An Implementation and Evaluation of a Hierarchical Nonlinear Planner

by

Steven G. Woods

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1991

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Planning is the process of generating sequences of actions in order to provide a method for agents to modify the state of the world in which they exist. It is well known that the application of abstraction can greatly reduce the search involved in creating plans. In this talk, an empirical evaluation of several different approaches to reducing search in AbTweak, an abstract nonlinear planning system, are presented.

To solve a complex problem using abstraction, one would like to protect parts of the abstract solution that have already been completed during the abstract plan refinement. However, it has not been well understood how this protection can be done profitably; some subgoal protection may indeed result in a decrease in efficiency. The Monotonic Property has been proposed as a form of goal protection in abstract planning. Different versions of this property are investigated with respect to their effect on planning performance in several domains. Furthermore, a series of empirical tests of the utility of the properties are presented, along with an evaluation of different search strategies for abstract planning.

In addition, we present a novel abstract planning control strategy known as Left-Wedge. This strategy adds a depth-first flavor to a complete search strategy by taking advantage of the fact that less abstract solutions are generally closer to a concrete level solution than more abstract ones. The relative benefit of this approach under various criticality assignments is demonstrated through comparisons with breadth-first strategy. Furthermore, two subgoal selection strategies are presented and compared empirically.

Acknowledgements

First and foremost I would like to thank my supervisor, Qiang Yang for his endless suggestions, guidance, and constructive criticism over the time I have known him. The many hours he spent discussing planning, AI in general, and Lisp helped me to stay focused on my topic and remain motivated through some frustrating times. If ever someone had an open-door policy, it is Qiang.

Thank you to my readers Fahiem Bacchus and Grant Weddell for the time they took to carefully read my thesis and offer many constructive commments. Thank you to Josh Tenenberg for taking the time to discuss many of the aspects of abstract planning mentioned in this thesis, and to Bruce Spencer and Peter Van Beek for suggestions and help on work that never made it. Thanks Keith Mah, Hamish Macdonald, Verna Friesen and John Sellens for help when the really hard work of getting LaTeX, Emacs, and system bugs out needed to get done.

Very special thanks to Verna Friesen for listening to my problems, making time to talk about my work, and for her immense patience, support, love, and unfailing ability to make every day fun.

Final thanks go to my family and friends, for much love, encouragement, and support across quite a few miles.

Funding for this thesis was made available in part by NSERC operating grant OGP0089686 to Dr. Qiang Yang.

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge but you have scarcely, in your thoughts, advanced to the stage of science. [1]

For my parents Dorothy and Frank ...

[1] William Thompson, Lord Kelvin [1824-1907], from *Popular Lectures and Addresses*, 1891-1894.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Traditional planners attempt to create a set of ordered actions, or a plan, which specifies the manner in which some particular system can modify its "world" from its existing or initial state to some desired final state. Robots are an example of one class of system which possess a set of explicit executable actions or operators which must be applied in some order in attempting to modify a given environment in some desired manner.

There are many divergent approaches to solving planning problems. Certain planning approaches attempt to use existing plans (plan reuse, [Kambhampati, 1989, Fikes *et al.*, 1972]) to solve problems that occur, others attempt to take action in a particular domain or world by repeatedly taking small courses of action, re-evaluating the world, and acting again (reactive planning, [Agre and Chapman, 1987]). Still other approaches attempt to interleave the creation of plans and their execution ([McDermott, 1978]).

Traditional planning is characterized by an approach which treats every attempt at plan creation as novel. Plans are created from only a set of action templates, an initial state description, and a goal state description.

Planning strategies can also be divided into those that plan in either *linear* or *nonlinear* manners. Nonlinear planners tend to avoid or postpone committing to a total ordering of all operators in a plan as long as possible, while linear planners commit to specific total orderings in an attempt to find a single ordering that adequately solves the planning task in question. A nonlinear plan, which possesses only a partially ordered operator set, represents a collection of totally ordered, linear plan completions.

Many different traditional planning approaches have been investigated in the search for systems that can create plans efficiently and accurately, including those seen in NOAH [Sacerdoti, 1977], STRIPS [Fikes and Nilsson, 1971], and TWEAK [Chapman, 1987]. Some of these approaches attempt to address the planning problem in a very general, or *domain-independent* sense, with little or no specific reference in terms of control strategy to the characteristics of a single domain or problem type. Other approaches attempt to utilize knowledge about specific domains to improve search control through *domain heuristics*, and are known as *domain-dependent*. These heuristics help to control the planner operation by taking advantage of existing knowledge about problem solving in these domains. Traditional planning approaches utilize strategies with certain aspects of both domain-independent knowledge about planning, and domain-dependent knowledge about problem solving.

One domain-independent method of reducing search that has been suggested is the use of abstraction in planning. In solving complex planning problems, one would like to distinguish parts that are crucial and difficult to solve from those

that are less important and easy to solve. Towards this end, research in abstract planning has been particularly active, ranging from the exploration of different types of abstraction systems [Christensen, 1990, Knoblock, 1989, Sacerdoti, 1974, Sacerdoti, 1977, Wilkins, 1984], to the formal characterization of the intuition behind abstraction in planning[Knoblock, 1990, Korf, 1985, Tenenberg, 1988, Yang and Tenenberg, 1990].

This thesis presents an evaluation of an abstraction based planner known as AbTweak, described in [Yang and Tenenberg, 1990]. AbTweak is an extension of Tweak, a non-abstract, nonlinear, least-commitment planner, described in [Chapman, 1987]. In [Yang and Tenenberg, 1990], a general property is defined which can limit the possible search paths required to find a goal in abstract search. Experiments evaluating the benefit of applying this property in abstract search control are presented in this thesis, and a novel complete search strategy for AbTweak is proposed and evaluated empirically.

1.2 Motivation

In attempting to construct AbTweak, it was first necessary to construct the simpler, nonlinear planner, Tweak. The intent was to reconstruct Tweak and then enhance it so that it was capable of abstract planning as described for AbTweak by Yang and Tenenberg. Once implemented, quantitative analysis of abstract properties suggested by Yang and Tenenberg to enhance abstract planning performance was made possible. The question of selecting a suitable abstraction hierarchy for AbTweak soon became evident, and a correlation between the properties defined in [Yang and Tenenberg, 1990], the abstraction hierarchies selected, and overall planning performance became obvious. These results offer some insight into the

potential benefit of utilizing abstraction in planning to limiting abstract search, and of evaluating potential abstraction hierarchies.

During the course of implementing and enhancing AbTweak, two very different dimensions of control became evident.

The first, or highest control dimension, consists of controlling the search through the set of all possible correct abstract plans. Multiple correct plans can exist at various levels of abstraction, and in order for a planning system to be complete, each of these correct plans must be viewed as being capable of leading to a concrete solution plan. The implementation of search in AbTweak is a simple application of a well-known breadth-first search strategy, A^* [Hart *et al.*, 1968, Nilsson, 1980]. While searching for a more efficient method of controlling planning search, a novel method was devised, referred to as Left-Wedge. Experimental results presented in this thesis contrast and compare the performance of breadth-first AbTweak and Left-Wedge AbTweak.

The second, lower level control dimension, consists of determining in what manner an individual plan should be modified in order to try and make it correct at a particular level of abstraction. This dimension involves selecting unsatisfied plan subgoals to solve, and rejecting portions of the planning space that need not be expanded en-route to a solution.

Both abstract and the individual level control strategies will be explained and examined in this thesis, and empirical results will be presented showing what combinations of planning strategies and properties result in the best abstract search performance.

In this thesis several implemented planning strategies are presented within a single view of the planning search space. Abstract planning as defined in AbTweak

is essentially a controlled search through a space of incomplete plans, where different plans have different corresponding levels of abstraction. The search space continually grows as planning continues, until a correct plan is located in the space. Within the context of AbTweak, and throughout this thesis, abstract plans can be pictured as "simpler" versions of plans, while concrete plans are more "complex". Abstracted versions of concrete plans are achieved via the elimination of certain less-important or critical preconditions. For example, if a certain incomplete plan has an operator with the conjunctive goals onbig(Peg1) and onsmall(Peg1), and the goal onsmall(Peg1) was somehow determined to be the least critical in some domain, then an abstraction of this operator might possess the goal operator with only one precondition, onbig(Peg1). Planning in an abstract fashion entails creating plans in a "top-down manner", first creating a solution to a problem at the next more abstract or "simplest" level, progressing down to the most concrete, or "complex" level iteratively until a lowest level solution is found.

Within this search framework, there are two separate, but not unrelated aspects of search control for an abstract planning system such as AbTweak. The first aspect of control is concerned with search *within* each individual level of abstraction, and the second is related to the coordination of search *across* different levels of abstraction. A non-abstract planner, such as Tweak, has only one control dimension within this framework: control within its single (concrete) abstraction level.

Within the first dimension of control, choices must be made about which plans show promise in the search for problem solutions to a problem at a particular abstraction level, and in fact, about how to best expand the chosen plans en route to a solution. Determination of the successors that show the most promise towards a solution, and of the goals within selected plans that will provide successors resulting in efficient planning search must be made.

The determination of promising successor plans is closely related to the formalization of the intuition behind successful abstraction hierarchies and systems. When an abstract solution is formed, the contribution of each operator in the plan as it relates to the solution of the goal can be determined by looking backward from the goal. This precondition/effect chaining will be referred to as the *subgoal establishment structure* of the plan. This structure, when passed down from an abstract plan to more concrete plans, can serve as a constraint during plan refinement at lower levels. Those parts of the problem satisfied by the abstract plan can be protected to some degree. For example, in Figure 1.1 we see an abstract or high level complete plan for opening a door. The operator OPEN-DOOR in this example has a single precondition we will view as a subgoal: stand-at-door. This former subgoal is accomplished by the effects of the operator GOTO-DOOR. After adding this operator to accomplish stand-at-door, this plan has a subgoal establishment structure that can be represented something like:

Establishes(GOTO-DOOR, OPEN-DOOR, stand-at-door)

where *Establishes* is a relation describing an establishing operator, the operator that benefits from the establishment, and the established subgoal.

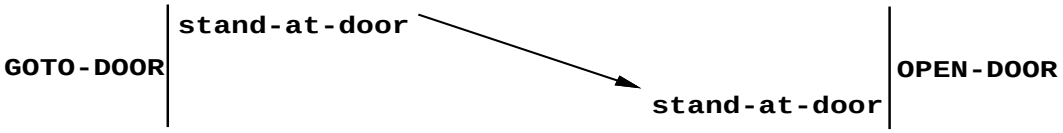


Figure 1.1: Simple subgoal establishment structure

Consider a descendant of the plan shown in Figure 1.1 at a lower level of abstraction. Suppose this descendant plan contains an operator corresponding to

OPEN-DOOR in Figure 1.1. This operator might have additional concrete preconditions or subgoals, such as have-key. If we now attempt to solve this new subgoal, we are constrained by the *Establishes* relation achieved earlier. The addition of an operator to satisfy have-key must not interfere with stand-at-door. For example, we cannot insert an operator between GOTO-DOOR and OPEN-DOOR if that operator has an effect that opposes stand-at-door. Consider the case where he operator GOTO-KEY-RACK achieves stand-at-key-rack, and has a side effect NOT stand-at-door. If the key were hanging on the wall, then any attempt to insert this GOTO-KEY-RACK operator to achieve stand-at-key-rack must be rejected.

In fact, without constraints of this nature passed down from the abstract levels, there seems little justification in using abstraction since high level subgoal establishments could be destroyed. This destruction of previous establishments essentially throws away work previously done in constructing the plan. To maintain search efficiency, one must make decisions regarding what subgoals to protect, how to protect them, and when. For example, is it more advantageous to backtrack when some previously satisfied goals are in danger of being undone, or to simply allow this destruction to occur, and re-plan for those goals later?

In addressing the subgoal protection problem within each abstraction level, we investigate the utility of protecting subgoals with a nonlinear, hierarchical planner AbTweak. Across different hierarchies within various domains, we test several different implementations of the *Monotonic Property* (MP) [Yang and Tenenberg, 1990], a property which allows for the application of goal protection within abstract planning while not destroying the the planner's ability to find a solution. The results demonstrate that depending on the search strategies used across different levels of abstraction, protecting higher level subgoals may not always improve the efficiency of planning. With analysis of these results, we point out several ways of making

the improvement possible.

In addition to subgoal *protection*, the method of subgoal *selection* within a particular abstract search level can dramatically affect search performance. The strategy used in selecting which operator and which subgoal within an operator to attempt to satisfy next in a planning process determines the branching factor at each branch point in the individual level search space. Subgoal selection strategies can be structured to reflect *casual* or *strict* approaches to retaining the planning effort expended within a level of planning. Strict selection entails selecting new subgoals with the intent of repairing existing subplans that have become "buggy" as planning progressed, while casual selection implies selecting the high level or general goals, and leaving previously created subplans unrepaired until later in the planning process. Empirical results that suggest casual approaches to goal selection outperform strict ones are presented.

The second dimension of search control while planning with abstraction involves search through all of the possible solutions at various levels of abstraction. Only one solution at each abstract level is committed to at any one point in the search process. This search must be performed carefully, in order to preserve completeness of a given search strategy. Even though abstraction has an intuitive appeal as a result of its straightforward, top-down nature, there is no guarantee that a complete search with abstraction will outperform one without abstraction. Since an explanation for this behaviour does not seem obvious, it is important to investigate various search strategies in an attempt to explain exactly when and how abstraction can benefit planning.

In this thesis, a method of implementing a complete search strategy across different abstract levels is examined. This new search control strategy, called the LEFT-WEDGE strategy, exhibits some depth-first behavior by assigning a higher

priority in search to a plan (prefers) that is more refined than others. For some hierarchies, search using the LEFT-WEDGE strategy is constrained to occur within a considerably smaller search space, and as a result, search efficiency with abstraction can be dramatically improved. Explanations for this search behaviour will be discussed later in this thesis with specific references to example problems.

This thesis reports an empirical study in addressing the above two control problems: controlling search within each level of abstraction, and controlling search across the abstract levels.

1.3 Outline

The organization of this thesis is as follows. Past work in linear, nonlinear, and abstract planning will be outlined briefly in Chapter 2 in order to give the reader some insight into the factors which have driven research into the more detailed areas of planning presented in this paper. Next, Chapter 3 will present the specific implementations of TWEAK and ABTWEAK with particular attention paid to the manner in which each implementation captures the nonlinearity in TWEAK, and the abstraction in ABTWEAK. In addition, an outline of the strategies utilized for search both through the space of abstract plan solutions, and within each level of abstraction, is given. Within the context of TWEAK and ABTWEAK, experiments are described in Chapter 4, and results presented. Results are evaluated with emphasis on explaining planning behaviour under various search strategies and applications of goal protection. The thesis concludes in Chapter 5 with a summary of the most interesting aspects of the experiments, and an outline of possible directions for future investigation of nonlinear, abstract planning strategies.

Throughout this thesis, planning examples are given both graphically and within

the context of discussion. In these examples, a convention is adopted of identifying propositions in lower case, and operators in upper case. Further, an operator's effects will be placed to the *right* of an operator in graphical examples, while the operator's preconditions will be on the *left* of the operator. Directed arrows between operators indicate the presence in the example plan of an ordering constraint between the two operators.

Chapter 2

Background

2.1 Linear Planning

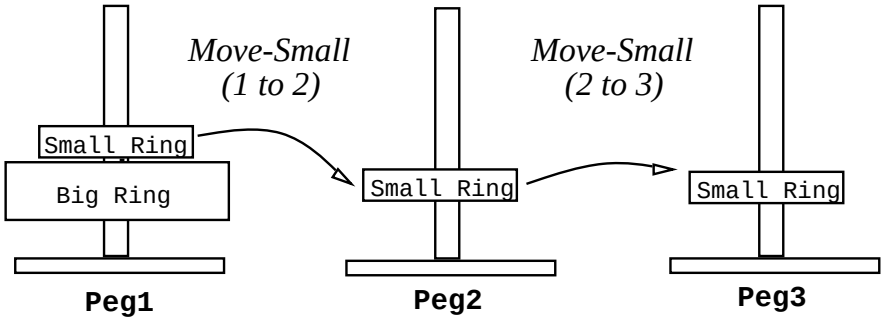
Domain-independent planners deal with goal interactions based upon one of two assumptions. Either the assumption is made that subgoals are basically independent, and hence the order in which they are achieved is irrelevant, or it is assumed that solution order will matter, and commitment to any single ordering should be postponed as long as possible while planning. The former assumption is made by linear planners such as STRIPS [Fikes and Nilsson, 1971] and HACKER [Sussman, 1973], the latter by nonlinear planners such as NONLIN [Tate, 1977], MOLGEN [Stefik, 1981], and SIPE [Wilkins, 1984].

The *linearity assumption* that subgoals are independent is basically a domain-independent heuristic for planning. Linear planners select a subgoal, apply an operator to solve this subgoal, and then choose another subgoal to solve, thus adding to the plan's growing linear set of operators. If at any point the plan's subgoals do interact in that one operator cannot be ordered before another and

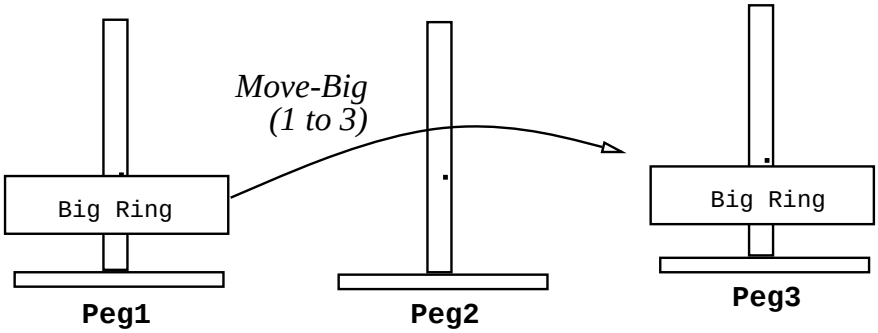
still have the plan solve both subgoals, the planner must backtrack to try solving the subgoals in a different order.

Unfortunately, many planning problems exist in which subgoals are not independent. Some problems require a set of operators solving one subgoal completely to be interleaved with another set of operators solving a separate subgoal. For example, consider the problem shown in Figure 2.1 on page 13, where there exist 3 pegs, and 2 different sized rings, both on the first peg, where the small one is above the large one. Suppose that a peg may only be stacked on an empty peg or a larger ring. If we wish to transfer both rings from the first peg to the third, two subgoals might be On-Small(Peg3) and On-Big(Peg3). A linear planner might select the On-Small subgoal first. An operator to accomplish the movement of the small ring from the first peg to the third would be added to the plan, say MOVE-SMALL(Peg1 Peg3). The planner now selects the second subgoal, On-Big. Unfortunately, the large ring cannot be moved to the third peg as it is larger than the small ring. The planner would backtrack and attempt another operator ordering. The actual solution to this problem involves interleaving the solution to the On-Small subgoal with the On-Big subgoal solution. The small ring must be moved to the second peg as an intermediate step, the large then to the third peg, and finally, the second moved to the third peg.

