*,*

Initial/Goal justification $\mathcal{I}$ *and* $\mathcal{G}$ *are justified,*

Establishment justification *If* β *is justified, and* $\exists \alpha, \exists u \in E_\alpha, \exists p \in P_\beta$ *such that*
Establishes(α, β, p, u)*, then*

> *(1)* α *is justified, (2)* $(\alpha \prec \beta)$ *is justified, (3)* $(u \approx p)$ *is justified*

White knight justification *If* WK$(\delta, \beta, \gamma, p, q, r)$ *and* β *and* γ *are justified, then* δ*,* $(\gamma \prec \delta)$*, and* $(\delta \prec \beta)$ *are justified. Moreover, if* CO *is a set of minimal codesignation and noncodesignation constraints that ensure the white knight relation, then*

1. *all operators whose parameters appear in* CO *are justified, and*
2. *all constraints in* CO *are justified.*

Separation justification *If* γ *and* β *are justified, and* $\exists p \in P_\beta, \exists q \in E_\gamma$ *such that* $(p \not\approx q)$*, then the constraint* $(p \not\approx q)$ *is justified.*

Promotion justification *If* β *and* γ *are justified, and* $\exists p \in P_\beta, \exists q \in E_\gamma$ *such that* $\Diamond(p = \neg q)$ *and* $\Box(\beta < \gamma)$ *then the constraint* $(\beta \prec \gamma)$ *is justified.*

Nothing else is justified.

The justification of plan Π, `Jus`(Π), is the set of operators, precedence and codesignation constraints of Π that are justified, but *not* including any of the operators or constraints that are not justified. It is obvious that the justified version of a plan is simpler than the plan itself, in the sense that the set of operators, and constraints of the justified plan are a subset of those in the unjustified plan.

With a complete plan, justification is a simple recursive definition: any operator that establishes a justified operator is itself justified, with the goal and initial states being trivially justified (Initial/Goal, and Establishment justification). However, with partial plans, the situation is more complex. Figures 3-5 will be helpful in illustrating why the additional justification conditions are necessary. In each of these figures, the boxes represent operators, and the arcs represent necessary precedence. If no arc is between two operators, this means that they are unordered with respect to one another. That is, each precedes the other in some completion. All literals have the same criticality, and in the first two figures, Establishes(a, b, p, u).

In Figure 3, suppose that c clobbers b (CB(c,b,p,q)), and w is a white knight for β (WK(rm w, b, c, p,q,r)). By Initial/Goal and Establishment justification, a, b, and c are justified, but w is not justified, since it is not a necessary establisher. Thus, eliminating the white knight in the justified plan will result in an incorrect plan for every completion in which c follows a. Thus, white knights must be justified (along with their precedence and codesignation constraints).

In Figure 4, suppose that there is a set D of codesignation constraints such that $(\Box(q \neq p))$, whose purpose is to prohibit c from clobbering b. Without Separation justification, c would not be justified, and there would exist a completion of the justified plan in which c clobbers b and for which there is no white knight.

In Figure 5, suppose that if c preceded rather than followed b, c would clobber b. That is, $\Diamond(\neg q = p)$; the precedence constraint (b $\prec$ c) exists to prevent the clobber. Without Promotion justification, this precedence constraint is not justified, and hence, in the justified plan, there exists a completion in which c precedes, and therefore clobbers, b.

For example, consider solving a Towers of Hanoi problem for achieving the goal of having the large disk at `Peg3`, where all 3 disks are initially at `Peg1`. At the lowest level, it is necessary to move the 2 smaller disks from the largest disk before the largest disk can be moved, as in the following.

$$\langle {}_2\texttt{MoveSmall}(\texttt{Peg1},\texttt{Peg3}), {}_2\texttt{MoveMedium}(\texttt{Peg1},\texttt{Peg2}), {}_2\texttt{MoveSmall}(\texttt{Peg3},\texttt{Peg2}), {}_2\texttt{MoveLarge}(\texttt{Peg1},\texttt{Peg3})\rangle$$

Using the abstraction hierarchy in Table 3, the above plan is correct at level 2. However, the first three operators in the above plan do not establish the

preconditions of any other operators. Therefore, they can be safely removed, and the remaining plan

$$_2\texttt{MoveLarge}(\texttt{Peg1},\texttt{Peg3})$$

is still correct at level 2.

Lemma 4.7 *If* Π *is a correct plan that solves goal* G*, then* $\texttt{Jus}(\Pi)$ *also is a correct plan that solves* G*.*

Proof: Suppose by way of contradiction that Π' is incorrect. By definition, $\exists\beta \in \text{Operators}(\Pi'), p \in P_\beta$ such that p is not necessarily true just before β. Thus, one of the two conditions of the modal truth criterion for p must be false.

Suppose the first condition is false; that is, there does not exist an establisher of p for β in Π'. But there must exist an establisher of p for β in Π, since Π is correct. By establishment justification, the establishing operator and constraints are justified, thus serving as an establisher of p for β in Π', a contradiction.

If the second condition is false, then from the MTC, there is a clobberer γ in Π', but not a white knight. γ must also be a clobberer in Π, since by separation and promotion justification, any constraints in Π that would prevent the clobbering would be justified, and hence would also prevent the clobbering in Π'. However, since Π is correct, by the MTC, there is a white knight δ in Π. By white knight justification, δ must also be a white knight in Π', a contradiction. Thus, Π' must be correct. Additionally, Π' also solves the goal, which follows from the definition of justification and a simple induction on the number of operators in Π. □

The following theorem establishes the *Upward Solution Property*: if there is a solution to a problem at the base level, then the justified version of that solution at each higher level of abstraction is correct, and also solves the problem on that level. More formally,

Theorem 4.8 *If* Π *is a correct plan that solves* G *at the base level, then the justified version of* $_i\Pi$*,* $\texttt{Jus}(_i\Pi)$*, is also a correct plan that solves* G *on the* i^{th} *level,* $0 \leq i \leq k-1$*.*

Proof: From Theorem 4.5, 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 4.7, $\texttt{Jus}(_i\Pi)$ is also a correct plan that solves G on level i. □

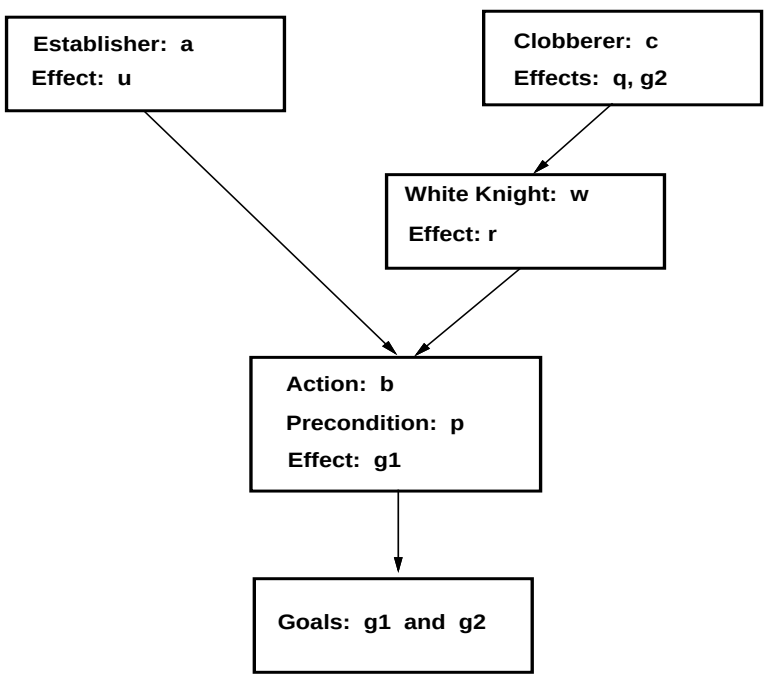

Figure 3: White Knight Justification.

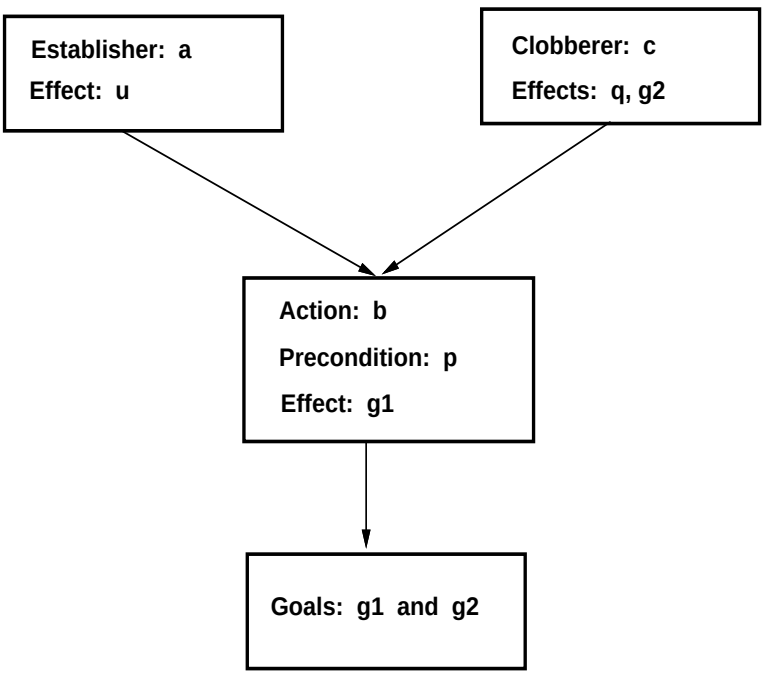

Figure 4: Separation Justification.