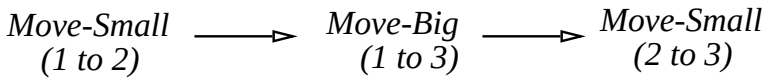
Linear planners are unable to solve these types of problems efficiently, and depending on how the operators are added to a plan, may not be able to solve them at all. Many other problems exist, even in very simple domains which allow for only one correct goal ordering to satisfy multiple or conjunctive subgoals. Problems with pairs of subgoals such as Make-the-Door-Open and Stand-In-Room have a definite implicit relationship that can only be reconciled by ordering the operators OPEN-DOOR before GO-INTO-ROOM. A planner that solved each of these goal



Solution to the first subgoal, On(Small-Ring Peg3)



Solution to the second subgoal, On(Large-Ring Peg3)



Interleaved solution for both subgoals

Figure 2.1: Simple Example of Subplan Interleaving

independently, and only committed to an ordering when it became apparent that a conflict existed between the two is known as a nonlinear planner. A nonlinear planner is a *least-commitment* planner, at least in terms of action ordering.

Just as some planners "overcommit" to operator ordering, others overcommit to the selection of certain objects when solving subgoals. In Figure 2.1, the subgoal On(Small-Ring Peg3) could be solved by an operator MOVE-SMALL($some-peg Peg3) rather than by MOVE-SMALL(Peg1 Peg3) [1]. The former operator is not committed to any particular original position of the small ring. The combination of a *least-commitment* approach to both operator ordering and object bindings results in a planner, like TWEAK, capable of solving problems with more complex goal interactions than is the case with over-committed planners like STRIPS.

2.2 Nonlinear Planning

As stated previously, the nonlinear approach to planning represents a domain-independent heuristic. The main assumption of these planners being that the ordering of solutions to subgoals will make a difference in the efficient generation of plans. A nonlinear planner can be viewed as an extension to the STRIPS approach where a partial temporal ordering is allowed on plan operators rather than a total ordering, and partial constraints on variable bindings of these operators are allowed in place of commitment to particular bindings.

As early as 1975, a planner (NOAH, [Sacerdoti, 1977]) existed which allowed plan operators to be left unordered until a need to linearize them became apparent. NOAH utilized a set of plan modification operators that allowed for the resolution

[1] The prefixing of a value by $ such as in $X indicates X is a variable, while X alone indicates X is a constant

of interactions among subgoals. Subsequent planners improved on the concepts presented in NOAH. In 1976, a planner (NONLIN, [Tate, 1977]) with a backtracking top-level control structure and an improved plan modification operator set emerged. In 1981, a planner (MOLGEN, [Stefik, 1981]) was described by Stefik, which viewed action orderings and object instantiations as *constraints* on a plan, and the addition of these in resolving plan conflicts as *constraint posting*. SIPE [Wilkins and Robinson, 1981, Wilkins, 1984], in a manner similar to MOLGEN, utilized constraints, although SIPE also treated objects in the plan as limited resources which had to be shared among operators.

In 1986, David Chapman [Chapman, 1987] created a nonlinear planner TWEAK, which he proved to be both correct and complete; correct in that any plan it returned solved the problem given it, and complete in that if a solution did exist, it would eventually find it. Chapman's intention in building and describing TWEAK was to clarify past planning research, and implement a planning algorithm that was straightforward enough so that its operation and construction would be provably correct, and comprehensive enough to encompass the goal modification methods of previous planners. He wished to use TWEAK merely as the planning portion of an integrated problem solver, and he simply wanted to use a state of the art planner that he could use with confidence. As a result of Chapman's resolve to build an understandable planning system, his description and discussion of planning tends to give rigor and neatness to the paradigm of nonlinear planning within the context he describes. If one is interested in examining the nature of solving planning problems in a domain independent fashion, with emphasis on learning about subgoal interactions, then it becomes apparent that the neatness and rigor that the nonlinear planner TWEAK possesses make it an interesting planner to implement, examine in detail, and to base future planners upon.

2.3 Tweak

This section describes Chapman's TWEAK. TWEAK is a domain-independent, conjunctive-goal planning system (DICP). DICP describes a planning method of achieving several goals simultaneously (conjunctive) in a generally useful or reusable manner (domain independent). The problem of interacting subgoals arises in this type of planning, a classic example of which is the Sussman Anomaly, [2] where the work done by a planner in achieving one goal is undone in the achievement of another goal. A class of planning known as *nonlinear planning* [Sacerdoti, 1975] has been shown to solve this problem: it is capable of solving conjunctive goal problems where solutions interact.

2.3.1 Background

Chapman had heard that Sacerdoti's NOAH [Sacerdoti, 1977] constituted the state of the art in planning. After several failed attempts to implement NOAH from its description in the aforementioned reference, he succeeded in creating a working planner. Chapman points out in his paper that to use some routine as a part of a system (in this case, the NOAH based planner in his integrated problem solver), it is important to know for certain that it works. In the course of showing that the planner was reliable, Chapman proceeded to simplify the algorithm of NOAH to justify the plan modification methods in NOAH rigorously, and to integrate the methods presented in other planners post-dating NOAH.

Of concern to Chapman was both the correctness of the planner algorithm he was to use, and its completeness. Sacerdoti states in [Sacerdoti, 1977] that:

[2] Chapman indicates in [Chapman, 1987] that this well-known example is actually due to Allen Brown, although popularly known as the Sussman Anomaly

> ... it should be possible to define an algebra of plan transformations...a body of formal theory about the ways in which interacting subgoals can be dealt with.

Chapman attempted to address this challenge directly in his construction and description of TWEAK. His solution is composed of three main parts:

1. The manner in which a plan is represented.
2. The way in which plan subgoals are achieved.
3. The method in which search through the space of all possible plans is controlled.

I will discuss each of these parts in the following section on the design of TWEAK.

2.3.2 Tweak Design

2.3.2.1 Tweak Plan Representation

Chapman [Chapman, 1987] describes at some length terms and definitions for the many parts of his planner. In the interest of both clarity and brevity, I will not repeat these at the length he does in his paper. However, I will attempt to capture the essence of the plan representation he describes. TWEAK makes plans by incrementally specifying partial descriptions or constraints that the plan must fit. This method is known as *constraint posting*. Constraint posting equates to a search methodology which progressively removes portions of the search space through addition of constraints which rule them out. The advantage of this *constraint* approach

is that a reduction in the amount of strictly arbitrary choice is enjoyed, and a corresponding reduction in the amount of backtracking required in planning is realized. When TWEAK is working on a problem, it maintains a specification of the planning it has done to a given point, specifically the incomplete plan, which is a partial specification of the eventual complete plan. TWEAK keeps on adding constraints to this plan until all of the possible completions of the plan necessarily solve the problem. We can say that a proposition which is satisfied in all of the possible completions of a plan is *necessarily true*, and that a proposition which is satisfied in some possible completion of a plan is *possibly true*. Necessary and possible truth are concepts that will be required to describe TWEAK more fully later.

It should be noted that the addition of a constraint to an incomplete plan could conceivably make that plan inconsistent. As an example of inconsistency, consider constraining some variable X to bind (*codesignate*) with a constant A, while the variable X is already constrained to bind to a different constant B. In this case backtracking would be invoked and other plan completion paths not constraining X to bind to A would be pursued.

Since the number of plan completions is exponential in size, computation of the necessary or possible truth of a proposition in a plan by searching all possible plan completions is prohibitively expensive. However, TWEAK does utilize a polynomial time algorithm to compute possible and necessary "properties" of the incomplete plan. This algorithm forms the "heart" of TWEAK, and will be examined in some detail in a later section of this paper.

As in other planners, a complete plan in TWEAK is a total order on a finite set of operators. An operator, (sometimes referred to as step or action) in TWEAK consists of a set of preconditions which must be true just before an operator in order for that action to occur, and a set of postconditions which are guaranteed to

be true just after the action has occurred. In each case, set elements are expressed as propositions which are function-free atomic (i.e. $emp(X), \neg p(X), p(X,Y)$ etc).

Chapman does not provide a formal definition of a TWEAK plan. However, one is given in [Yang and Tenenberg, 1990], and this one is adopted with some slight modification for the purpose of this thesis.

Definition 2.3.1 *A plan* Π *is a triple* (A, B, NC)*, where*

- $A(\Pi)$ *is a set of operators, defined in terms of precondition and effect propositions,*
- $B(\Pi)$ *is a partial ordering on* A $(\prec)$*,*
- $NC(\Pi)$ *is a set of non-codesignation constraints of the form* $p \not\approx q$*, where* p *and* q *are both either terms or propositions in the operators of* A*.*

A simple example of a TWEAK plan is shown in Figure 2.2 [3]. In this example, the operator set A consists of { PICKUP(BlockA), STACK(BlockA, $some-block) }, B consists of a single ordering relation { (PICKUP $\prec$ STACK) }, and NC contains a single non-codesignation constraint, { (BlockA $\not\approx$ $some-block) }. Each operator in A possesses a set of precondition propositions, and a set of effect propositions, where each proposition or effect is a literal.

It should be noted that Chapman describes a plan as possessing both codesignation and noncodesignation constraints. For purposes of simplification, the above definition includes only explicit noncodesignation. Since removal of constraints is not allowed in the plan achievement of TWEAK, codesignation of a variable and a

[3] Note that "=/=" indicates non-codesignation.

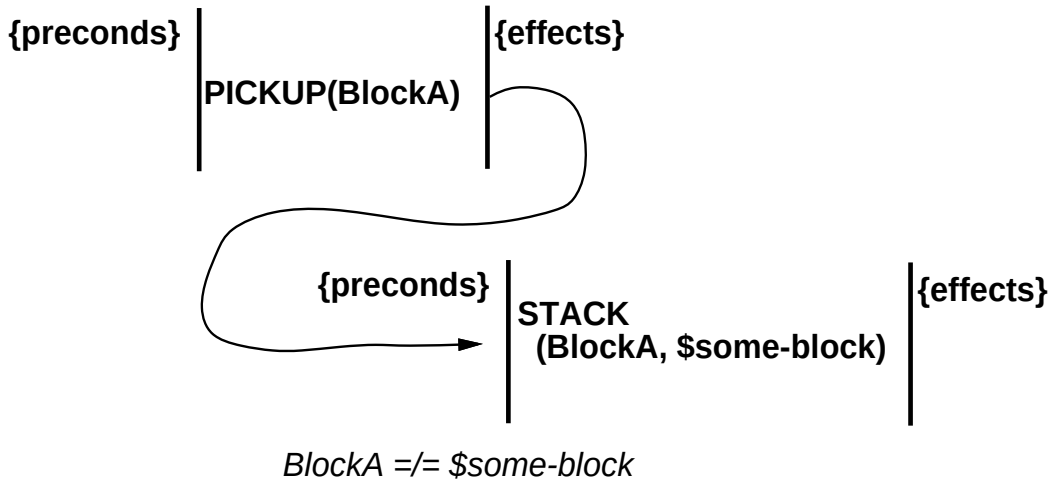


Figure 2.2: Simple example of a Tweak plan

term, or a variable and another variable is simply equated to the global replacement of all variable references in A and NC by the assigned variable or term identifier.

TWEAK's plans are deemed to be incomplete in two cases only. In the first case, a plan is incomplete while the time order of the plan's operators is not completely specified through *temporal* constraints which require that certain operators precede others. For instance, if a plan had three operators A, B, and C, and at some point in planning, A was constrained before B, and C before B then a partial order is imposed: $A \prec B$; $C \prec B$. In order for the time order of this incomplete plan to be completely specified, the relationship between A and C needs to be clarified. Additionally constraining A before C gives the total order $A \prec C \prec B$. In the second case, plans are incomplete when operators in the plan are not completely specified through *codesignation* or unification constraints which determine variable to constant "assignments" or "codesignations". All variables must be "bound" or "codesignated" to a specific constant in order for a plan to be complete.

Simply put, codesignation constraints can enforce either codesignation of elements (i.e. $X \approx Y$ meaning variable X is constrained to codesignate to variable Y), or non-codesignation (i.e. $X \not\approx Y$ meaning variable X is constrained not to codesignate with variable Y). Rules for codesignation parallel those of unification.

Yang and Tennenberg [Yang and Tenenberg, 1990] specifically represent *possible* ($\Diamond$), and *necessary* ($\Box$) codesignation and ordering precisely for some plan Π, where a and b are operators in Π, and p and q are propositions in Π. A simplified version of this representation allowing for the removal of codesignation constraints in Π is shown in Figure 2.3 (Note: "=" indicates identity, while "$\approx$" indicates codesignation).

$$
\begin{aligned}
\Box(a \prec b) &\iff B(\Pi) \vdash (a \prec b), \\
\Diamond(a \prec b) &\iff \neg\Box\neg(a \prec b) \iff B(\Pi) \nvdash (b \prec a), \\
\Box(p \approx q) &\iff A(\Pi) \vdash (p = q), \\
\Diamond(p \approx q) &\iff \neg\Box\neg(p \approx q) \iff NC(\Pi) \nvdash (p \not\approx q), \\
\Box(p \not\approx q) &\iff NC(\Pi) \vdash (p \not\approx q), \\
\Diamond(p \not\approx q) &\iff \neg\Box\neg(p \not\approx q) \iff A(\Pi) \nvdash (p = q).
\end{aligned}
$$

Figure 2.3: Necessary and possible plan properties.

States of the world in TWEAK are described through implicit "situations". A plan has an initial situation (before execution of any operators), and a final situation (after plan execution), and associated with each plan operator are input (*before*), and output (*after*) situations. Quite simply, a proposition p is true in a given situation U if it is asserted by some operator E as an effect, and if that operator is necessarily before the given situation such that no other effect between the effector

E and the situation U contradicts the effect p. Since any situation in TWEAK is arrived at as a result of operators that are only partially constrained in terms of ordering and variable codesignation, every situation has both necessary and possible proposition truth values. The situations just before each operator in the partially completed plan, and the situation just after each operator in the plan consist of propositions possibly or necessarily asserted or denied by operators constrained to occur either possibly or necessarily previous to the operator in question. A plan is correct only if all operators in the plan have all preconditions necessarily true in their own input situation.

All changes in the world of TWEAK must be mentioned explicitly as operator postconditions. Neither uncertainty of execution of an action, nor indirect or implied effects are allowed. Uncertain action execution implies that the effects of some operator are only true "usually", or in some manner expressed in terms of probabilities. An example might be in some domain where the stacking of blocks is sometimes buggy. A robot might attempt to stack some BlockA on another BlockB, with the effect On(BlockA, BlockB) holding usually, but with a 10 percent chance that the stacking will fail, and the result will be On(BlockA, Table) when BlockA falls off of BlockB. Implied effects occur when some operator has effects which hold only under certain conditions true at the time of the operator's execution in the plan. For instance, a domain which allowed for the movement of the bottom block in a stack of blocks of arbitrary height has implied effects. As the position of the base block changes, so too does the position of the blocks above the base.

TWEAK essentially solves problems which are composed of initial situation propositions, the plan starting point, and final situation propositions, which must be achieved in order for the plan to be complete. A *complete* plan solves a problem if the plan can be executed in the initial situation or world state of the problem,

and the final situation of the problem is a correct partial description of the world state after execution. TWEAK's aim is to produce a plan that necessarily solves the problem it was given.

2.3.2.1.1 Tweak and the Frame Problem Planning involves the successive addition of operators to an incomplete plan. As operators are added to a plan, the state of the world changes. In general terms, unless every proposition has its truth value reasserted in every state, it is impossible to state exactly what is true in any state. This problem is known as the Frame Problem. A simple solution to the Frame Problem, known as the STRIPS Assumption [Fikes and Nilsson, 1971] , is to assume that the state of the world in terms of truth value of propositions does not change unless subsequent operators either assert or deny these propositions. TWEAK utilizes the STRIPS Assumption in a nonlinear planning environment.

Chapman [Chapman, 1987] defines his *Modal Truth Criterion* (MTC), which implies a procedural method of determining the truth of a proposition in a nonlinear plan. The knowledge of the truth of propositions at certain points in a plan will help us direct the construction of a plan.

Yang and Tenenberg redefine Chapman's MTC, replacing the notion of loosely defined plan "situations" with the more concrete notion of plan operators. This MTC definition is given below, and is represented in Figure 2.4 [4].

> Proposition p is necessarily true just before operator U in plan Π if and only if two conditions hold:
>
> 1. there is an operator $E \in A(\Pi)$ and e is in the set of effects of E ($e \in E_{effects}$), such that $\Box(E \prec U)$ and $\Box(p \approx e)$, and

[4] Note that "==" denotes codesignation.

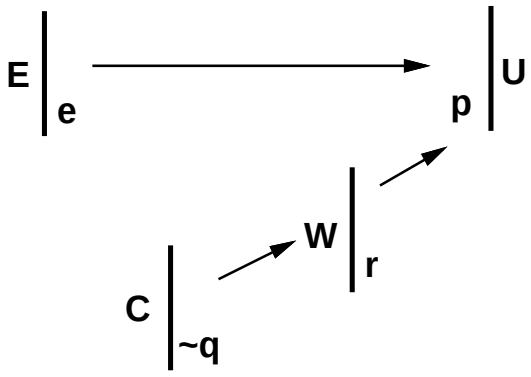


Figure 2.4: Modal Truth Criterion

2. for every operator $C \in A(\Pi)$ and $q \in C_{effects}$, if $\Diamond(C \prec U)$, and $\Diamond(\neg q \approx p)$, then there is an operator $W \in A(\Pi)$ and $r \in W_{effects}$ such that $\Box(C \prec W \prec U)$ and $NC(\Pi) \cup \{(\neg q \approx p)\} \vdash (r \approx p)$.

   Item 2 is illustrated in Figure 2.4. It basically states that whenever $\neg q$ codesignates with p, r is constrained to codesignate with p.

In a similar manner, possible truth can be determined by substituting *possible* for *necessary* and vice versa in the above description.

In the implementation section of this thesis, the application of MTC as a solution to the Frame Problem is described in detail, and the complexity of MTC in nonlinear planning is discussed.

In the next section I will outline the way in which TWEAK ensures that a plan achieve goals, using a procedural interpretation of MTC.

2.3.2.2 Goal Achievement in Tweak

The TWEAK search space consists of a set of incomplete plans. Each incomplete plan has a set of operators that have preconditions that are not necessarily satisfied. TWEAK expands this space in search of a correct plan that solves all of the preconditions of the goal state.

The basic control strategy of TWEAK is to repeatedly select a plan in the search space, select an unsatisfied operator-precondition from this plan, and then generate successors of this plan by modifying the incomplete plan in all ways possible such that the selected operator-precondition is necessarily achieved.

In treating the truth criterion (MTC) as a nondeterministic algorithm, a goal achievement procedure is created which can function as the driver for a planning system. MTC describes *all* of the ways in which a proposition can be made to hold or be necessarily true in some situation. The criterion defines explicitly the constraints that need to be added to the plan definition to make the goal true.

Chapman's algorithm is nondeterministic in that there are several choices to be made in the planning process which control the search aspect of this planning implementation. Selection of the next plan to expand is nondeterministic. Control over plan selection is controlled by a breadth-first search, or by some other complete search strategy. Heuristic plan selection can be based on some measure of the distance the plan is perceived as being from the goal plan. Selection of the unsatisfied operator-precondition within the plan is also non-deterministic. A simplistic, computationally inexpensive solution might just use the first precondition found, while a more involved solution might be to find all unsatisfied preconditions, and select one "intelligently" according to some heuristic. Selection of this precondition directly controls the number of successors a particular plan will have, and as a

result directly impacts the search performance. It will be shown in [Yang *et al.*, 1991] that the arbitrary selection of preconditions to solve for successors generation does not affect the completeness of the overall strategy.