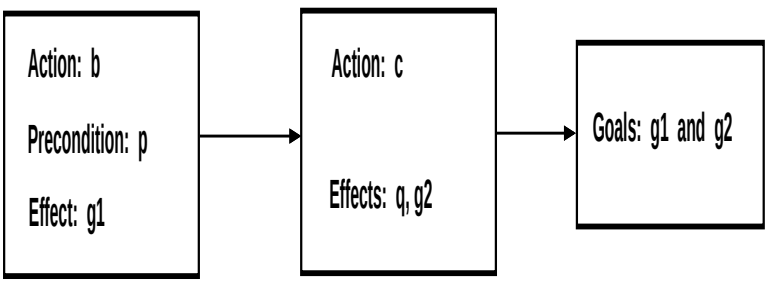

Figure 5: Promotion Justification.

4.4 Monotonic Refinement

The Upward Solution Property guarantees the existence of an abstract level solution to a problem, whenever there exists a lowest level solution. Length-first search, on the other hand, proceeds from the highest level to the lowest. Since the converse of the Upward Solution Property does not hold, one cannot be sure that an arbitrary solution obtained at the abstract level is one which can be refined into a low level solution. It is therefore important to uncover constraints that will be helpful in plan refinement. The monotonic property is one such constraint. This property is defined for linear, ABSTRIPS-type abstraction systems in [4]. We define it here for nonlinear, least-commitment planning abstraction systems.

Definition 4.9 *Let* Π' *be an abstract plan that solves* ρ *at level* i, $i > 0$. Π' monotonically refines *to 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 noncodesignation relations of the plan Π'. Moreover, the establishment structure of Π' is also preserved in Π. This is because justification does not create new establishment relations. Therefore, for the equality, $\texttt{Jus}({}_i\Pi) = \Pi'$, to hold, every establishment relation of Π' must have been one in ${}_i\Pi$. Thus, this definition captures the intuition that one does not want to clobber at lower levels those goals already established at higher levels.

As an example, consider again the Towers of Hanoi problem of moving the large disk to `Peg3` from `Peg1`, with an ABTWEAK hierarchy in Table 3. Recall that the following plan is correct at level two.

$$\Pi' = \langle {}_2\texttt{MoveLarge}(\texttt{Peg1},\texttt{Peg3})\rangle$$

At level one, the plan Π below is a monotonic refinement of Π'.

$$\Pi = \langle {}_1\texttt{MoveMedium}(\texttt{Peg1},\texttt{Peg2}), {}_1\texttt{MoveLarge}(\texttt{Peg1},\texttt{Peg3})\rangle$$

To establish this, note that the abstract version of Π is identical to Π, and that after justification, the operator ${}_2\texttt{MoveMedium}(\texttt{Peg1},\texttt{Peg2})$ can be

dropped without affecting the correctness of the plan. Therefore, it is the case that Π_2, when justified at level 2 is identical to Π.

Definition 4.10 *A k-level* ABTWEAK *system is monotonic, if and only if, for every problem ρ solvable at the concrete (0th) 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 4.11 *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 4.9, Π is a monotonic refinement of Π'. □

5 Search Control

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.

5.1 Completeness of ABTWEAK

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. In particular, abstract plans will constrain lower level search in that every abstract establishment relationship between plan steps will be *protected* during refinement. That is, if at the abstract level Establishes(α, β, p, u), one protects this establishment by ensuring that at lower levels no plan steps are added that clobber p before β (a protection *violation*). 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 [1]. 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 6 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 partial plan, found within a particular abstract level. It is possible that there may exist multiple solutions within a particular level of abstraction. The leftmost solutions shown in Figure 6 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 6, 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.

A search strategy that we employed in our experiments (described in the following section) is breadth-first search in a search space in which each state corresponds to a plan. During each iteration, a plan Π is selected for correctness checking using 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. Although this ensures completeness, it exacts a heavy computational cost. 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 6), 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 6, in breadth-first search, solution plan 2 at level k may be expanded with the same prefer-

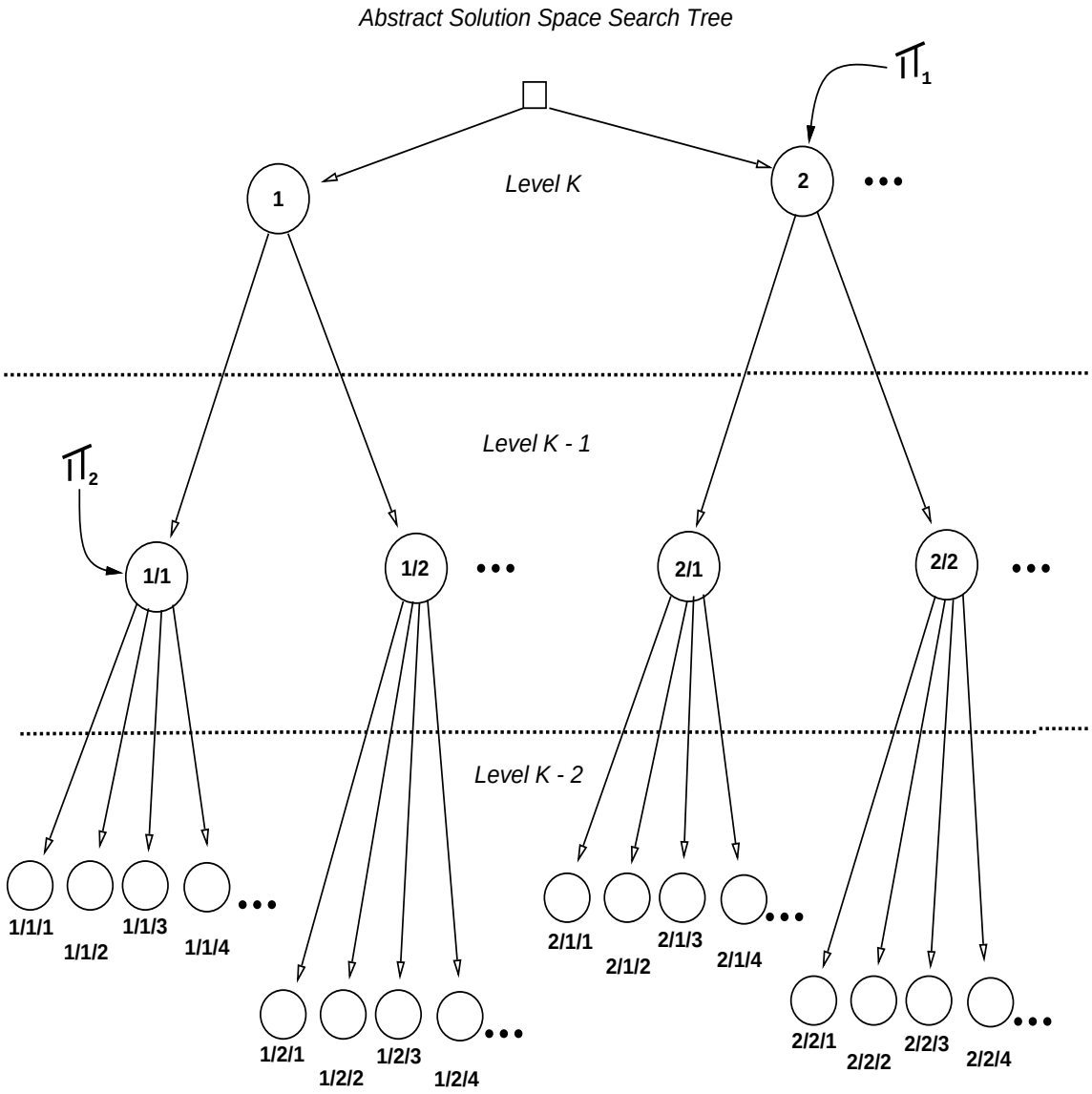

Figure 6: Representing the abstract solution space

ence 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. In the same appendix one can also find a description of how the left-wedge control strategy can be implemented.

5.2 Combining Abstraction with Domain-Dependent Heuristics

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. In this way, the branching factor is pruned, since operators are only considered as plan steps when the current subgoal is a primary effect. 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. In the case with searching only primary preconditions, 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 7 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 [10], in order to eliminate the need for axioms.

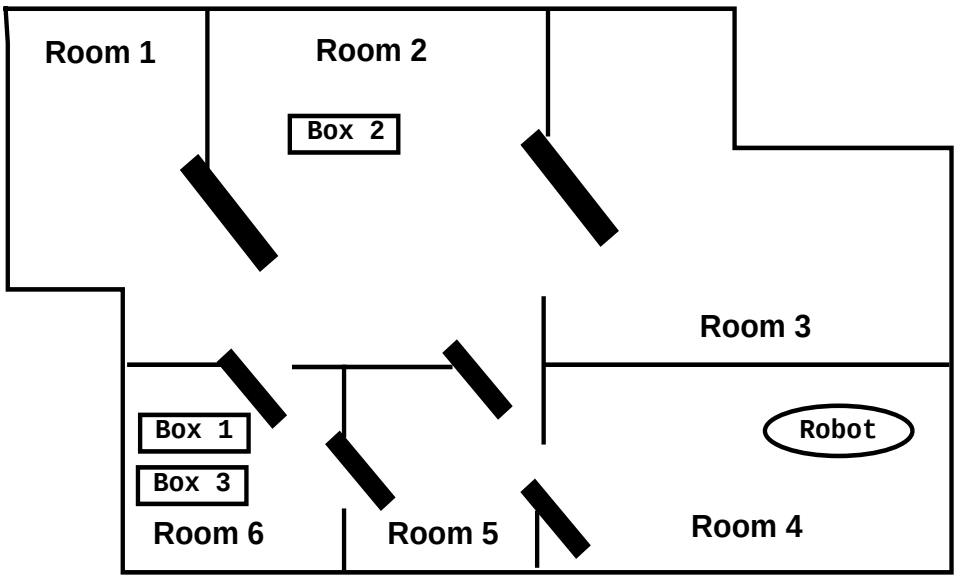

Figure 7: Robot Task Planning Domain.