Within the MTC algorithm, several *choice points* exist. Each choice point indicates a disjunction in the possible construction of the solution plan. The *choice* is simply a selection of which constraint "type" or "plan modification operator" to use in goal achievement. The planning search space consists of all possible completions within the scope of these choices, although at any point in time, only one of these completions is considered, Each completion represents a particular branching point in the search space where each of several plan successors represents the selection of one choice. A planning search strategy dictates which search path to pursue based on certain heuristic methods or indicators within the plans themselves.

If the constraints required by the selected "operator" are inconsistent with the existing constraints in the current incomplete plan (e.g. attempting to constrain some operator E before some operator S when operator S has already been constrained before E), then a "failure" would be signaled from the constraint maintenance module of TWEAK, and the control structure would backtrack, and select a different "plan modification operator" path.

A description of MTC as a plan modification algorithm would be useful in clearing up any confusion at this point. The algorithm on page 28 captures TWEAK's usage of MTC as a plan modification procedure. Notice that square bracketed titles indicate the specific plan modification operators, and also that "(choice pt n)" indicates a choice as described previously. Additionally, references in the algorithm to "Ops" indicate "Operators", and references of the form "Op C" indicate "Operator C".

It should be noted that the algorithm shown is strictly my interpretation of his

description, and is not included in [Chapman, 1987].

MTC Plan Modification Algorithm

[Select some unsatisfied goal proposition p in operator U]

1. Either (choose nondeterministically):

   A. [Simple Establishment]

   Find (*Choice point 1*) some Op E existing, which both:

   a) Is possibly before or equal to U

   b) Could assert a proposition u, which could possibly $\approx$ with p

   Constrain E to be necessarily before or equal to U

   Constrain u to necessarily $\approx$ with p

   B. [Operator Addition]

   Add (*Choice point 2*) some Op E, which both:

   a) Could be possibly before or equal to U

   b) Could assert a proposition u, which could possibly $\approx$ with p

   Constrain E to be necessarily before or equal to U

   Constrain u to necessarily $\approx$ with p

   [Insure no Clobbering occurs]

2. Loop through all Ops C:

3. Declobber either way possible (*Choice point 3*)

   A. [Promotion]

   Constrain U to be necessarily before C

   B. [Loop through all propositions q in Op C:]

   Either (*Choice point 4*):

   i) [Separation]

   Constrain q to $\not\approx$ with p

   ii) [White Knight]

   Either (*Choice point 5*):

a) Find (*Choice point 6*) some *EXISTING* Op W which satisfies:

b) Add (*Choice point 7*) some *NEW* Op W which satisfies:

(1) C possibly before W

(2) W possibly before U

(3) W asserts r such that if $p \approx q$, $p \approx$ with r

Constrain C before W

Constrain W before U

Constrain r in W such that if $p \approx q$, $p \approx r$

Chapman proves in the appendix of his paper [Chapman, 1987] that the MTC is correct in that the goals (and hence problems) solved by the goal achievement procedure are solved correctly. As an extension of his proof of MTC's correctness, Chapman also proves that TWEAK is complete. So, if TWEAK is given a problem and terminates execution by returning a solution, the produced plan solves the problem. Furthermore, if a solution exists, TWEAK is certain to eventually terminate by returning that solution.

I will mention each specific plan modification operation shown in MTC goal achievement procedure briefly, and I will outline a simple example of each. Chapman claims that the MTC approach includes all possible ways of accomplishing goals, except for allowing the removal of certain clobbering operators C. Since every operator or action in an incomplete plan has been added to accomplish some goal, removal of an operator would result in negative progress in every case. The MTC control strategy guarantees that any incomplete plan solving a subgoal by removing an operator would be reachable the MTC search space via some different *choice point* selection path. Further, it is never the case that the only way possible

to achieve some goal is remove a operator, if a goal is satisfiable in a particular domain, new operators can be added, or existing operators used to establish this goal.

PLAN MODIFICATION OPERATIONS

The MTC approach to goal satisfaction has two main steps. The first is goal establishment, and the second is establishment declobbering. Goal establishment determines all ways that a certain goal could be asserted in this particular incomplete plan. Declobbering determines all ways that any parts of the incomplete plan interfering with a goal establishment can be repaired.

A. ESTABLISHMENT Operations

In order to establish a goal in all ways possible, we can either use the effect of an existing operator in our incomplete plan, or we can add a new operator to our plan. The former option is known as simple establishment, the later as operator addition.

1. Simple Establishment

   Description: A precondition literal of an operator U is p, and an operator E exists in the current plan asserting a proposition q. Simple Establishment of p entails constraining q to codesignate with p, and constraining E to necessarily precede U.

   Example: Consider the example for three blocks A, B, and C shown in Figure 2.5, where two operators UNSTACK(C, \$X) and PICKUP(A) exist unordered w.r.t each other in some plan. An unsatisfied precondition of PICKUP(A) is clear(A). UNSTACK asserts clear(\$X) as an effect where \$X

is unbound, and PICKUP requires that clear(A) hold as a precondition. If we order UNSTACK $\prec$ PICKUP and constrain \$X $\approx A$, then the precondition clear(A) is established in PICKUP, by the existing establisher UNSTACK.

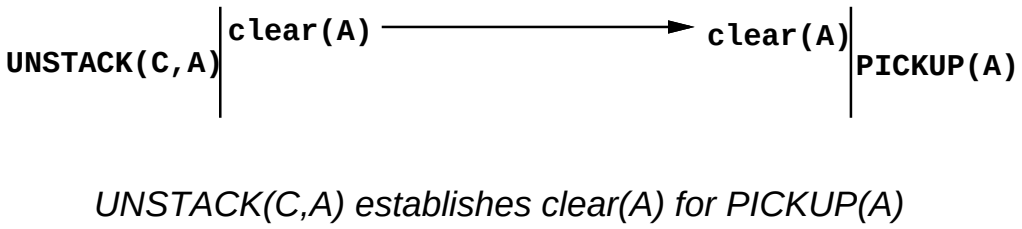


Figure 2.5: Simple Establishment

2. Operator Addition

   Description: A precondition literal of an operator U is p, and an operator E is added to the current plan such that E asserts a proposition q. Operator Addition establishment of p entails constraining q to codesignate with p, and constraining E to necessarily precede U.

   Example: Along the lines of the previous example, say a plan exists where UNSTACK is not in the plan at all. PICKUP(A) has an unsatisfied precondition of clear(A), but no operator exists in the plan that could establish Clear. If UNSTACK exists in the domain operator list, then a new operator UNSTACK(\$X, \$Y) could be added to the plan such that \$Y $\approx A$, and UNSTACK $\prec$ PICKUP. In this case, clear(A) is established for PICKUP by the new establisher UNSTACK.

   B. DECLOBBERING Operations

Even though the chosen goal has now been established, it is still possible that the goal does not necessarily hold in all completions of our incomplete plan. Clobbering operators may be possibly constrainable between our establishing operator E and our "user" operator U, and so may be capable of denying our goal. In order to insure that this does not happen, the plan must be declobbered by either moving the denying operator out of the way in the sense of operator ordering (promotion); constraining the denying operator effects to not be unifiable with the goal (separation); or by "repair" operator - a "white knight" can be inserted between the clobberer and the goal operator such that the white knight re-asserts the goal condition.

Consider this example case (Figure 2.6) for declobbering: UNSTACK(C,A) asserts clear(A), a precondition that PICKUP(A) requires. Also in this plan is an operator STACK(D,$X) unordered w.r.t UNSTACK or PICKUP. STACK asserts not clear($X) as an effect. Since STACK is possibly between the establisher (UNSTACK), and the user (PICKUP), and STACK asserts an effect ¬ clear($X) that possibly codesignates with clear(A), STACK is a clobberer of clear(A).

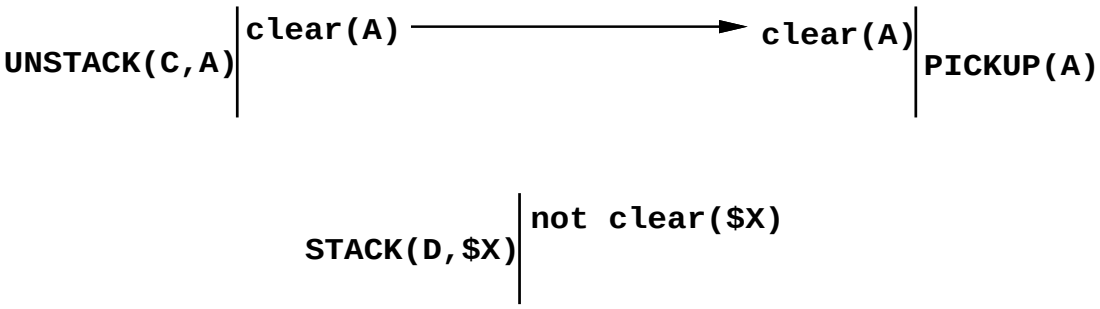


Figure 2.6: Goal clobbering

3. Promotion

   Description: For all operators C possibly before U where C denies a proposition q that possibly codesignates with a desired goal proposition p in U, "promote" operator C by constraining C to be strictly after U (U before C).

   Example (Figure 2.7): Since STACK is unordered w.r.t. PICKUP, we can simply constrain PICKUP to $\prec$ STACK. Now, even though STACK still asserts not clear($X), this assertion can no longer affect the truth of the precondition of PICKUP, clear(A).

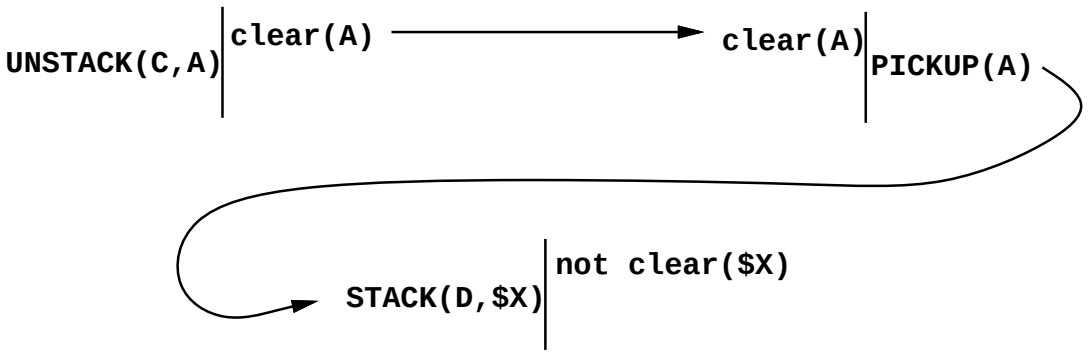


Figure 2.7: Promotion

4. Separation

   Description: For all operators C possibly before U where C denies any proposition q that possibly codesignates with a desired goal proposition p in U, "separate" p and q by constraining p to not codesignate with q.

   Example (Figure 2.8): Since STACK asserts not clear($X), where $X is unbound, we can constrain $X to $\not\approx A$, and now STACK no longer affects the truth of the precondition of PICKUP, clear(A).

5. White Knight - existing establisher

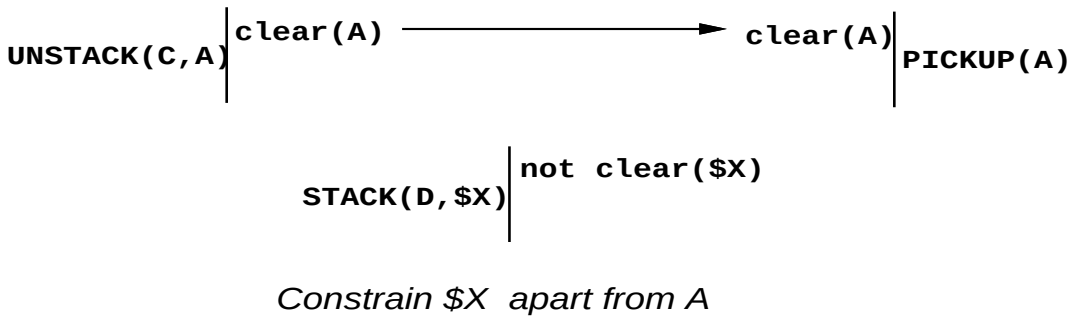


Figure 2.8: Separation

Description: A goal of an operator U is p, and an operator E (necessarily before U) exists in the current incomplete plan asserting p. Some operator C also exists such that C possibly lies between E and U, and some proposition q is denied in C, and possibly codesignates with p in U. Some operator W exists in the current plan such that W asserts a proposition r that also possibly codesignates with p. Performing "White Knight - existing establisher" entails constraining operator W to necessarily follow C and also to necessarily precede U. As well, the proposition r in W is constrained to codesignate with p if proposition q in C codesignates with p.

Example (Figure 2.9): STACK asserts not clear($X), $X unbound, and this possibly denies clear(A) for PICKUP. STACK only denies clear(A) when $X $\approx$ A. Say another operator UNSTACK2($X2,$Y2) exists in the plan such that UNSTACK2 asserts clear($Y2), and UNSTACK2 is unordered w.r.t. PICKUP or STACK. If we constrain STACK $\prec$ UNSTACK2, $Y2 $\approx$ $X whenever A $\approx$ $X, whenever STACK clobbers PICKUP, UNSTACK2 will rescue PICKUP by re-establishing clear(A).

6. White Knight - new establisher

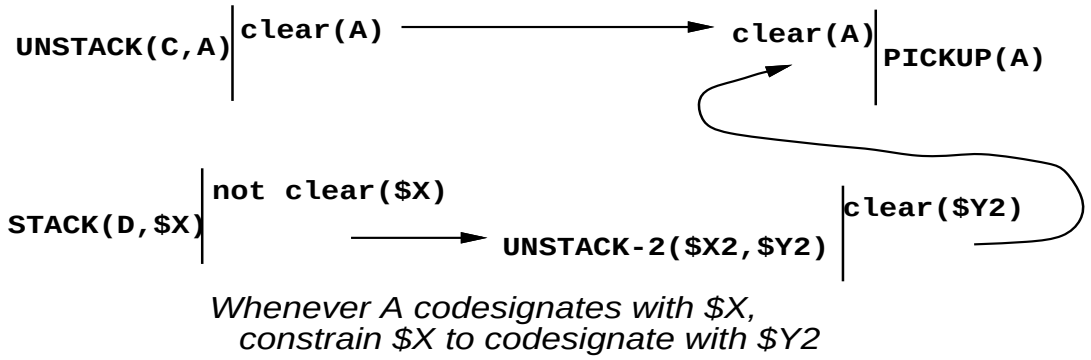


Figure 2.9: White Knight

Description: This operation is identical to the above operation except that the operator W (UNSTACK2 in the example) would be added to the plan from an action or operator template, rather than selected from existing operators in the plan.

It must be noted that operator addition clearly adds new preconditions to the incomplete plan, each of which must be achieved at some point. This additional operator could possibly deny previously achieved goals. These two factors suggest that the fewer the operators in a solution plan, the simpler it should be to create, and the less the amount of work will be required by MTC to determine the necessary or possible truth of preconditions in that plan. Quite simply, a nonlinear plan's complexity in terms of possible precondition "clobberings" grows quickly with respect to the number of operators in the plan. Chapman states that TWEAK "prefers" constraint posting to operator addition, although he is not specific about how this preference is accomplished. Our implementation achieves this preference through the use of a plan selection heuristic in which plans with fewer operators are preferred.

Chapman points out that since the MTC is sufficient as well as necessary,

these operations encompass ALL the operations needed to make an incomplete plan achieve a goal. While this result must be seen as a large accomplishment in view of earlier planners which made little or no attempt to fully describe or justify their operators, it is important to note that the concept of white knight operator insertion is unnecessary.

2.3.2.2.1 White Knight Operation Removal It can be shown that the White Knight declobbering operation can be dropped from the goal establishment procedure of MTC without loss of completeness. In fact, by dropping this operator, the branching factor is decreased at each goal establishment. When a new white knight operator is added following the addition of a new operator as establisher, the plan can be seen to grow by two operators in solving just one goal. In general, this is not necessary since only one operator is ultimately responsible for the establishment of any single goal.

Chapman observed in [Chapman, 1987] that White Knight did not seem to be useful in general since its effect on a plan can be considered to be either a "Separation" operation or "Establishment" operation. In fact, Chapman points out that he did not implement White Knight in the original version of TWEAK. A simple derivation sketched out by a number of researchers is shown below, demonstrating that White Knight is unnecessary in a complete TWEAK planner.

From the derivation it can be seen that the "White Knight" will either turn out to assert the goal in question directly, (and we might as well consider the output situation of the "White Knight" operator as the actual establisher), or it will not, and we might as well have used the modification operator "Separation" to defeat the clobberer. The White Knight operation outlined by Chapman is of the following form:

> We wish to achieve some goal proposition p in an arbitrary operator U in our incomplete plan. Some operator E necessarily before U has asserted p, however the problem which exists is insuring that no operator between E and U denies p. So, for each operator C possibly denying some proposition q which possibly codesignates with p (thus possibly denying p) between E and U, White Knight ensures that there exists another operator W, asserting some proposition r, necessarily between C and U, such that if C denies q and q codesignates with p, then r in W codesignates with p as well.

This *insurance* logically reduces to "W between C and U, and if C denies q codesignating with p in U, then r asserted by W codesignates with p in U." We can see this slightly more formally in the following derivation:

1. $(C \prec W \prec U) \wedge [\ (p \approx q) \Rightarrow (p \approx r)\ ]$
2. $(C \prec W \prec U) \wedge [\ \neg\ (p \approx q) \vee (p \approx r)\ ]$
3. $(C \prec W \prec U) \wedge [\ (p \not\approx q) \vee (p \approx r)\ ]$
4. $[(C \prec W \prec U) \wedge (p \not\approx q)] \vee [(C \prec W \prec U) \wedge (p \approx r)]$

The first part of the derived conjunction shown in point 4 can be read as:

> Constrain W after C and before U, and constrain p in U to not codesignate with q in C.

Clearly W is extraneous, and the result is "Separation" as defined by Chapman.

The second part of the disjunction can be read as:

> Constrain W after C and before U, and constrain p in U to codesignate with r in W.

Here W is acting as the "Establisher" as defined by Chapman, and E is extraneous.

In this manner, we have reduced Chapman's White Knight operation to be either a case of "Separation" or of "Establishment", and nothing else, and a simpler new model emerges, just as sufficient and complete as Chapman's MTC. This simpler model forms the "heart" of my TWEAK implementation.

It should be noted that the removal of White Knight in favor of Separation and Establishment results in a planner that is more "committed". A planning paradigm which allowed the White Knight operator necessarily includes constraints of the form:

> If $p \approx q$ becomes true, then constrain $r \approx p$.