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`).

To achieve efficiency, many previous planners which have experimented in this domain have used what is known as *primary effects* to achieve goals and subgoals[10, 8, 5]. For example, the primary effect of pushing a box between two rooms r_1 and r_2, is just `Box-Inroom`(Box, r_2). Any additional effects, such as that the robot is also in room r_2, are considered not primary.

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, which can effectively reduce the branching factor of the search space. In fact, without the application of this heuristic, many trivial problems cannot be solved by ABTWEAK.

However, the application of this heuristic threatens the validity of the Upward-Solution Property for some hierarchies. For example, suppose a hierarchy is built by placing all `Robot-Inroom` literals at a higher level than `Box-Inroom` literals. Let a goal be represented by `Box-Inroom`. Then at the `Robot-Inroom` level of abstraction, there is no plan for moving a box across more than one room, simply because the `Box-Inroom` preconditions cannot be observed at that level, and the moving box operators cannot be used for moving the robot around. Thus, although a plan exists for solving the `Box-Inroom` goal at the concrete level of abstraction, no plan can be found at the highest level of abstraction, if primary effects are used. Therefore, the monotonic property is not satisfied for the hierarchy either.

Fortunately, there is one hierarchy in which both the upward-solution property and the monotonic property are satisfied. This hierarchy is described in Table 4. 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. Thus, one always first plans the location of the box before the location of the robot. Doing so while achieving only the primary effects still preserves the completeness of ABTWEAK.

Thus, placing domain-dependent constraints 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 described below:

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* $\mathrm{crit}(e_1) \geq \mathrm{crit}(e_2)$.

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 higher 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 *Suppose that an abstraction hierarchy satisfies Condition 5.1. Let* Π *be a correct, concrete level plan, in which every operator is justified with respect to a primary effect. Then the Upward-Solution Property holds, in that there is also a correct plan* Π_i *at each higher level* i*, in which every operator is also justified with respect to a primary effect.*

Thus, the hierarchy described in Table 4 ensures that the abstract planner remains complete. We omit the proof here, since it is similar to that for Theorem 4.8.

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 SUN4/Sparc station. A detailed explanation of the implementation can be found in [12]. 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 [10]. We have described the Towers of Hanoi domain in detail earlier, with the full operator descriptions 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. Finding empirical measures of "good" abstraction hierarchies. This measure should enable one to predict the performance of planning using any given abstraction hierarchies.

3. 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.

4. 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

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 [5]. 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 three categories: those demonstrating the usefulness of the monotonic property in restricting search, those comparing the left-wedge and breadth-first search strategies, and those establishing empirical criteria for identifying good abstraction hierarchies. 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 Tables 5 to 8, and Figures 8 and 9

6.1.1 Testing the Monotonic Property

Table 5 shows the performance results of ABTWEAK using a breadth-first strategy with monotonic property, while Table 6 shows those without the monotonic property. Overall, breadth-first search using the monotonic property (MP) outperforms search without using the monotonic property, in CPU time, in 15 out of 24 cases of the criticality permutation. 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 property 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. 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. 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. For example, in all cases where OnSmall is above OnMedium and which in turn is above OnLarge, using MP slows down the search. This effect confirms a general 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.

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 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.

We see that when using the MP, the criticality assigned to object-type predicates such as `IsPeg` has a noticeable effect on search efficiency. Tables 2 through 5 give evidence that placing `ISPeg` type of predicate at the highest level of abstraction reduces search.

6.1.3 A Criterion for Good Hierarchies

Figures 8 and 9 display the CPU time as functions of the number of monotonic violations in each experiment. Figure 8 shows the comparison with ABTWEAK using breadth-first search, while Figure 9 shows those with left-wedge search. The correlation coefficient for CPU time versus monotonic violations is 0.68 using breadth-first search, and 0.8 using left wedge. We can thus see a general rule emerging from the results in these figures, that