A planner with White Knight removed essentially commits to limiting constraints (via Separation and Promotion) earlier than would be the case if White Knight was left in. One very large advantage of dropping White Knight is that a "White Knight" planner requires inferences to be made as codesignation constraints are added to a plan. The cost of performing these inferences [5] is not addressed by Chapman in [Chapman, 1987], however, it could conceivably be quite large, and could even dominate the cost of MTC if many such constraints existed.

[5] In fact, these inferences would be required with every constraint insertion in order to verify the consistency of the resulting plan

2.3.2.3 Major Tweak Limitations

The following points constitute a summary of some of the major limitations pointed out by Chapman [Chapman, 1987] for his TWEAK formulation. All of these limitations are carried forward into the implementation of TWEAK used to obtaining experimental results in this thesis.

1. Although TWEAK is complete, and will find a solution if one exists, it is not the case that TWEAK will always complete when attempting to find a solution to a problem that cannot be solved. In fact, TWEAK has three possible outcomes:

   Success by finding a solution plan.

   Failure when the search space has been exhausted; specifically all plans left in the search space have subgoals remaining that cannot be satisfied within the constraints of the domain.

   TWEAK never terminates; specifically the use of "Operator Addition" results in the addition of more subgoals into the plans in the search space, and no progress is made in reducing the number of the unsatisfied subgoals.

2. There are two major restrictions on TWEAK's action representation:

   Situation dependent actions are not allowed

   All actions in TWEAK must have effects that are independent of the situation in which they are applied, and all action effects must be explicitly represented in the postconditions of the action. In the first case, TWEAK cannot allow for situations such as (example due to Chapman) in the blocks world where only zero, one, or two blocks may be on a given block. A *space(block)* function tells how much room is left on a block. A precondition of using a *Puton(onblock,*

baseblock) action would then be that *space(baseblock)* be non-zero. Hence a situation is achieved in which the representation has postconditions (results) that are functionally dependent on the input situation. Chapman demonstrates that if TWEAK were extended in its representation just enough to allow for this, the Modal Truth Criterion would fail as a result of two operators acting together to deny some proposition p in some operator U. Neither of the two individual operators (say *Puton* where *space* is decremented by 1 as a result) would deny p, and so MTC would not require that they not precede U. Clearly if not constrained, it is possible that both operators are before U, and therefore p (say space non-zero) is denied.

Implicit Side Effects are not allowed

The second major restriction on TWEAK is that actions cannot have side effects not explicitly specified in the postconditions. This can be illustrated in a blocks world example due to Chapman [Chapman, 1987]. Consider the case where block a is on block b, and block b is moved somewhere. Explicitly, block b is moved, but the sideeffect is that block a is moved also. If TWEAK is allowed to include deduction within situations, this side effect action can be represented. Once again, with this addition, MTC fails through two (or more) operators working together to deny some proposition where none of the offending operators themselves does the dirty work.

The major contribution of this demonstration of MTC's failure in cases of extension of TWEAK's action representation is in Chapman's explanation of the "costs" that are necessarily incurred by modifying the MTC. Chapman [Chapman, 1987] points out that the extension of action representation over that described in TWEAK runs into difficulties with the Frame Problem, which is equated in TWEAK to finding an efficiently implementable truth criterion. Chapman proved that determining

necessary truth in the case where action representations are extended to represent conditional actions, dependency of effects on input situations, or derived side effects is NP-hard

Planning is still an area of interest in AI (despite it's nature in general), and Chapman suggests that we have three options for living with the NP-hard nature of planning. We can either:

1. Invoke the "Ostrich Principle"[6] by assuming that plans will be produced in the particular domain efficiently enough to meet our planning needs.

2. Relax our correctness requirement thus producing a planner that sometimes doesn't quite work.

3. Relax our generality requirement and head off into domain dependent planning.

This thesis is concerned with the improvement of nonlinear planning without losing either correctness or generality, and so one could argue we have accepted the Ostrich Principle as our option. Nevertheless, what is demonstrated in this thesis is that methods exist that can improve the performance of planning in a general, useful, domain independent fashion.

[6]Chapman claims this term is due to Yoav Shoham

2.4 Abstraction in Planning

2.4.1 Overview

Many AI researchers have suggested that in attempting to solve problems such as planning or plan recognition we should consider a problem to consist of several levels or layers of a hierarchy. It is a popular idea that difficult problems can be made more manageable if we break them down into certain kinds of higher level sub-problems first, solve these, and then progressively descend the layers of the hierarchy solving our problem at each layer until a solution is found that satisfies the problem at the base layer of the abstraction hierarchy.

Abstraction based approaches similar to this have appeared in a wide range of AI fields: ISA-hierarchies in knowledge representation, generalizing ISA-hierarchies to actions over abstract types in linear planning systems [Nau, 1987], plan recognition [Kautz, 1987, Carberry, 1990], and theorem proving [Plaisted, 1981], represent but a few.

Sacerdoti [Sacerdoti, 1974] utilizes an abstraction strategy known as precondition-elimination which is formalized by Tenenberg in [Tenenberg, 1988]. This strategy basically eliminates a subset of predicates in the domain as one ascends the abstraction hierarchy. The predicates are partitioned, with each partition assigned a particular integer criticality. At higher levels of abstraction, a new planning problem is derived from the lower level by elimination of preconditions having criticality less than the current level of abstraction. Consider the Towers of Hanoi domain for 2 rings (1 large and 1 small), and 3 pegs. It is only possible to move rings on to empty pegs or on to larger rings. The operator set consists of MOVE-SMALL-RING which can move the small ring between any start and any destination peg,

and MOVE-BIG-RING which can move the large ring between any two pegs. Since we have said that it cannot be the case that the large ring can be on top of the small ring, MOVE-SMALL-RING has a single precondition on-small(initial-peg). MOVE-BIG-RING has three preconditions: on-big(initial-peg), ¬on-small(initial-peg), and ¬on-small(destination-peg). We can imagine an abstraction hierarchy of 2 levels where on-big preconditions have a criticality of 1, and on-small preconditions have a criticality of 0. If we are planning at the most abstract level (criticality 1), we ignore all on-small preconditions in the operator precondition lists, and solve all subgoals with the "reduced" operators. Once solved, at level 1, the problem is made less abstract by adding the level 0 preconditions back into the plan and operator set, and planning continues at the next more concrete level.

AbTweak extends the precondition-elimination strategy of abstraction seen in the linear planner Abstrips [Sacerdoti, 1974] to the nonlinear, least commitment planning paradigm outlined by Chapman for Tweak [Chapman, 1987]. In AbTweak, an attempt is made to benefit from both Tweak's ability to solve problems in a least-committed fashion, and from ABSTRIP's hierarchical approach to limiting search.

2.4.2 A Hierarchical Planning Example

As an example of how abstraction can be used as an approach to planning, consider a simplified two disk Tower of Hanoi domain. The operator set and criticality assignments for this domain are show in Figure 2.10 and Table 2.1.

Consider the initial situation as { *onbig(Peg1), onsmall(Peg1)*} , and the goal situation as { *onbig(Peg3), onsmall(Peg3)* } . At the highest level of abstraction, level 1 in this example, the abstract goal set consists of { *onbig(Peg3)* } . The

MOVEBIG (x, y)

Preconditions $\{\neg onsmall(x), \neg onsmall(y), onbig(x)\}$

Effects $\{\neg onbig(x), onbig(y)\}$

MOVESMALL (x, y)

Preconditions $\{onsmall(x)\}$

Effects $\{\neg onsmall(x), onsmall(y)\}$

Figure 2.10: Simple Hanoi Domain Operators

Predicate	Criticality
onbig	Level 1
onsmall	Level 0

Table 2.1: Simple Hanoi Domain Criticality Assignments

abstract initial set of conditions consists of {*onbig(Peg1)* } . In addition, at this abstract level, the operator set can be viewed as only involving preconditions and effects at this level, specifically *onbig*. The initial plan would be formed as shown in Figure 2.11.

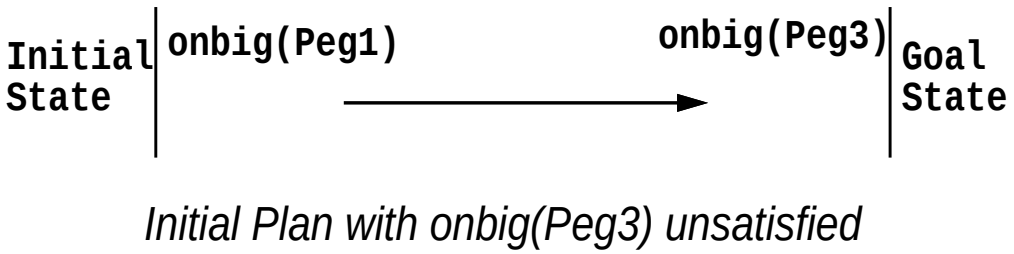


Figure 2.11: Initial abstract plan in Simple Hanoi

Only a single goal or precondition is unsatisfied at this abstract level, *onbig(Peg3)*. This precondition can only be established by assertion of a new establishing operator, *MOVEBIG($X, Peg3)*. Insertion of this operator into the current plan as an establisher for *onbig(Peg3)* would result in a partial plan as shown in Figure 2.12 on page 46.

Again, this plan has only a single unsatisfied precondition, *onbig($X)* in the *MOVEBIG* operator. Two possible establishments exists for *onbig($X)*, either the initial state operator can be constrained such that the initial condition *onbig(Peg1)* is constrained to codesignate with *onbig($X)*, or a new operator *MOVEBIG($X1, $X)* is added to the plan. The former establishment results in a plan with no unsatisfied preconditions at this level of abstraction, and thus the first level 1 solution plan is found (Figure 2.13, page 46).

We now decrease the level of abstraction by one level for the level 1 solution and

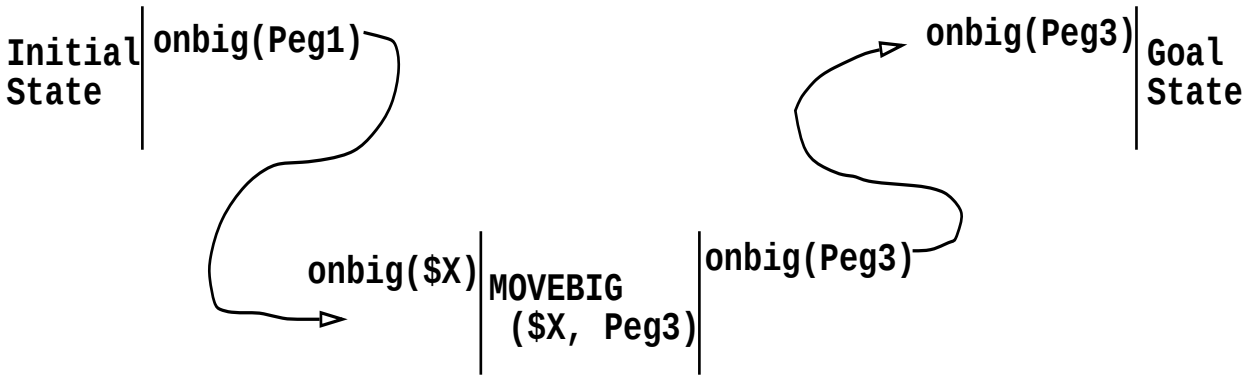


Figure 2.12: Step 1 in abstract planning example

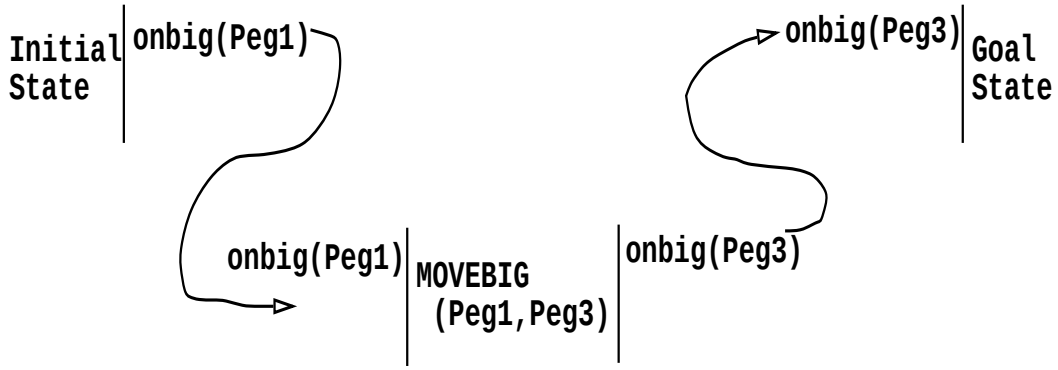


Figure 2.13: Step 2 in abstract planning example

operator set, and arrive at a level 0 plan that is shown in Figure 2.14 on page 47.

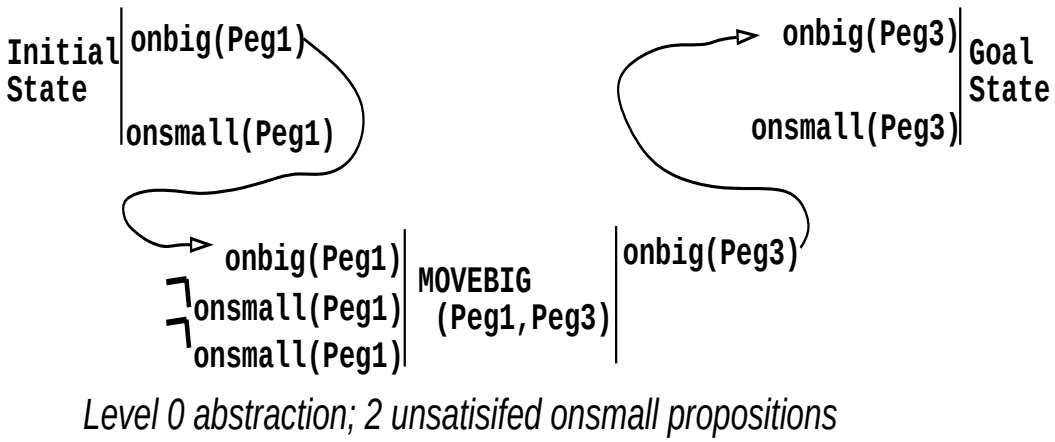


Figure 2.14: Step 3 in abstract planning example

This level 0 plan has two unsatisfied preconditions, *onsmall(Peg3)* in the goal state operator, and ¬ *onsmall(Peg1)* in the *MOVEBIG(Peg1, Peg3)* operator. Either one could possibly be selected. For this example, say that *onsmall(Peg3)* was chosen. Only one possible establishment exists for this precondition, the addition of a *MOVESMALL-1($X2, Peg3)* operator to the plan. The resultant plan is shown in Figure 2.15, page 48.

This resultant plan has several preconditions unsatisfied: *onsmall($X2)* in *MOVESMALL-1* and preconditions ¬ *onsmall(Peg3) and* ¬ *onsmall(Peg1)* in *MOVEBIG*. If we choose *onsmall($X2)* in *MOVESMALL-1*, the only establishment could be the insertion of a new MOVESMALL operator, *Move-Small-2($X3, $X2)*, shown in Figure 2.16 on page 48.

Certain orderings of this partially ordered plan cause preconditions to be denied.

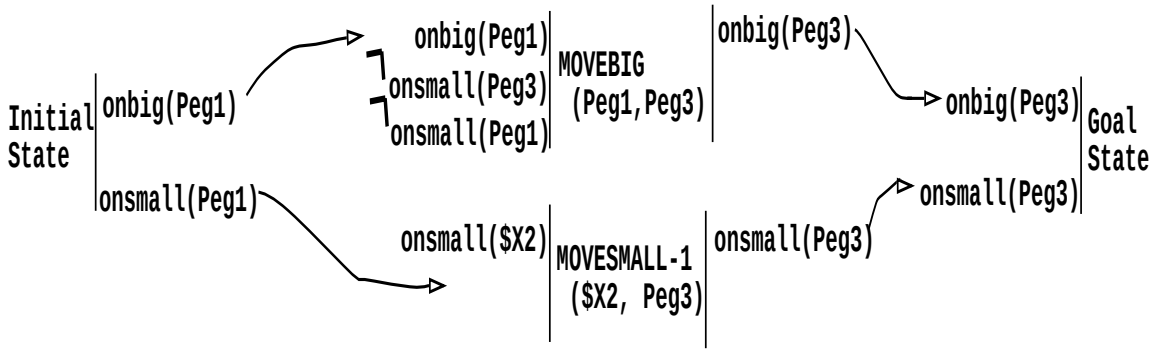


Figure 2.15: Step 4 in abstract planning example

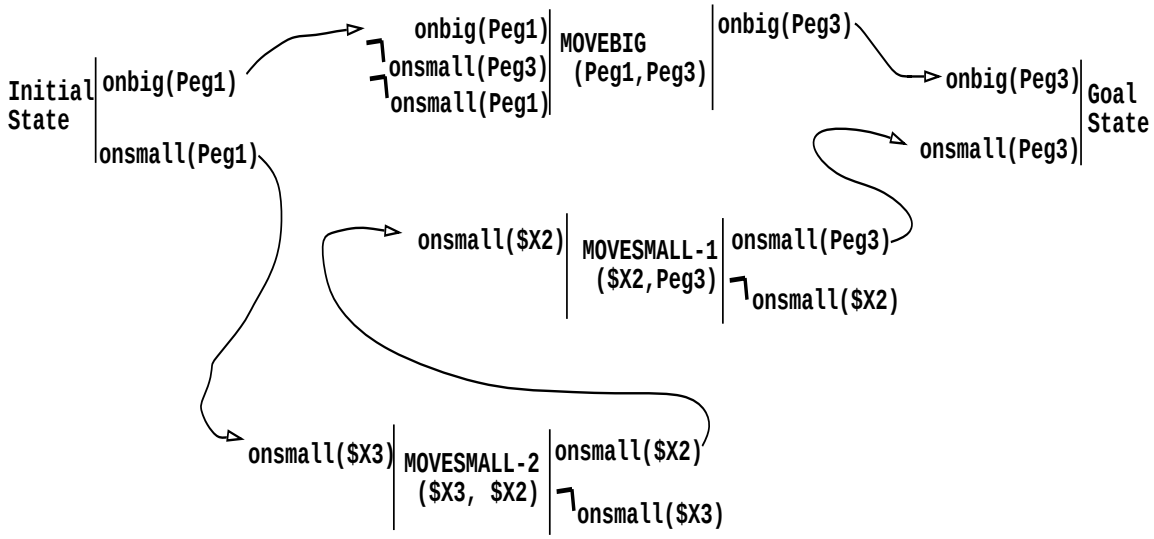


Figure 2.16: Step 5 in abstract planning example

In two more steps, establishing *onsmall($X3)* in *MOVESMALL-2* by the existing establishment of the initial operator causes *$X2* to be codesignated with Peg1. Establishing ¬ *onsmall(Peg3)* in *MOVEBIG* with the existing establishment of "*Move-Small-2*, ¬ *onsmall($X3)*" results in the codesignation of *$X3* and *Peg3*, and the promotion of *MOVESMALL-1* over *MOVEBIG*. The final solution plan at the concrete level is shown in Figure 2.17 on page 49.

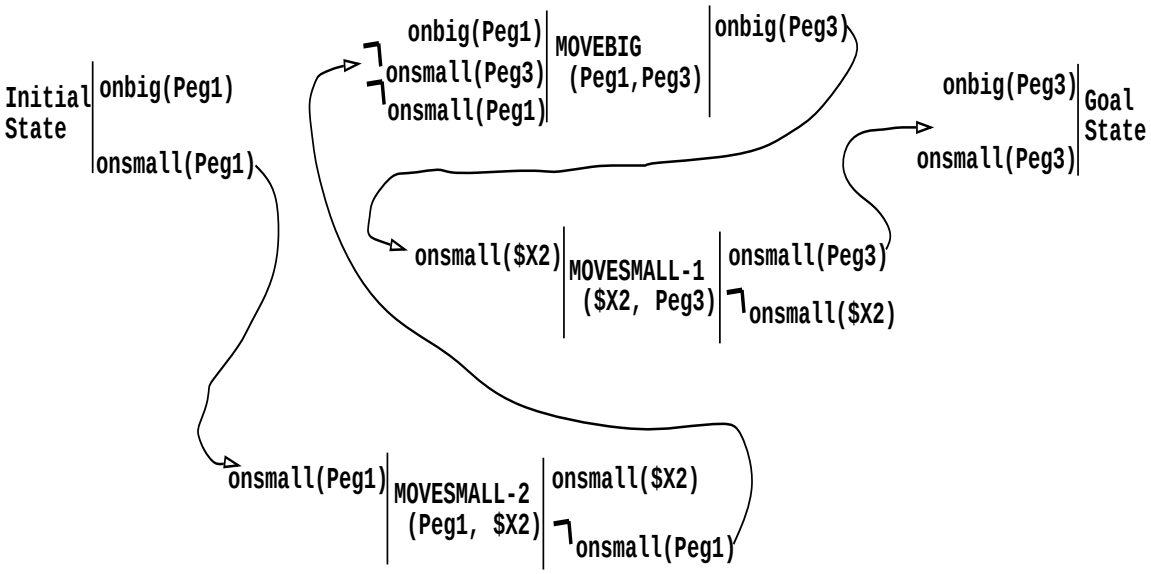


Figure 2.17: Step 6,7 Concrete level solution plan

It should be noted that one variable ($X2) remains unbound in this solution plan. In the simple domain described in this example, there is no specific limitation on the location to which the rings may be moved. MOVE operators only have preconditions that insist that a ring must be to a "destination" that has no smaller rings already on it. Essentially, the concept of pegs as individual entities is poorly captured in the domain definition. If it was desired to limit the number of possible

Peg locations available for moving, it would be necessary to limit the destination locations in some manner. One possible approach is to explicitly identify the subset of all possible locations that are pegs via a "typing" constraint. Addition of a constraint of the form "is-peg(Peg1)" for each of the three pegs, and addition of a conditions for MOVE operators that objects can only be MOVED between locations that are characterized by "is-peg" will remove this ambiguity. In the example of Figure 2.17, the location variable is unbound as a result of the fact that no typing is made of the pegs on which rings can rest. An assumption is captured in the domain as it exists that an infinite number of pegs exist, while in a correct implementation, it must be specified that only a finite number of pegs are available, namely Peg1, Peg2, and Peg3. In this fashion, the variable $X2 would have an associated precondition for each operator that required $X2 to be of some type "Peg", and no correct solution plan would contain unbound variables.

2.4.3 AbTweak

2.4.3.1 Background

The precondition-elimination approach to abstraction has been presented with linear planners, as in ABSTRIPS [Sacerdoti, 1974]. ABTWEAK [Yang *et al.*, 1991, Yang and Tenenberg, 1990] possesses the capability to plan in a nonlinear, least committed fashion, and thus is capable of returning partially constrained solution plans such that any fully constrained version of the partially constrained solution would also solve the problem. In this manner, ABTWEAK is a more general planner than ABSTRIPS.

2.4.3.2 AbTweak Design

AbTweak is based largely upon the previously described Tweak nonlinear planner. The addition of a dimension of control for the successive levels of precondition-elimination results in the abstract, nonlinear planner AbTweak.

It is shown in [Yang and Tenenberg, 1990] that not every abstract solution plan in AbTweak is reducible to a less abstract solution plan, even though a less abstract solution plan may in fact exist. This factor must be taken into account when designing an abstract search strategy. Search at any particular level of abstraction is accomplished by allowing Tweak to plan based only upon preconditions at the current or higher levels of abstraction.

2.4.3.3 AbTweak Plan Representation

An operator in Tweak is defined by a set of precondition literals and effect literals. A plan, Π, is defined as a triple (A, B, NC), where A is a set of operators, B is a partial order on A, and NC is a set of noncodesignation constraints.

As in Tweak, a complete plan in AbTweak is a total order on a finite set of operators or actions. An operator in AbTweak consists of a set of preconditions which must be true in order for an operator to occur, and a set of postconditions which are guaranteed to be true just after the operator has occurred. In each case, set elements are expressed as propositions which are function-free literals (i.e. $p(X)$, $\neg\ p(X)$, $p(X, Y)$, etc).

An incomplete plan in Tweak may be made complete by the addition of operators, ordering constraints, and noncodesignation constraints to A, B, and NC respectively. In addition to these constraints, a complete plan in AbTweak must also be at the lowest, or concrete abstraction level. Since AbTweak plans can be

at various levels of abstraction, a slightly more complex plan description is necessary than is the case for TWEAK. Yang and Tenenberg [Yang and Tenenberg, 1990] define ABTWEAK in the following way:

A k level ABTWEAK system is a triple $\Sigma = (L, O, crit)$, where

(1) L is a TWEAK language;

(2) O is an operator set, as in TWEAK, and

(3) *crit* is a function:

$$\bigcup_{o \in O} P_o \rightarrow \{0, 1, \ldots, k-1\}.$$

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

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

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

and ${}_i\alpha$ is operator α 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:

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

2.4.3.4 Goal Achievement in ABTWEAK

The ABTWEAK search space consists of a set of incomplete plans, each of which has a particular level of abstraction between the maximum and minimum criticality value assigned for preconditions in a given domain. Each incomplete ABTWEAK plan is simply a TWEAK plan with a subset of preconditions removed to reflect

the abstraction level. Some of these preconditions are not necessarily satisfied. TWEAK searches for a solution plan by successively modifying incomplete plans in the space, and selecting unsatisfied preconditions to establish and declobber in each chosen plan. A plan that solves the given problem at some level of abstraction k is expanded to generate a level *k-1* successors plan, and in this fashion the entire search space is expanded in search of a correct plan at the most concrete level.

The basic control strategy of ABTWEAK is to repeatedly select a plan in the search space at some level of abstraction, select an unsatisfied operator-precondition from this plan, and then generate successors of this plan by modifying the incomplete plan in all ways possible so that the selected operator-precondition is necessarily achieved.

Once again, we treat the truth criterion of TWEAK as a nondeterministic algorithm, and obtain a goal achievement procedure which functions as the driver for a planning system. The abstract MTC (or AB-MTC) differs from MTC only in that when the criterion is applied to some plan at level k of abstraction, only preconditions and effects at level k are considered in the determination of necessary truth. Plan successors are generated in the following manner:

1. An *incorrect* plan at some level k, has successors as indicated by MTC, considering only preconditions of criticality of level k or higher.

2. A *correct* plan at level k, $(k > 0)$, has only a single successor, the level *k-1* version of this plan.

3. A *correct* plan at the concrete level where $k = 0$, has no successors, and is a solution plan.

2.4.3.5 Search Control in AbTweak

In this abstract search space, the selection of plans is somewhat more complicated that is the case for the single-level planning of TWEAK. Plans in the abstract space have varying levels of abstraction, and expansions occur in a manner which interleaves selection over different abstraction levels. While plan selection was already a concern for TWEAK, ABTWEAK has an additional difficulty of chosing the correct abstract plan under which to search for the next lower level solution. ABTWEAK requires search strategies in two dimensions then, one to control the current abstract solution under which to plan, and one to control the search expansion within the selected level of abstraction.

Intuitively, less abstract plans are "closer" to the concrete level solution plan in terms of overall abstract "depth" in the search space, however, it is not guaranteed that any abstract level k solution plan will lead to a lower, concrete solution. Domains in which multiple level k solutions exist tend to be common. For example, if one wished to move the big ring in Towers of Hanoi from Peg1 to Peg3, you could simply move it directly from Peg1 to Peg3 in one move. However, a more complex solution might be to move it from Peg1 to Peg2, and then move the ring from Peg2 to Peg3. Both plans are valid solutions.

2.4.3.6 Tweak Limitations affecting AbTweak

ABTWEAK retains all of the restrictions on planning that TWEAK experiences. Specifically:

1. Since ABTWEAK is based upon the control strategy of TWEAK, ABTWEAK has the same completion conditions as TWEAK: successful completion by

finding a solution plan, failure by exhausting the search space, or failure to terminate indicating progress towards a goal is never made in the planning process.

2. AbTweak's plan representation, based upon that of Tweak, must:

   Not allow situation dependent actions

   Not allow implicit side effects for operators

2.4.4 Goal Protection in Planning

In goal-driven planning such as Tweak, Abstrips, or AbTweak, operators are added to the partial plan for a particular reason. These operators have been selected since they satisfy a particular plan goal. The addition of later operators for later goals can have destructive side effects that essentially undo the work accomplished by earlier operators. It seems likely then, that if we build plans more carefully we can avoid this inadvertent destruction of established goals.

We can protect goals in a nonlinear, non-abstracting planning system by identifying *important* establishments in some fashion perhaps, or even by protecting all establishments made previously. Further, a non-abstracting planning strategy which protected all previous goal establishments can be show to be complete. Essentially, Tweak establishes a selected goal proposition in every way possible at a given branching point in the plan search space. So, the establishment of some goal p for some operator U by a set of establishers $\{ E_1, E_2, \ldots E_n \}$ results in n successors. If a rule existed in the planning strategy which insisted that each of E_i remain establishing p for O in that particular successor plan, then a property known as the *Single Producer Property* (SPP) in enforced. Say one of the establishers E_k, had it's establishment clobbered by some other operator C which asserted p, and was

constrained between E_k and U. The question is: does the pruning of this successor violating SPP make the planner incapable of reaching a solution? The answer is no. The new establishment relation for p in the pruned plan was C establishing p for U. Since C can assert p, we know that C must be one of the operators E_i, call it E_c. So, if the only solution plan possible possessed the establishment of p by C, then this solution plan is reachable as a descendant of the successor utilizing E_c to establish p.

Goal protection in an abstract planner can be accomplished as in a linear planner within a particular level of planning, or can be done across levels of abstraction by protecting certain work done at a high level while refining a plan at less abstract levels. As opposed to goal protection in a non-abstract planning system, goal protection in an abstract planner must be done carefully if one wishes to retain completeness for a particular search strategy. The example in Figure 2.18 on page 57 shows how, in an abstract planning system, the protection of two parallel establishing operators at level K prevents a level k-1 solution from being found.

In any goal directed planning, a complete goal protection strategy will have the result of eliminating certain branches of the search space, and will thus reduce the branching factor of the search. While this does not necessarily imply that a constrained search strategy will outperform an unconstrained one, it is true that the branches removed from the space need not be traversed in order to find a solution.

Part of the purpose of this thesis is to evaluate the utility of various levels of goal protection across abstraction levels. In later sections of this thesis, I will outline properties of abstract search spaces which allow us to protect abstract search work, and still keep our strategy complete.

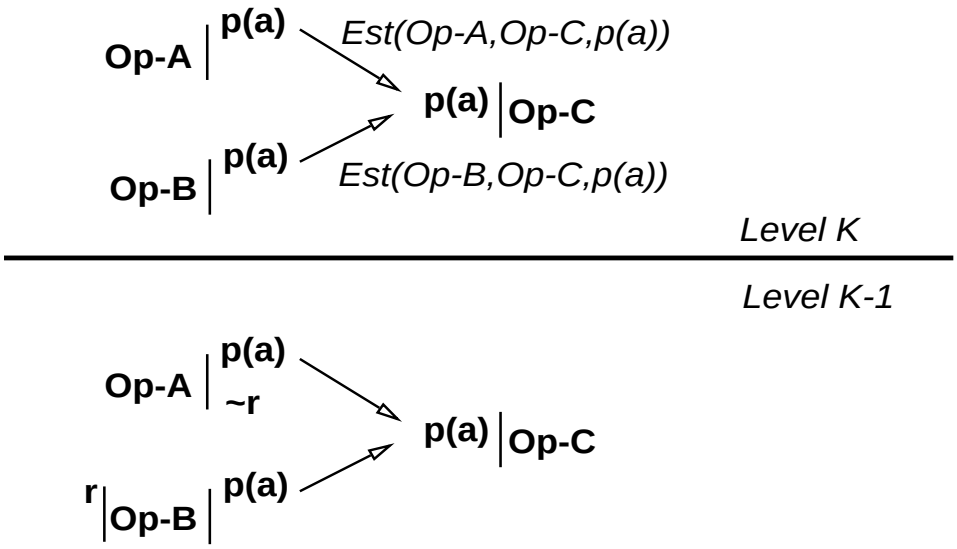


Figure 2.18: Abstract goal protection incompleteness

Chapter 3

Planner Implementation

3.1 Overview

TWEAK and ABTWEAK have been implemented in Common Lisp, and all experiments have been run on a Sun 4 [1]. The implementation has been done with strong emphasis placed on keeping various control strategies, plan modification, and plan representation consistent between TWEAK and ABTWEAK, and domain-independent. In my planner implementation, ABTWEAK is built directly upon a working version of TWEAK. Individual level search control in ABTWEAK is performed by TWEAK functions, and the truth criterion ABTWEAK utilizes in determining plan correctness is exactly a call to the TWEAK truth criterion while ignoring certain abstracted portions of the abstract plan. Similarly, the modification of plans within each level of abstraction in ABTWEAK is performed entirely by TWEAK.

Basically, there are three levels to my implementation:

[1] The implementation was done jointly with Qiang Yang.

1. Search within each level

   In order to compare and contrast abstract search and non-abstract search, we must strive to keep the two implementations consistent in their control strategies. Of particular importance is the control of search in non-abstract search, and the corresponding search control within each abstract level in abstract search. Both TWEAK and ABTWEAK utilize search strategies based upon the complete search strategy A^* [Nilsson, 1980]. The space searched for both TWEAK and ABTWEAK is not reduced by checking for plan redundancy as this is still a difficult open problem in nonlinear planning.

   Successor generation in the search space is controlled by the selection of unsatisfied precondition goals in selected plans. This is an important search factor both within each abstract search level, and in single level TWEAK search. I attempt to insure that neither the abstract nor the non-abstract selection gives a particular strategy an unwarranted advantage. The implementation of goal selection procedures and heuristics are exactly the same for the individual level search in ABTWEAK, and for TWEAK search.

2. Search across abstract levels

   Search between levels in ABTWEAK is essentially a selection process for which a correct level k plan in the space to choose next for expansion at the next more concrete k-1 level. In TWEAK, there is only one level of abstraction, the concrete, to consider, but in ABTWEAK the successors of many different levels are all potential plans to consider for expansion. We may wish to formulate our global search strategy to prefer certain types of abstract solution plans at each level of abstraction. For example at a high abstract level, we may prefer "simpler" plans in some sense, perhaps in terms of operator set size,

while at lower levels of abstraction this preference may not make sense. The challenge is to fit the strategy and the abstract plan preference to both the nature of abstract planning and to the domain in question. Since this thesis is concerned with domain-independent strategies, I describe an abstract level search strategy that tends to take advantage of the nature of the abstract search space in general. This approach will be discussed in a later section.

3. Plan representation

   Plan representation is consistent from TWEAK to ABTWEAK, with the additional complexity for ABTWEAK plans of carrying an indicator of level of abstraction. A plan in ABTWEAK is determined to solve the problem completely only when its level of abstraction is concrete, any other plan which has unsatisfied preconditions must be made less abstract by one level, and further expanded in the search space accordingly.

   Additional information is carried in the plan representation according to the requirements of certain heuristic approaches. For example, certain pruning heuristics require a plan's higher level causal relations in order to determine whether certain successors can be discarded. While this information could be determined from the plan, it is computationally much cheaper to simply carry this information along with the plan as it is built up. Similarly, certain other goal selection heuristics require the goal achievement hierarchy of a particular plan to be known, and in these cases this information is recorded and used in successor generation.

3.2 Truth Criterion

Chapman's MTC has been implemented as a method of determining the necessary or possible truth of propositions in nonlinear plans. The MTC has a complexity related to the product of the number of operators in the plan, the number of preconditions each operator has, the number of effects each operator has in the plan, and the degree of nonlinearity of the plan.

Each of the i operators U_i in a plan Π is considered in turn, and within each U_i, each of j preconditions p_j. For each p_j, all establishing operators E_j are found such that $E_j \prec U_i$, and has at least one effect p_j. Each establishment of some p_j in an operator U_i by some operator E_J can thus be represented as a relation of the form $Est(E_j, U_i, p_j)$. Every Est relation in Π is examined, and if there is at least one Est relation for every p_j in Π such that no operator C necessarily between E_j and U_i possibly denies p_j, then we can say that p_j necessarily holds. A plan Π is only correct if all p_j in all U_i in Π necessarily hold.

The complexity of determining unsatisfied plan preconditions directly affects the performance of a planning system. In the next section, we will the cost of this determination in plans exhibiting various degrees of nonlinearity. The worst-case complexity of fully determining the unsatisfied preconditions in a nonlinear plan Π is $O(n^4)$, where n indicates the number of operators U in Π. Factors detailing the average and best case complexity are discussed in the next section as well.

3.2.1 Plan Nonlinearity and the Truth Criterion

Intuitively, it would seem that the complexity of MTC varies directly as a function of the degree of nonlinearity of a particular plan. If we look at a spectrum of possible

plans, from strictly linear to completely parallel, we can see to what degree this complexity changes. Clearly, the more operators possibly between the operator with a precondition to establish and the actual establishing operator, the greater the number of comparisons that must be made to determine if a proposition is necessarily true. However, does this difference affect the overall complexity of the MTC algorithm? The following descriptions of MTC show worst case time complexity in plans with special cases of ordering construction.

For the purposes of the following complexity analysis, the following assumptions are made:

1. There are n operators in each of the plans considered.

2. Each of the n operators has l effects and l preconditions.

3. An operator A_e *establishes* proposition p for another operator A_i iff:

   A_e necessarily precedes A_i

   A_e necessarily asserts p as an effect

   For all A_j such that necessarily $A_e \prec A_j \prec A_i$, A_j does not necessarily assert either p or $\neg p$.

4. An operator A_e *negates* proposition p for another operator A_i iff:

   A_e necessarily precedes A_i

   A_e necessarily asserts $\neg p$ as an effect

   For all A_j such that necessarily $A_e \prec A_j \prec A_i$, A_j does not necessarily assert either p or $\neg p$.

5. The MTC used for analysis is the simplified version, with White Knight removed.

Chapman's MTC described in Chapter 2 and shown as a plan modification algorithm on page 28 is known to require time on the order of $O(n^4)$ to determine the truth of all preconditions in a plan. In Figure 3.1 the modification algorithm is presented so as to make the complexity with respect to the n clear.