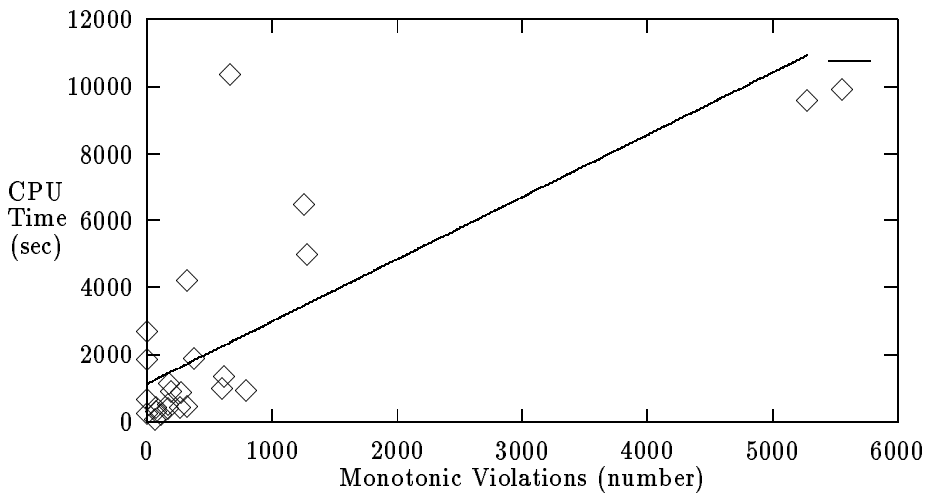

Figure 8: Violations versus CPU time, including regression fit curve. Data are obtained using the breadth-first strategy with MP.

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.

This effect can be explained as follows. The number of such violations measures the number of attempts we make at a particular level of abstraction to undo the work done at a previous level. Undoing an abstract goal or subgoal achievement generally represents a waste in effort spent at the abstract level. Thus, a good abstraction hierarchy should arrange the subgoals at each level, such that the chances for such violation is minimal.

Hence, a good abstraction hierarchy is one with few monotonic violations. One application of this criterion is to identify a good hierarchy via *syntactical analysis* of the operator set. For example, it has been shown from the operator definitions that, for the hierarchy ILMS, no monotonic violations will ever occur during plan refinement for any refinement of the abstract plan[4]. Knoblock ([5]) provides an example and algorithms of this type of syntactic analysis.

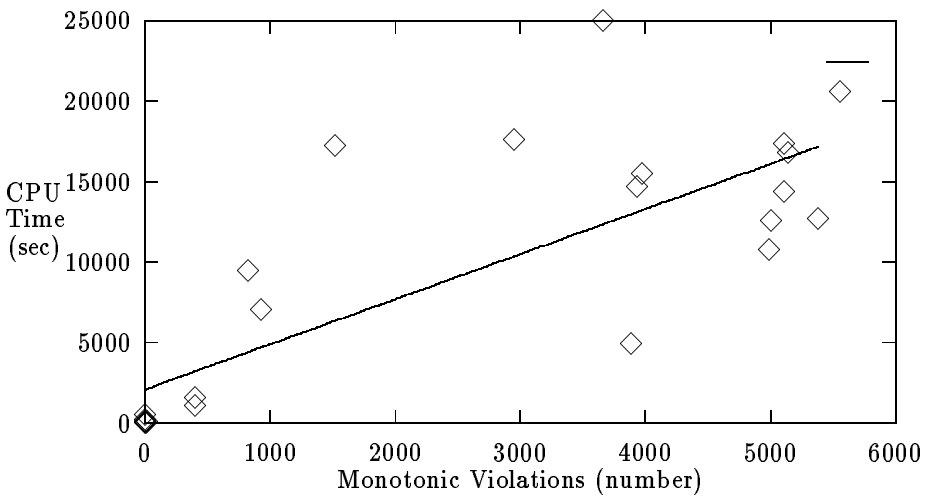

Figure 9: Violations versus CPU time, including regression fit curve. Data are obtained using the left-wedge strategy with MP.

6.1.4 Testing the Left-Wedge Control Strategy

Tables 7 and 5 provide a comparison of ABTWEAK using left-wedge and breadth-first, 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. However, no improvement, or even a decrease in performance is seen for certain other criticality assignments, notably IMLS, IMSL and SMLI. 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 the monotonic property (Tables 7 and 8), the results indicate that, in general, using the monotonic property with left-wedge works well with good criticalities (those which generally result in few protection violations), and poorly with bad criticality assignments (those resulting in many protection violations). This result again confirms our conclusion above. ABTWEAK with the monotonic property clearly outperforms one without the monotonic property in the three hierarchies ILMS, ILSM and IMLS. However, for hierarchies ISLM and ISML it appears that not using the monotonic properties 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 the monotonic property 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 violating the monotonic property. 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.

6.1.5 Comparing TWEAK with ABTWEAK

The utility of abstract search will not be completely understood without also comparing it with 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.

Figure 10 shows the result of the comparison in the Towers of Hanoi domain. In this test, AbTweak was run with both the monotonic property 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.