For each of n plan operators A_i,

For each of l operator preconditions p,

a) Find each of n possible ESTABLISHERS A_e of p,

call the set of establishers Est-List

b) For each establisher A_e in Est-List

For each of n possible CLOBBERERs A_c of A_e,

Either:

Promote A_c over A_e, or

For all l effects (q) of A_c

Either:

Separate q and p, or

For all n possible White Knights,

Make a White Knight.

Figure 3.1: Complexity of Chapman's MTC

With respect to the number of operators in a plan, Chapman's algorithm can be seen to require time:

$n \times (n + [n \times n \times n])$,

This complexity reduces to $O(n^4)$.

The simplified algorithm used in the implementation of ABTWEAK this thesis does not use White Knight, and as a result enjoys a saving in terms of time complexity. This simpler algorithm can be seen in Figure 3.2.

For each of n plan operators A_i,

For each of l operator preconditions p,

Find each of n possible ESTABLISHERS A_e of p,

a) Find each of n possible ESTABLISHERS A_e of p,

call the set of establishers Est-list

b) For each establisher A_e in Est-List,

For each of n possible CLOBBERERs A_c of A_e,

Either:

Promote A_c over A_e, or

For all l effects (q) of A_c

Separate q and p

Figure 3.2: Complexity of Simplified MTC

The simpler algorithm can be seen to require time:

$(n \times l) \times (n + [n \times n \times l])$,

This complexity reduces to $O(n^3 l^2)$. The complexity saving this algorithm enjoys over Chapman's MTC algorithm is at the cost of some "over commitment" on behalf of the simpler algorithm. This commitment is explained in more detail in Chapter 2 on page 36. The simpler algorithm is the basis of the following analysis.

3.2.1.1 Strict Linearity

A fully linearized, or fully ordered plan is of the form $A_1 \prec A_2 \prec ... \prec A_n$ for a plan of n operators. Determination of the necessary truth of a precondition p in any operator A_i involves: For each operator in A necessarily before A_i, find A_e such that A_e establishes p for A_i by having p necessarily asserted or negated as an effect of A_e. Note that in a linear plan, there can only be one operator A_e that establishes or negates p.

1. If no A_e is found necessarily before A_i such that A_e asserts established or negates p, then the proposition p does not necessarily hold just before A_i, and the plan is incorrect.

2. If A_e negates p, then the proposition p does not necessarily hold just before A_i, and the plan is incorrect.

3. If A_e establishes p, then the proposition p necessarily holds just before A_i.

There are at most $i-1$ operators that necessarily precede A_i. Each operator has l effects. In the worst case, A_e is found in the $i-1$ operator comparison. In order to find A_e for each of n operators and l preconditions requires time:

$$l \times \sum_{i=1}^{n} \{[l * (i-1) + (1 \times (i-1) \times l)]\}$$

This complexity reduces to $O(n^2 l^2)$.

3.2.1.2 Multiple Parallel Orderings

Consider a plan that has k totally ordered subplans. Each of the k subplans is of size s, where the plan has n operators, and $n = s * k$. In addition, each of the k subplans follows the initial operator I, and precedes the goal operator G.

In a plan with k completely parallel (and fully ordered) subplans, the k^{th} subplan has a form $\{ A_{k_1} \prec A_{k_2} \prec ...A_{k_n} \}$ with an initial operator I necessarily before all A, and a goal operator G necessarily after. The determination of the necessary truth of a proposition p in any operator A_{k_i} is made in the following manner: For each operator in A find A_{k_e} such that A_{k_e} establishes p for A_i by having p nessarily asserted as an effect of A_e, or negates p. Since only those A in A_k necessarily precede A_{k_i}, A_{k_e} must be in A_k. If fact, as in a linear plan, there can only be one A_{k_e}.

1. If no A_{k_e} is found necessarily before A_{k_i} such that A_{k_e} either establishes or negates p, then the proposition p does not hold just before A_{k_i}, and the plan is incorrect.

2. If A_{k_e} is found necessarily before A_{k_i} such that A_{k_e} negates p, then the proposition p does not hold just before A_{k_i}, and further, the plan is incorrect.

3. If A_{k_e} is found necessarily before A_{k_i} such that A_{k_e} establishes p,

   For each operator C_k possibly between A_{k_e} A_{k_i}, (in this case there are at most $(n-s)$ such operators), if C_k possibly denies p, then p is clobbered, and p does not necessarily hold just before A_{k_i}, and further, the plan is incorrect. If no C_k exists that possibly denies p, then p necessarily holds just before A_{k_i}.

There are at most $i-1$ operators that necessarily precede A_{k_i}. Each operator has l effects. In the worst case, A_{k_e} is found in the $i-1$ operator comparison. In addition, the effects of the remaining $(n-s)$ operators possibly between A_{k_e} and A_{k_i} must be examined in order to determine if a clobberer exists. Thus the (worst) total cost to determine the necessary truth of a precondition in A_{k_i} is:

$$\{l * (i-1) + [(1 \times (i-2) \times l) + (1 \times (n-s) \times l)]\}$$

In order to determine that an A_{k_e} exists, and no clobberers C_k exist for each of l preconditions in each of n operators requires:

$$l \times \sum_{i=1}^{n} \{l * (i-1) + [(1 \times (i-1) \times l) + (1 \times (n-s) \times l)]\}$$

$$= l^2 n^2 \times (2 - (1/k))$$

This complexity reduces to $O(n^2 l^2)$.

3.2.1.3 Strict Parallelism

A completely parallel, or unordered plan has totally unordered operators $\{A_1, A_2, ...A_n\}$ with an initial operator I necessarily before all A, and a goal operator G necessarily after. Determination of the necessary truth of a precondition p in any operator A_i (other than G) involves: For each operator in A necessarily before A_i, (in this case only I necessarily precedes A_i), find A_e such that A_e establishes p for A_i by having p nessarily asserted as an effect of A_e, or negates p.

1. If I does not either establish or negate deny p, then the proposition p does not necessarily hold just before A_i, and the plan is incorrect.

2. If I negates p, then the proposition p is not satisfied, and further, the plan is incorrect.

3. If I establishes p, then the proposition p is established.

For each operator C possibly between I and A_i (in this case $n-3$ such operators exist), if C possibly denies p, then p is clobbered, and p is not necessarily true just before A_i, and further, the plan is incorrect.

There is only 1 operator (I) that necessarily precedes A_i. In the worst case, I is found in 1 operator comparison. Each operator has l effects. In addition, the effects of the remaining $n-3$ operators (exclude I, G, and A_i) possibly between A_e and A_i must be examined in order to determine if a clobberer exists. Thus, the complexity of determining the necessary truth of a all l preconditions in the n operators of the plan is:

$$l \times \sum_{i=1}^{n} \{(1 * l) + [1 \times (n-3) \times l]\}$$

This complexity reduces to $O(n^2 l^2)$.

The analysis of the three preceding types of operator orderings shows that the computation of MTC for plans restricted to orderings in one of these classes in simplified by an order of magnitude over the larger class of nonlinear plans. In the general case, MTC has complexity $O(n^3 l^2)$. However, in a totally linear plan, a totally parallel plan, or in a plan possessing k parallel total orderings of equal size, the complexity of MTC is only $O(n^2 l^2)$. The factor that makes MTC computation for fully linear, fully parallel, or k-parallel plans an order of magnitude simpler than the general case is the fact that in these cases only a single establisher may exist for any particular precondition, while in the general case, there may exist many establishers.

3.2.2 Plan Conflicts

In general, the truth criterion is used to create a list of *conflicts* that each plan has. Each conflict represents the manner in which some plan establishment fails to guarantee the necessary truth of the operator precondition it establishes. Based on the establishment relation described above, $Est(E_j, U_i, p_j)$, a conflict can be described in terms of the clobbering operator C_j interfering with the establishment. A conflict is represented as $Conf(E_j, U_i, C_j, p_j, q_k)$ where C_j is an operator possibly between E_j and U_i, and C_j asserts q_k such that q_k possibly denies p_j. Hertzberg [Hertzberg and Horz, 1989] shows that, based on the positional relationship of E_j, U_i, and C_j, any conflict can be classified as one of *Linear* (Figure 3.3), *Left Fork* (Figure 3.4), *Right Fork* (Figure 3.5), or *Parallel* (Figure 3.6). This conflict specification forms part of the conflict resolution approach of both TWEAK and ABTWEAK.

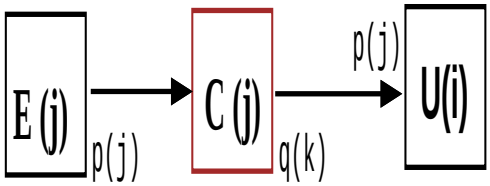


Figure 3.3: Linear conflict.

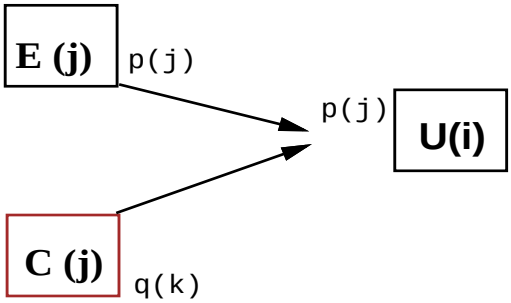


Figure 3.4: Left Fork conflict.

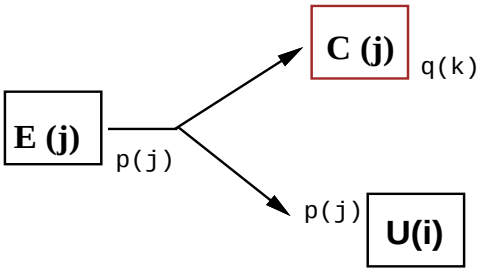


Figure 3.5: Right Fork conflict.

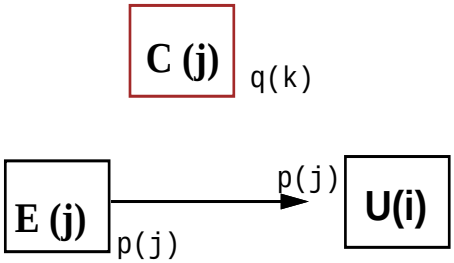


Figure 3.6: Parallel conflict.

3.2.3 Conflict Resolution

Based upon Chapman's [Chapman, 1987] MTC formulation for goal achievement, Yang [Yang, 1990b] has specified the application of MTC within the framework of conflict resolution. Yang's methods in [Yang, 1990b] includes a white knight operation described as demotion-establishment, however, as explained in the MTC outline, this operation can be left out without loss of completeness. In this manner, each of the aforementioned conflicts types of $Conflict((E_j, U_i, C_j, p_j, q_k)$ can be resolved by imposing plan constraints T indicated by MTC as follows: [2]

$$
\begin{aligned}
Resolve(Linear_{conflict})) \supset \quad T = \quad & (p_j \not\approx q_k), \\
Resolve(LeftFork_{conflict}) \supset \quad T = \quad & (C_j \prec E_j) + (p_j \not\approx q_k), \\
Resolve(RightFork_{conflict}) \supset \quad T = \quad & (U_i \prec C_j) + (p_j \not\approx q_k), \\
Resolve(Parallel_{conflict}) \supset \quad T = \quad & Resolve(LeftFork) + Resolve(RightFork)
\end{aligned}
$$

Basically, A *Linear* conflict can only be resolved by "separating" the established precondition p_j and the clobbering effect q_k $(p_j \not\approx q_k)$. A Left-Fork conflict can be resolved by either separating the precondition and effect, or by "demoting" the clobbering operator C_j such that it necessarily precedes the establishing operator $(C_j \prec E_j)$. A Right-Fork conflict can be resolved by separation, or by "promoting" the clobbering operator such that it necessarily follows the operator containing the established precondition $(U_i \prec C_j)$. Finally, a Parallel conflict can be resolved in the same way as either a Left-Fork or Right-Fork conflict: with separation, demotion, or promotion.

[2] Note that in the resolution equations, "+" indicates a logical 'OR".

As an example, consider the case shown in Figure 3.7 where we have added $PICKUP(BlockA, X)$ as the establisher of $clear(X)$ for the operator-precondition $STACK(BlockB, X)$ and $clear(C)$.

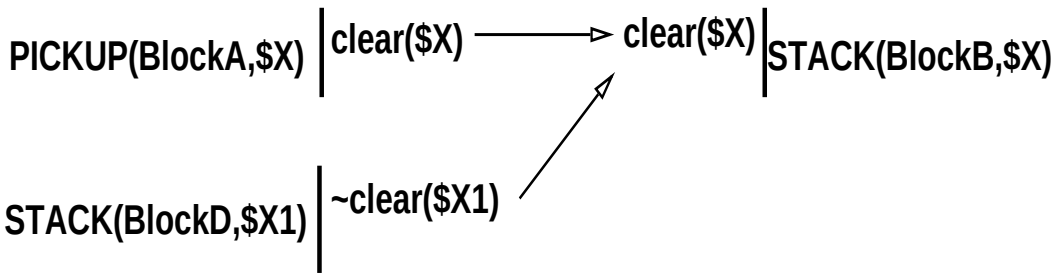


Figure 3.7: Left-Fork conflict example

We can identify a left-fork conflict caused by operator $STACK(BlockD, \$X2)$ which is parallel with $PICKUP(BlockA, X)$, and necessarily before $STACK(B, X)$. From the resolution methods outlined, our options for declobbering this conflict are to either use promotion as shown in Figure 3.8 or separation, as seen in Figure 3.9.

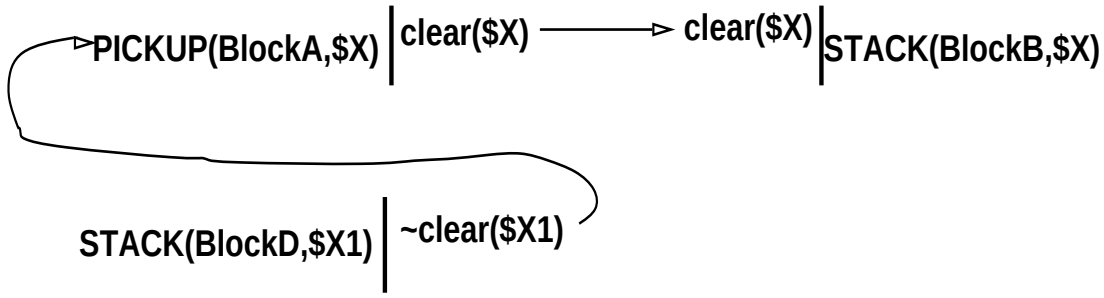


Figure 3.8: Left-Fork resolution 1, Promotion

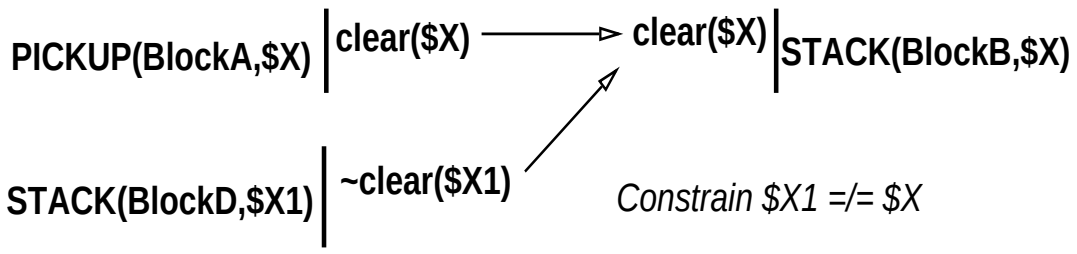


Figure 3.9: Left-Fork resolution 2, Separation

3.2.4 Successor Generation

The generation of plan successors is a two part process in the planner implementation presented in this thesis. The first step involves establishing an unsatisfied operator precondition in all ways possible, exactly as specified in the MTC goal achievement procedure, creating a set of *intermediate* plans or *I-plans*. MTC is then used to determine all conflicts in each I-plan which prevent the satisfaction of the selected precondition. In order for the precondition to be necessarily satisfied, all of the conflicts must be resolved. Each conflict can be classified in the manner specified above, and can be resolved based upon the methods indicated by Yang. Since each conflict may be resolved in several ways, and there are multiple conflicts possibly affecting a single precondition, we must consider all possible combinations of conflict resolution for each conflict in a cartesian product of solutions.

For example, consider the case where operator U has precondition p is established by operator E in an I-plan I_u, but I_u has 3 conflicts $C1$, $C2$, and $C3$ associated with U, E, and p. Now, if $C1$ is classified as a linear conflict, $C2$ a parallel conflict, and $C3$ a right fork conflict, then we see that there is 1 way (separation)

to resolve $C1$, 3 ways to resolve $C2$ (separation, demotion and promotion), and 2 ways to resolve $C3$ (promotion and separation). A cartesian product of these resolutions would result in a set of plan modification operations for generating all of the successors possible. This cartesian product "successor enumeration" is shown in Figure 3.10.

$$
\begin{aligned}
Plan_1 &= C1_{separation} + C2_{separation} + C3_{promotion} \\
Plan_2 &= C1_{separation} + C2_{demotion} + C3_{promotion} \\
Plan_3 &= C1_{separation} + C2_{promotion} + C3_{promotion} \\
Plan_4 &= C1_{separation} + C2_{separation} + C3_{separation} \\
Plan_5 &= C1_{separation} + C2_{demotion} + C3_{separation} \\
Plan_6 &= C1_{separation} + C2_{promotion} + C3_{separation}
\end{aligned}
$$

Figure 3.10: Cartesian product specification of plan successors

Each of the six members of the cartesian product shown above would be applied to the I-plan to which they apply, resulting in six possible successor plans for the particular I-plan.

Yang [Yang, 1990b] describes an algebra for the combination of these conflict resolution methods that will allow for the elimination of inconsistent combinations of modification operators. The implementation created for this thesis does not take advantage of the pruning possible with this algebra, and notices inconsistent plans only once the operations are actually applied to a plan.

3.3 Abstract Planning Search Strategies

In the search for a concrete level abstract solution plan, there are two major areas of search control to address. The first is the selection of which abstract or high level solutions to expand next in the search space. The second is how to expand this selected abstract solution at level k in the search for a less abstract solution at level k-1. The planning search space can be pictured as a tree in which each node is actually a single level search space determining a level k-1 plan based upon a level k solution. This search space representation can be better shown as a diagram, as in Figure 3.11. Within each node in the figure we are concerned with finding a single level solution. Within the larger search framework we are concerned with which abstract solution to concentrate upon.

Within each dimension of search control there are various heuristic approaches that are capable of improving planning performance. We will look at these approaches within the context of each level of control.

3.3.1 Search Within Each Level of Abstraction

Search at a single level, or at a single node in our two level search control diagram, is basically the same search control that TWEAK performs. Specifically, search control within a single abstract level determines which incomplete plan and unsatisfied goals to attempt to solve next. The selection of a goal within a plan determines the possible successor plans that will be generated. Within this successor plan set, it is possible to prefer some establishment and declobbering combination in a heuristic fashion, although any strategy must be careful in ignoring particular successors entirely. If we were to drop some successors entirely, the possibility exists that completeness could be lost.

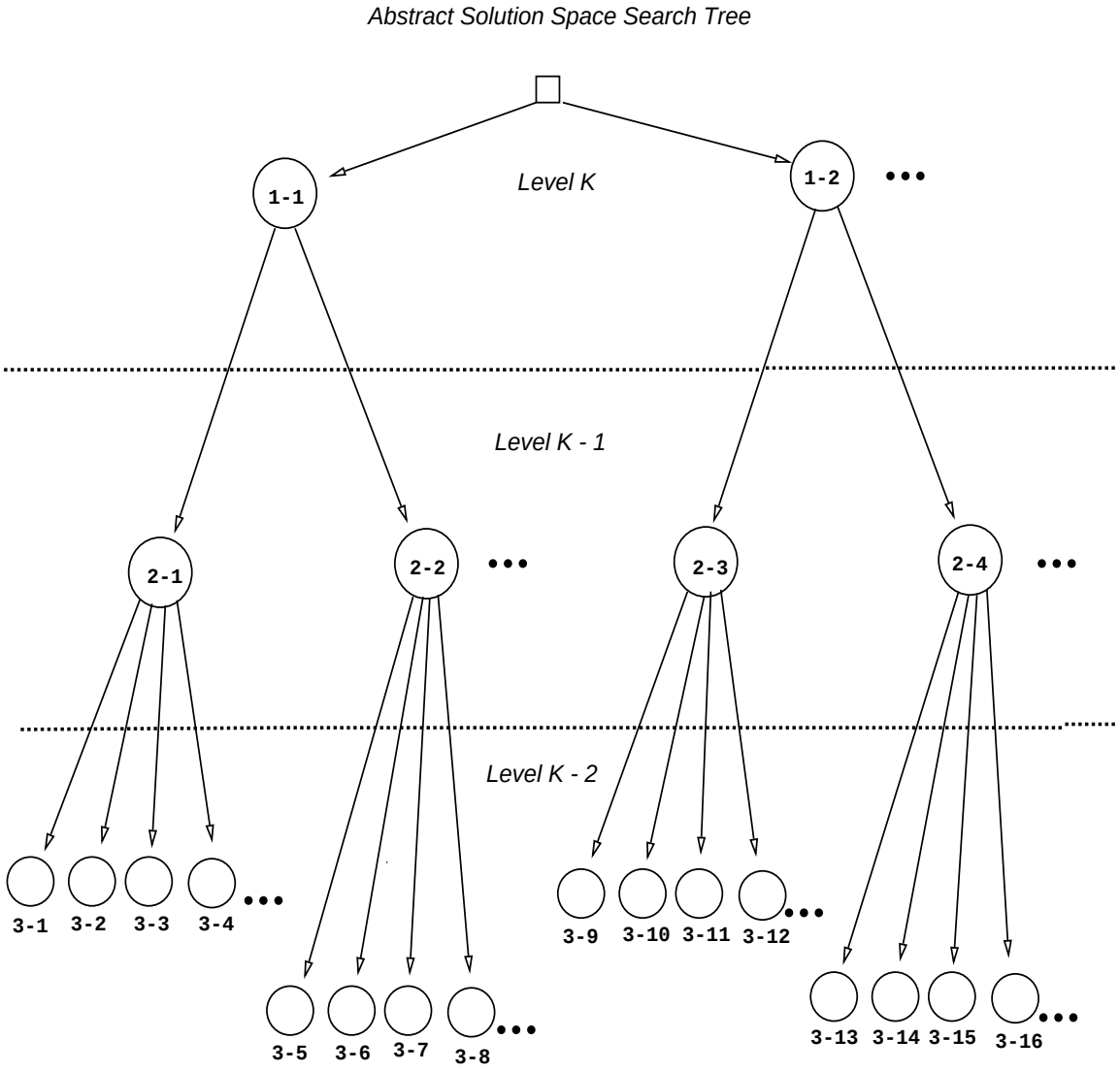


Figure 3.11: Representing the abstract solution search space

The addition of certain plan operators into an incomplete plan can result in conflicts with previously established and declobbered goals. As a result, some plan successors may make less "progress" towards a solution than others, simply because they cause conflicts to arise in the successor plan. These conflicts will have to be resolved at a later time, and this will involve more work on the part of the planner. The protection of previously established goals in successor plans is one way of reducing the branching factor of search.

One heuristic that allows us to reduce the branching factor through goal protection has been suggested in [Yang and Tenenberg, 1990]. The next section will outline this heuristic approach as a form of successor pruning within the context of abstract planning.

3.3.1.1 The Monotonic Property : Abstract Goal Protection

The purpose of subgoal protection is to ensure that the previously achieved tasks are not undone when completing a plan. In doing this, one would like to use subgoal establishment structure as a constraint on search, while preserving completeness of the systems. Towards this goal, Yang and Tenenberg [Yang and Tenenberg, 1990] define the *Monotonic Property*.

In a correct plan Π, if an operator α necessarily achieves a precondition p of an operator β, and if no other operators necessarily between α and β does so, then we say α *establishes* p for β, or *establishes*(α, β, p).

Informally, a refinement of an abstract plan is *monotonic*, if the subgoal-establishment structure of the abstract plan is preserved in the refinement. More precisely, let Π_a be a plan at abstract level i for solving goal G, and Π be a plan at level $i-1$ for solving the same goal. Suppose that in both plans every operator directly or

indirectly achieves a goal, as given by the establishment structure. Then Π is a *monotonic refinement* of Π_a if the abstract version of Π is Π_a. It has been shown that in every abstraction hierarchy based on precondition elimination, if there is a correct plan at the base-level, then one of the abstract plans can monotonically refined to a solution at the base level [Yang and Tenenberg, 1990].

The monotonic property imposes constraints on search by providing the ability to backtrack on violations of protected goal establishments. A violation occurs when another operator γ is inserted between α and β, where $establishes(\alpha, \beta, p)$, and γ necessarily asserts p. It was shown that backtracking on protection violations in this manner retains completeness in a search strategy [Yang and Tenenberg, 1990].

3.3.1.1.1 Strong and Weak Monotonic Properties Due to the nonlinear nature of plans, it is possible that a precondition p of an operator β is established by two or more unordered operators α_1 and α_2 in the plan (Figure 3.12).

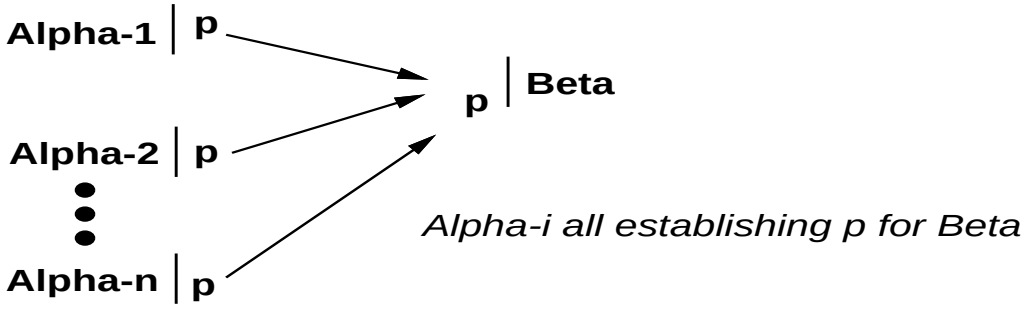


Figure 3.12: Two or more establishments of a single precondition

With this in mind, two versions of monotonic property can be designed:

Strong Monotonic Property (SMP): When refining an abstract plan, protect *all* establishment relations, i.e. include both $establishes(\alpha_1, \beta, p)$, and $establishes(\alpha_2, \beta, p)$.

Consider the case where Π_a is a plan at abstract level i for solving goal G, and Π is a plan at level $i-1$ for solving the same goal. Suppose that in both Π_a and Π, every operator U directly or indirectly achieves G.

An operator U_{direct} directly achieves G if one of its effects p asserts G, and an operator $U_{indirect}$ indirectly achieves G if one of the effects of $U_{indirect}$ asserts a precondition of an some other operator U_{other} where U_{other} either directly or indirectly achieves G. Every correct plan P has an implicit set of i establishments Est that completely describe these achievement relations. This set consists of relations of the form *Est(E_{est} U_{user}, p)* where E_{est} achieves proposition p for operator U_{user}, and where each Est_i is specifying either a direct or indirect achievement of G.

Π is a refinement of Π_a satisfying the *Strong Monotonic Property* of Π_a if the abstract version of Π is Π_a, and all establishment relations at level i, Est_i, in Π_a exist in Π at level $i-1$.

Weak Monotonic Property (WMP): When refining an abstract plan, protect at least one of the establishment relations, i.e., either $establishes(\alpha_1, \beta, p)$, or $establishes(\alpha_2, \beta, p)$, but not both.

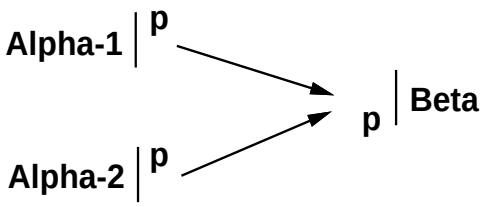
Definition 3.3.1 Π *is a refinement of* Π_a *satisfying the* Weak Monotonic Property *of* Π_a *if the abstract version of* Π *is* Π_a*, and at least one of the establishment relations* Est(E_{est},U_{user},p) *of every proposition* p *in* Π_a *exist in* Π *at level* $i-1$.

As suggested by their names, SMP imposes a stronger constraint on search, by backtracking on *any* monotonic violation. On the other hand, WMP backtracks only when *all* abstract establishment relations are violated. One would be tempted

to conclude that the strong version is always superior. Unfortunately, SMP cannot guarantee completeness in general. This can be shown by the following counter-example. Suppose for a planning problem, the only solution at the base level is a linearized plan,"$\Pi = \alpha_1, \alpha_2, \beta$." Also suppose that in the abstract plan both α_1, and α_2 are establishers for γ. Note that a nonlinear, least-committed planner does not compute every linearization of a correct abstract plan. Thus, protecting both establishment structures, the planner can never find the solution Π. On the other hand, with WMP completeness is preserved since $\mathit{establishes}(\alpha_2, \beta, p)$ is never violated. As a result of SMP's completeness sacrifice, the WMP version is used in experiments presented in this thesis.

3.3.1.1.2 Two versions of the Weak Monotonic Property To apply WMP in a straightforward fashion, one ensures for each abstract precondition p that, whenever a new operator is inserted into a plan, or a set of constraints is imposed upon a plan, at least one of the establishment relations for p still holds. More precisely, let $\mathit{establishes}(\alpha, \beta, p)$ hold in a abstract plan. In its refinement, let γ be an operator inserted *necessarily* between α and β. If γ necessarily asserts or denies p, then a monotonic violation occurs. This version of the WMP is called *necessary* weak monotonic property, or N-WMP. An example of a violation of the N-WMP is shown in Figure 3.13.

With more domain knowledge, one can do better. Then a *possible* monotonic violation occurs for $\mathit{establishes}(\alpha, \beta, p)$, whenever an operator γ is inserted necessarily between α and β, such that γ asserts q, and q possibly codesignates with p. For example, suppose a robot can only hold one thing at a time, and in an abstract plan, an operator α =`PICKUP(Cup)` establishes the precondition `holding(Cup)` for β =`FILL(Cup, Tea)`. If the operator γ =`PICKUP(x)` is inserted between α and β,



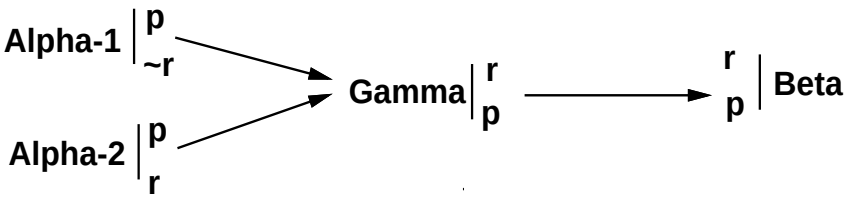
Level K Solution : Alpha 1 and 2 establish p for Beta

.....

Level K-1 Successor #1

no violation, since Est(Alpha-2, Beta, p) still holds

Level K-1 Successor #2



N-WMP violation: all level K Establishers of p violated at level K-1

Figure 3.13: Necessary Weak Monotonic Violation

a possible monotonic violation occurs. An example of a P-WMP violation is shown in Figure 3.14.

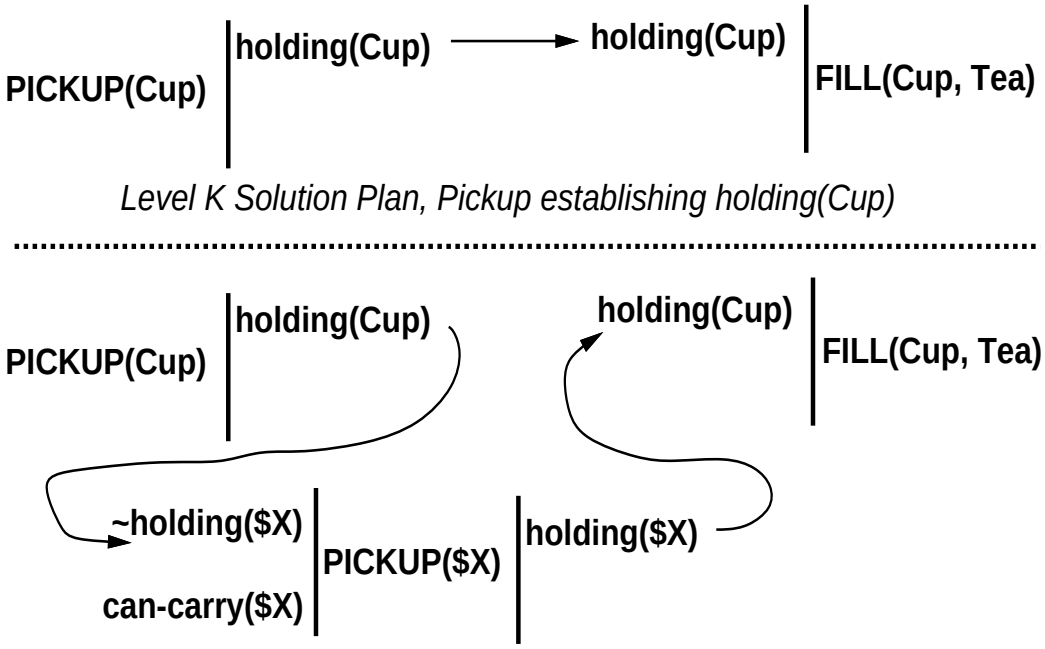


Figure 3.14: Possible Weak Monotonic Violation

If certain predicates of a domain satisfy the following *P-WMP exclusivity* condition, then prevention of possible monotonic violations guarantees completeness, and the hierarchy is said to have the *possible* weak monotonic property (P-WMP). The condition is as follows:

> Let r be a predicate. For two different sets of ground parameters X and Y, $r(X)$ and $r(Y)$ cannot hold at the same time.

Any domain exhibiting P-WMP exclusivity for a proposition type p cannot have

a correct plan where two predicates $p(a)$ and $p(b)$ are true just before some operator U, such that a and b are distinct.

Consider some abstract plan π_a correct at level k that contains an operator U with $p(a)$ as a precondition of U. $p(a)$ is established by an operator E in π_a.

Consider some descendant of π_a at level k-1, π. A P-WMP violation of π would exist if the establishment from level k, *Est(E, U, p(a))* is destroyed. Any plan violating this establishment possesses some operator C necessarily between E and U ($E \prec C \prec U$), and has an effect $p(x)$ such that x possibly codesignates with p. Since any complete plan must have all variables bound, x must be bound at some point in the planning process. Any binding of x must fall into one of two categories:

1. x is bound to a

   This binding thus would result in a plan which necessarily violates the establishment relation *Est(E, U, p(a))*, and thus constitutes a violation of the N-WMP.

2. x is bound to a term other than a

   Any binding of this nature constitutes a violation of the property of P-WMP exclusivity for p. Specifically, just before U, both $p(a)$ and $p(b)$ are true, where a and b distinct.

A direct consequence of this categorization is that:

> In any domain possessing P-WMP exclusivity for some proposition type p, every fully bound descendant of a plan π which contains a P-WMP violation of a p proposition will violate N-WMP.

This exclusivity condition is true for many predicates. In the robot example above, `hold` is one such predicate. Moreover, in the Towers of Hanoi example to be shown later, the predicates describing the location of individual object `on-small, on-medium, and on-big` are all of this type since each object can only be in one place at any moment in time.

P-WMP has the advantage of allowing successor pruning earlier in search than N-WMP. In fact, entire subtrees that would have been pruned one successor at a time by N-WMP are eliminated in one pruning by P-WMP, thus foregoing the expansion of many fruitless search paths.

3.3.1.1.3 Weak Monotonic Property Complexity In subsection 3.2.1 on page 61, the complexity of determining the necessary truth of operator preconditions via MTC in nonlinear plans is discussed. MTC determines whether or not an establisher exists for the precondition, and then insures that no clobberers exist for that establishment. Determining whether a N-WMP violation occurs requires examining all existing establishment relations $Est(E, U, p)$, and determining whether these relations still hold. Specifically, for each establishment, for each operator between E and U the effects must be checked to insure that p is neither necessarily asserted nor necessarily denied. If there are k establishments, d operators between E and U, and l effects for each operator, then the complexity of checking N-WMP violations is $O(k \times d \times l)$. Determining whether a P-WMP violation is of the same complexity, with the only difference being that the effects are checked to insure that p is neither possibly established nor possibly denied.

3.3.1.2 Goal Ordering

An incomplete nonlinear plan which exists in the search space while seeking a correct solution plan is incorrect if one or more of the operators in the plan have at least one precondition which is not necessarily true in every completion of the plan. We know that in order to find all of these unsatisfied preconditions would require time on an order of $O(N^4)$, however, in order to continue planning we need only find one unsatisfied precondition to repair at a time, a much cheaper requirement in the average case. Although it is true that Chapman's approach is complete no matter which of these preconditions are selected, it is not at all obvious how to best choose one which will help to limit the search for a solution.

It must be noted that planning with abstraction using criticalities in AbTweak is essentially a method of ordering goals in the plan. The first goals selected and solved involve those preconditions which have been deemed more abstract or general. Within these goals at each level of abstraction, we also must select which to solve first. This selection process determines completely the branching factor at any given point, and the order in which these successors are generated has a profound effect on any search with breadth-first characteristics.

Certain orderings within a level can create many successors that are committed to incorrect constant bindings, yet these successors may need to be explored with the same preference as the single successor which is actually on the solution path. Individual level goal ordering determines the performance of individual level planning search, and since poor individual level search performance can dominate the overall search performance of AbTweak, poor goal ordering can essentially cripple any goal-driven planner. Similarly, good goal ordering can drastically improve search performance, even to the point where a non-abstract approach outperforms

an abstract approach in a domain well-suited to abstraction.

While examining the behaviour of TWEAK, an interesting factor was discovered which explains some of the unexpected efficiency observed for TWEAK in this domain. A simple implementation of TWEAK might provide goal selection by simply repeatedly selecting the first discovered possibly false precondition, solving it as described by MTC, and repeating this process until a solution is found. Specifically, the implementation of TWEAK presented in this thesis outperforms ABTWEAK in the Towers of Hanoi problem.

This unintuitive result can be explained by the tendency of TWEAK to select goals such that subplans are created which satisfy the primary goals of a particular problem individually. In this manner, TWEAK plans are built in a manner that exploits a belief that plans tend to possess independent, noninterfering subplans.