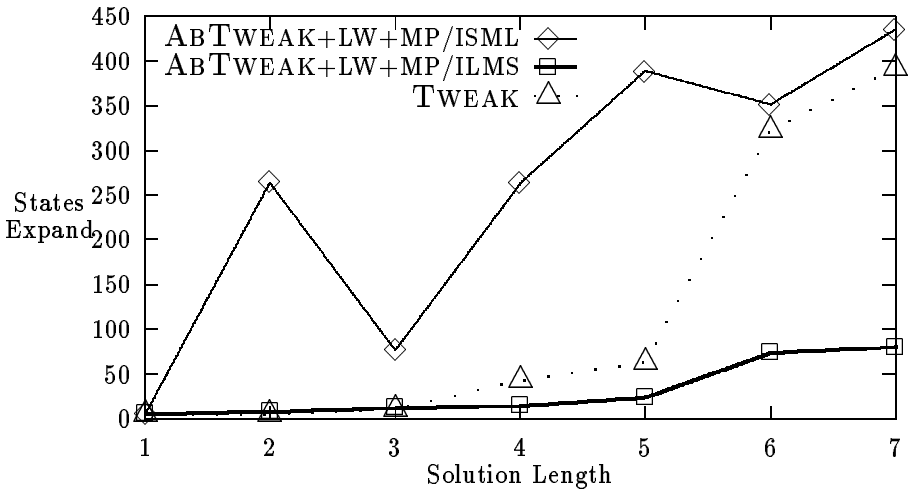

Figure 10: Comparing Tweak with AbTweak.

The same figure also compares the performances 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 [4] and [5].

6.2 Robot Task Planning Domain

We have run 50 tests of AbTweak with the hierarchy in Table 4 using the primary effects heuristic. Without this heuristic, many simple problems were not solvable by any of the planners that we tested. Five different planning problems of each length were solved using Tweak, AbTweak with breath-first, and AbTweak with both the monotonic property 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 11 shows the number of states expanded as a function of solution length. It is clear that AbTweak with the monotonic property and the left-wedge control strategy dramatically outperforms both Tweak and AbTweak with only the breath-first control strategy.

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

Table 4: Criticality assignments for the Robot Task Planning Domain.

6.3 Summary of the Experiments

To sum up, we make the following observations from the experiments:

1. Using the monotonic property in abstract planning is often more advantageous than without, especially when difficult-to-achieve conditions

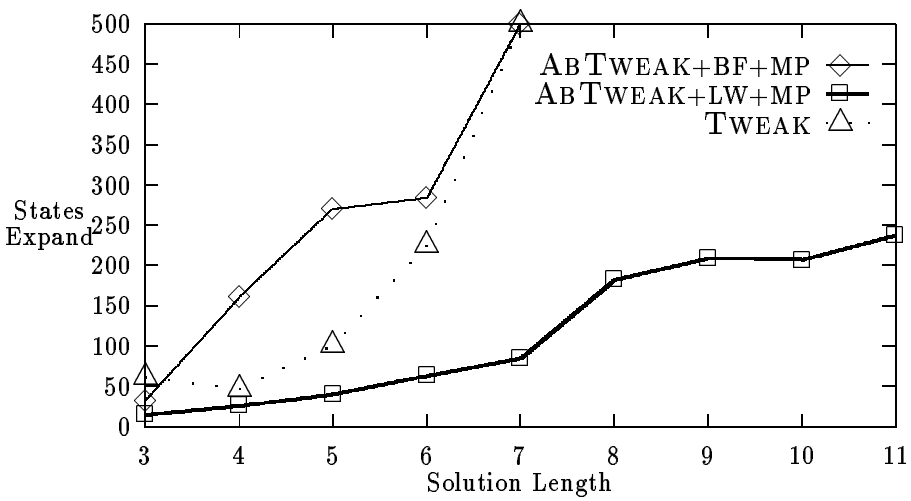

Figure 11: Comparing Tweak with AbTweak in the robot task planning domain.

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 property can reduce search efficiency.

2. The number of monotonic violations provides a criterion for judging the effectiveness of an abstraction hierarchy. The fewer there are, the better the hierarchy. This motivates the investigation of syntactical restrictions on the criticality assignments which can guarantee good performance.

3. Having a good abstraction hierarchy is not sufficient for superior performance over planning without using abstraction. An additional requirement is a control strategy that takes advantage of the structure of the hierarchies. Left-Wedge is one such strategy that preserves completeness and outperforms TWEAK, on good abstraction hierarchies.

4. Certain domain-dependent heuristics, such as the use of primary effects in goal-achievement, can jeopardise 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.

5. The placement of object-type predicates, such as `IsPeg`, can affect the efficiency of a hierarchical planner dramatically. Our general conclusion is that placing them at the highest level of abstraction is almost always better.

7 Conclusion

This research has been aimed at formalizing and testing domain-independent, nonlinear planning systems that plan in hierarchies of abstraction levels. The

Table 5: AbTweak With Monotonic Protection and Breadth-first search. Search space expansion bound: 5000 expanded.

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	5001	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	5001	12496	665	10339.2
SLMI	640	1388	200	892.3
SMLI	2519	6795	1286	4987.9

Table 6: AbTweak using Breadth-first search, and without Monotonic Protection. Search space expansion bound: 5000 expanded.

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	5001	13029	0	6849.5
SLMI	2940	7619	0	3821.4
SMLI	2359	6445	0	2892.1

Table 7: AbTweak using Left-wedge search, and with Monotonic Protection. Search space expansion bound: 5000 expanded.

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	5001	6268	3979	15491.3
ISML	5001	5841	5391	12688.5
LIMS	56	98	0	79.1
MILS	94	138	21	81.3
MISL	5001	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	5001	6011	5116	17385.6
LSIM	1587	2116	931	7030.2
SLIM	5001	6592	3670	24998.1
SMIL	5001	6161	5151	16803.1
LMSI	250	636	0	540.5
MLSI	170	363	9	255.1
MSLI	5001	13780	3935	14721.4
LSMI	3142	7587	831	9513.8
SLMI	5001	12050	1526	17247.3
SMLI	5001	15215	4991	10773.0

Table 8: AbTweak using Left-wedge search, and without Monotonic Protection. Search space expansion bound: 5000 expanded.

Hierarchies	Expanded	Generated:	MP Pruned:	CPU Seconds:
ILMS	57	99	0	30.5
IMLS	1009	1811	0	1470.5
IMSL	5001	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	5001	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	5001	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	5001	14416	0	9294.4
LSMI	4737	12387	0	9627.0
SLMI	5001	13633	0	10813.7
SMLI	5001	14571	0	10189.5

resulting planner, ABTWEAK, extends the precondition-elimination methods in ABSTRIPS for building abstraction hierarchies, and allows for a least-commitment representation of plans in TWEAK. We have shown that ABTWEAK satisfies the monotonic property; that is, as planning descends from abstract to concrete levels, the precondition establishment structure of a plan need not be changed. This, to a large extent, formalizes our intuition for using abstraction in planning: that it is generally more efficient to use an abstract solution to guide search at lower levels of abstractions than without abstraction. In addition, 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 show optimizes performance over those abstraction hierarchies having fewest monotonic violations.

In the future, we plan to further investigate criteria for checking good abstraction hierarchies, as well as extend the present framework of ABTWEAK to include also other types of abstraction hierarchies used in practice.

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.

References

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

[2] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, 1985.

[3] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach.* PhD thesis, University of Maryland, College Park, Maryland, Oct. 1989.

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

[5] 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.

[6] Richard Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1985.

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

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

[9] Nils Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers Inc, 1980.

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

[11] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, CA, 1988.

[12] Steven G. Woods. An implementation and evaluation of a hierarchical non-linear planner. Master's thesis, Computer Science Department, University of Waterloo, 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* $crit(precond) > crit(\Pi)$ **Then**
  $Est :=$ an abstract establishment relation, Establishes$(\alpha, User, precond, e_\alpha)$,
      that has been clobbered at the current level,
  $succ := \{(\Pi, Est)\}$
Else
    Let *Old* be the set of operators in Π which 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
      $Est =$ Establishes$(\alpha, User, precond, e_\alpha)$ holds
      (2) $succ := succ \bigcup \{(\Pi', Est)\}$
Endif
{ **Declobber** }
For each pair (Π',$Est =$ Establishes$(\alpha, User, precond, e_\alpha)$) 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) $successors := successors \bigcup \{\Pi_1, \Pi_2, \Pi_3\}$,
      {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 each plan Π' in *successors*, **Do**
    If there is an operator γ and an abstract establishment relation Establishes$(\alpha, \beta, p_\beta, e_\alpha)$ such that

    {Note: This condition defines monotonic violation.}
      (1) $\Box(\alpha < \gamma < \beta)$,
      (2) For some effect e_γ of γ,
        either $\Box(e_\gamma = p_\beta)$ or
        $\Box(e_\gamma = \neg e_\beta)$.
    Then *successors* := *successors* $- \{\Pi'\}$
    Endif
  Endfor
Endif
Return *successors*

C.4 Cost Function and Left Wedge Implementation

The cost function $cost(\Pi)$ can be defined as the total number of operators in Π, if a breadth-first control strategy is used. The left wedge heuristic is implemented by adding to the cost function an additional value, which depends on the level of abstraction:

$$cost(\Pi) = |\text{Operators}_\Pi| - lw(crit(\Pi)),$$

where $lw(i)$ is any monotonically decreasing function of i, such that $lw(k-1) = 0$, for a hierarchy with k levels of abstraction.

C.5 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.