An example of this behaviour can be seen in a simple domain where a robot can move blocks about in a set of rooms. Consider an example where there are two blocks: Block1, and Block2. Each of these blocks can be picked up by the robot via a PICKUP(block) operator. The robot can move between one of two existing rooms via a GO-ROOM(from-room, to-room) operator, and can carry a block between rooms via a CARRY-ROOM(block, from-room, to-room) operator. Initially, the robot is in Room1, Block1 is in Room1, and Block2 is in Room2. A simplifying assumption is that there is only one door, and a robot can always go through this door. If the primary conjunctive goal consists of the propositions in the set { InRoom(Block1, Room2), InRoom(Block2, Room1) }, then two possible subplans can be built, one for each goal.

The first subplan might be built in the following manner:

1. The primary goal InRoom(Block1, Room2) can only be accomplished by one

operator, CARRY-ROOM(Block1, Room2), so a partial plan is built solving InRoom.

2. The operator CARRY-ROOM(Block1, Room2) must have a precondition Holding(Block1) that needs to be satisfied. Only PICKUP(Block1) can satisfy this goal, and so PICKUP(Block1) is added to precede CARRY-ROOM.

In a similar manner, the second subplan satisfying the primary goal InRoom(Block2, Room1) would be built such that CARRY-ROOM(Block1, Room1, Room2) is added first, and then the operator PICKUP(Block2) is added.

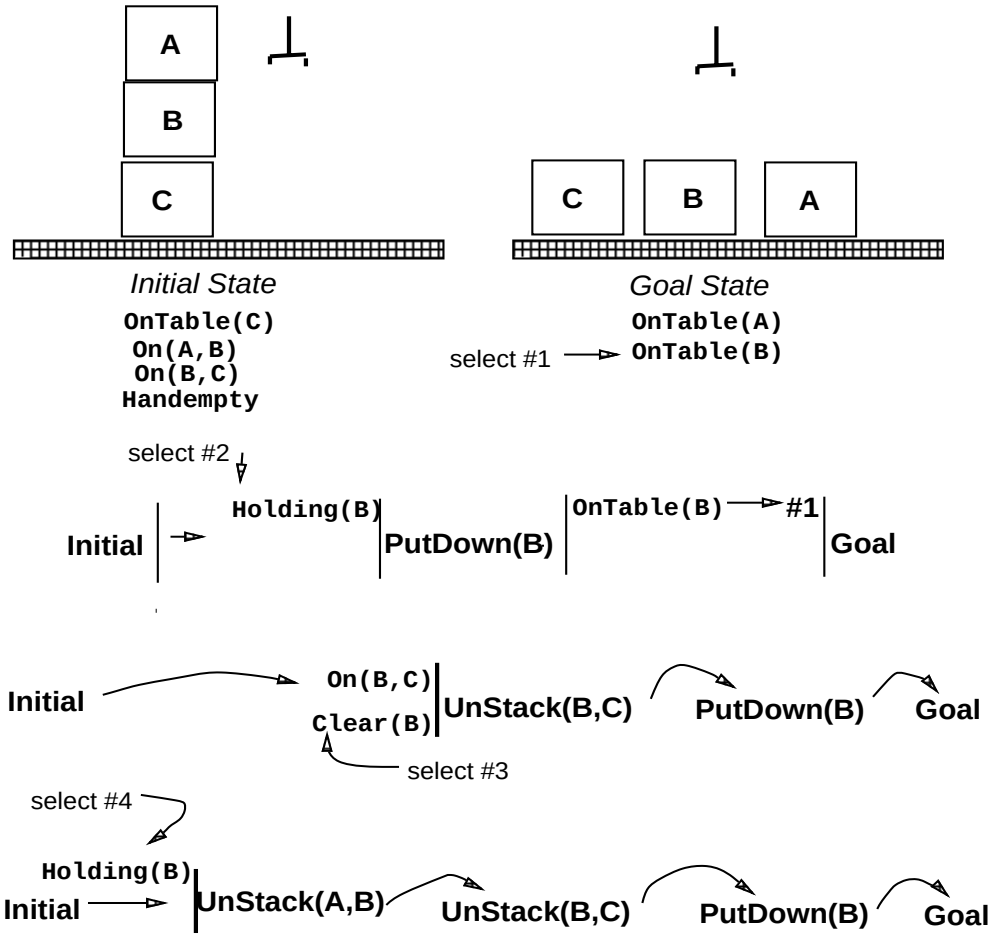
TWEAK tends to solve conjunctive problems in the following fashion. First, each primary goal is planned for completely, and only after all are complete is the interaction between the plans looked at and resolved. In this example, the location initially of the robot in Room1 would require that the first subplan precede the second subplan in order for a total plan to solve both primary goals. Furthermore, TWEAK adds operators in a very linear fashion while building each subplan. CARRY-ROOM was added to solve the condition InRoom, and then PICKUP was added to solve a precondition of CARRY-ROOM. Operators chain together in this fashion, where the most recently added operator in the plan is chosen to have a precondition satisfied next. This chaining action captures an implicit assumption about planning: that operators recently added to a plan will not interfere with the goal older operators added to the plan have accomplished. Many domains exhibit this subgoal and operator independence, and so an exploitation of this independence via careful subgoal selection proves profitable. ABTWEAK is not capable of chaining operator linearly, primarily because the operator set is partitioned across the criticality levels, and different operators are necessarily added to a plan at different hierarchical levels. As a result, TWEAK can outperform ABTWEAK in

certain problem instances. An example where TWEAK profitably selects the first discovered precondition in the most recently added operator is shown in Figure 3.15.

The very act of separating a problem into levels of abstraction can destroy the ability of a planner to plan for all goals at a single level, and thus eliminate the possibility of taking advantage of the right-to-left construction assumption and the operator independence of the domain. Stated quite simply, some domains exhibit a large degree of linearity across a single abstraction level. Domains such as Nilsson's Blocks World, where there is little difference in the difficulty of applying various operators, it is not at all clear what makes one operator more abstract in any real sense. Abstracting a domain such as this "artificially" tends to gain little from abstraction, and in fact removes whatever advantage would be gained from selecting operators that can "chain" naturally.

In an attempt to more fully understand this goal ordering phenomenon, we define two approaches for subgoal ordering, each applicable to some extent in both abstract and non-abstract planning.

3.3.1.2.1 Stack Goal Ordering In an attempt to more accurately specify a goal ordering approach that tends to take advantage of the aforementioned subgoal independence, we define the *Stack* goal ordering method. This method essentially attempts to solve the preconditions in the "newest", or most recently added plan operator first. This approach essentially places operators in a plan "stack", where the most recent addition sits on top, and the oldest addition on the bottom. In simple terms, the *Stack* approach equates to not bothering to try and repair any conflicts that may arise between subgoal solutions until the very end of the planning process, basically assuming that any unsatisfied preconditions of "older" operators will go away as a result of the addition of future plan constraints. In the sense of



The Nilsson Blocks World operators tends to select goals that chain operators in a linear subgoal solution. The precondition set of each operator basically drives the next action selection, and since the operator set is small, selected actions work well.

Figure 3.15: Profitable first goal selection in TWEAK

repairing conflicts that appear in previously completed subplans, this approach is very *casual*. An example of *Stack* approach is given in Figure 3.15.

3.3.1.2.2 Tree Goal Ordering In contrast to the casual approach of *Stack*, it is possible to attempt to repair any conflicts that do arise in the existing operators of the partial plan as soon as they are detected. In order to achieve this "repair" of partial plans, we could implement operator goal selection as a queue. Oldest rather than youngest operators are selected for goal achievement first. Unfortunately, this method does not quite accomplish subplan repair, since operators added as "repair" are very young, and so any conflicts in these operators would be postponed. If we truly wanted an approach that "repaired" subplans, we would have to recognize the "reason" each operator was added to a plan in some hierarchical fashion. For example, operator A1 and operator A2 were added to establish preconditions of operator B, while operator B was added to establish a precondition for operator C. In this scenario, we would first check the operator B establishments for C, then the A establishments for B, etc.

One such method of "repair" could be accomplished by selecting operators by inorder traversing a carefully built tree of operators. Operators are added to a tree as children of the operator they are establishing for. A traversal of such a tree in a top-down manner would essentially equate to an attempt to maintain the establishments already made earlier in the planning process. Operators added as "repair" to existing subplans would be placed as children of the operator they fix. This approach, named the *Tree* subgoal ordering method, tends to assume that subplans achieving previous subgoals should be continually "repaired" as planning progresses, and in fact should be completely satisfied before new future goals are solved. In this respect, the *Tree* method embodies a more protective or strict

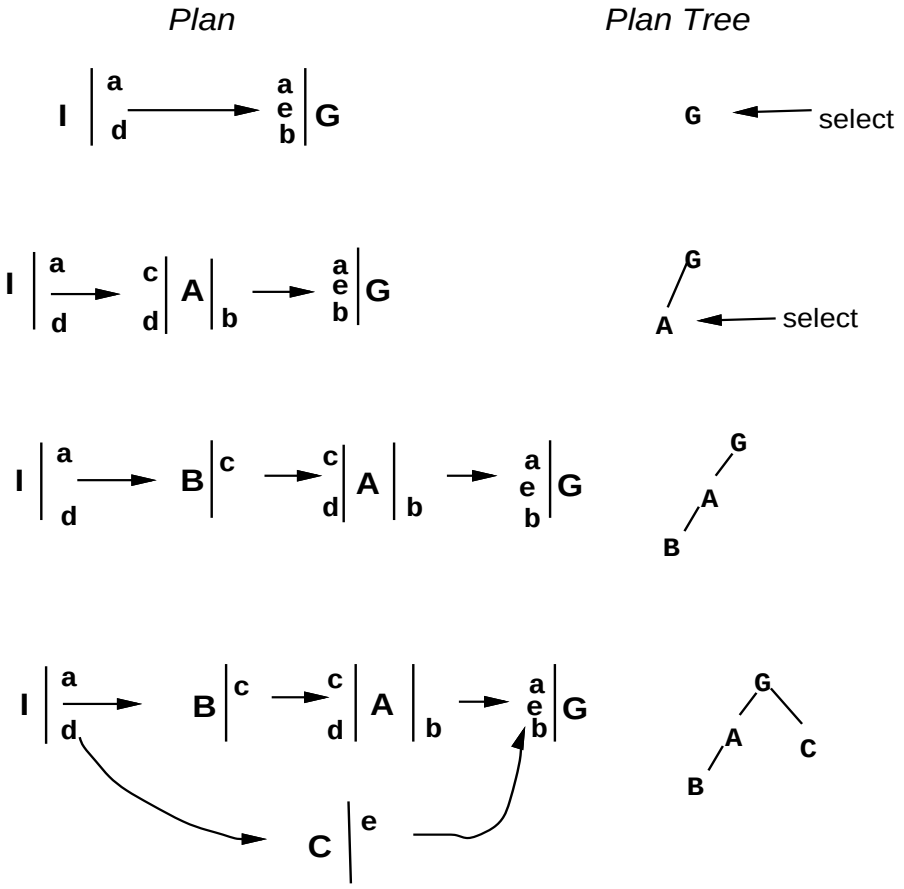
attitude towards established goals. The *Tree* structure grows with operators added as children of the operator they establish a proposition for originally. Search in the tree for the next operator to check progresses Left-Right-Node, thus attempting first to build a subplan for the top level subgoal in G (Goal), then commencing a second subgoal, but resolving any conflict in the first (left most) subgoal as soon as it occurs. An example of the *Tree* operator ordering approach is given in Figure 3.16.

3.3.1.2.3 Random Goal Ordering In order to justify any claim one would like to make about either *Stack* or *Tree* goal ordering, it would be useful to see how planning strategies would behave if this control level were removed entirely. If we were to select these goals randomly and see a marked decrease in performance, it would be much more clear that either *Stack* or *Tree* are reasonable.

The importance of subgoal selection is emphasized by several experiments we performed utilizing random subgoal selection in place of either *Stack* or *Tree* methodology. While *Random* is capable of reaching a solution in fewer expansions than *Stack* or *Tree* cases, the effective removal of a goal ordering method preformed much worse in the average case. In fact, random is capable of causing pathologically bad goal selection, effectively outweighing any search benefits achieved through abstraction, thus backing up our previous claim that it is not clear which (if either) of the two search control dimensions is more important to a certain planning problem.

There is much work left to be done in determining an optimal goal selection strategy in nonlinear planning environments, and we have touched only briefly on two simple, opposing approaches for the purposes of comparison.

3.3.1.2.4 Goal Ordering Summary In attempting to compare the performance of the abstract planner AbTweak with the non-abstract planner Tweak,



In the last plan, a new top level subgoal is started by C for e. Rather than pursue C preconditions like STACK would, TREE will now check ops B then A before continuing with C. All ops other than G must be checked before a new top goal starts.

Figure 3.16: Tree goal ordering

we must realize that the comparison cannot be "fair" in that the very abstract nature of AbTweak restricts its ability to fully plan (at the concrete level) for each subplan, as Tweak is capable of. While this observation, coupled with Tweak's superior performance in some cases, tells us something about the nature of plans in general, it does not allow us to fairly evaluate the utility of abstraction. In order to restrict our "field of view" to abstraction, we must eliminate this implementational advantage of Tweak. One way of doing this is to randomize individual goal selection for both Tweak and AbTweak, thus removing Tweak's advantage. Another (more efficient) way is to give AbTweak a similar "flavor" of advantage. In attempting the later, the *Tree* approach to goal ordering was born.

3.3.1.3 Operator Set Applicability based upon Primary Effect

Some approaches to planning, such as shown in [Minton, 1989], have attempted to restrict the branching factor of plan successor creation by limiting the kinds of operators that are allowed to establish certain goals via control rules. For example, it may be the case in some domains that certain goals should only be accomplished via sideeffects from existing plan operators, and that never is it necessary to actually add a new operator to the plan to accomplish these goals. Specifically, in the robot domain, an operator like PUSH-OBJECT-THROUGH-DOOR might have several effects such as changing the room that a given object is in, and as well, of changing the room that the robot is in. If a problem in this domain had an unsatisfied goal such as InRoom-Robot(Room1), a possible new establishing operator like ROBOT-GO-THROUGH-DOOR could be added to the plan. Potentially, PUSH-OBJECT-THROUGH-DOOR could also be added as a new establisher since it too can assert InRoom-Robot. However, this addition does not make intuitive sense if one assumes that we only want to do as much work as necessary in order to accomplish the goals

given. A simple plan of getting to Room1 should not also involve the unnecessary manipulation of objects in the domain. We define a domain rule in order to address this issue such that encountering a particular goal does not result in the addition of all possible establishers in the operator set of the domain, but rather only a certain, predefined subset of these operators. Specifically, operators are determined to have a *primary effect*. Operators are only *added* to a plan in order to satisfy a precondition corresponding to the operator's primary effect.

While this approach seems to make sense intuitively, and can certainly reduce the overall branching factor of search, it must be used carefully, since it does risk sacrificing completeness of the planning search strategy in general. Consider the following example in the robot domain with the additional restriction that the robot may only enter a room once. If we had two goals: InRoom-Robot(Room1), InRoom-Object(Box1), and we selected InRoom-Robot(Room1) as the first goal from which to create successors, we might only add ROBOT-GO-THROUGH-DOOR to the plan as establisher. From now on, every possible plan in the space is committed to including this operator. Now, selecting the second goal InRoom-Object(Box1) leaves us with no existing establishers in the plan for it, and we must add a new operator PUSH-OBJECT-THROUGH-DOOR. The problem is that operators cannot be merged, and both operators cannot occur at the same time. Since only a single goal is chosen as the branching point at any one successor generation, it must be insured that, for the domain in which operator sets are limited in this way, that the goal chosen would always preclude missing a solution. In this example, it would have to be the case that the goal InRoom-Object(Box1) was always chosen over InRoom-Robot(Room1).

3.3.2 Search Across Abstract Search Levels

We are interested in finding complete search strategies to coordinate planning at different levels of abstraction. This is a difficult problem because we would like AbTweak to be both complete, and more efficient than planning without abstraction.

It must be noted that a simple-minded application of Tweak's search strategy at each level of hierarchy is *not* complete across different levels. Tweak is only semi-decidable in the sense that termination is not guaranteed if planning for a given problem which does not have a solution. A particular abstract solution at level k may not be refinable to a solution at level $k-1$. As a result, an abstract strategy that is committed to only an individual level-k solution may never find a concrete level solution, even if the search within each level is complete.

AbTweak's search strategy does not overcommit in this fashion, 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 3.11 an abstract search space is presented. In this figure, notation 1-1 indicates the first abstract solution at the highest level of abstraction, 2-2 the second, and so on. Beneath 1-1, abstract solutions at level k-1 are found, and beneath them, solutions at level k-2. Abstract search control strategies choose which abstract plans at which abstract levels to prefer in terms of expansion within that levels. Preferring plans on the left of the left-to-right generation of abstract plans equates to preferring simpler abstract plans. Later abstract plans found at a level involve more operators, and are considered more complex. Any abstract strategy must somehow assign "weights"

to the solutions in the plan, thus determining how "depth-first" a strategy is with respect to commitment to simple, abstract plans. A breadth-first strategy through this space might ignore abstraction level for selection, and choose based upon plan size irrespective of depth of a particular plan in the abstract space.

An obvious control strategy for search control is to use breadth-first search. In the search space, a state corresponds to a plan. During each iteration, a state Π is selected for correctness check 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. 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 state in the search tree is simply the total number of operators in the plan.

One problem with the above strictly breadth-first search strategy is that plans at higher levels of abstraction are preferred equally with plans at lower abstraction levels. Another way to organize search is to prefer the selection of a lower level plan *more* than an abstract one, because a less abstract plan is potentially closer to a concrete level solution. The reason for this is that if we simply want to obtain any concrete level solution, it may be beneficial to progress more deeply in the abstract space beneath existing high abstract level solutions, rather than continue to generate more high level possibilities. Towards a more "depth-first" yet complete strategy, we have devised a hybrid strategy which retains the completeness of breadth-first, and also has the advantage of preferring less abstract solutions in a somewhat depth-first manner. This is called the LEFT-WEDGE strategy. Given two states Π_1 and Π_2 in the search space, such that Π_2 is at a more abstract level than Π_1, it assigns a higher priority (or less cost) to Π_1 than Π_2, such that for every plan expansion to Π_2, several more expansions are done for Π_1.

In general, the LEFT-WEDGE abstract search strategy is expected to work well if a base-level solution tends to have simple, short abstractions. One can view the search tree as organized by ordering the shortest solutions on each level in a left-to-right way. Then search basically proceeds along the left wedge of the search tree, the tip of the wedge being deeper towards its left side.

3.3.2.1 Hierarchy selection

In planning with abstraction, the construction of the hierarchy itself is of great importance. Certain methods have been suggested for the automatic generation of hierarchies, as in [Knoblock, 1991]. From the results presented this thesis, the effect of chosing various *good* and *bad* hierarchies can be observed, and a relationship becomes evident of the relative benefit of certain hierarchies and certain indicators that arise from the planning process itself.

In general, a hierarchy that results in a low branching factor high up in the search space will likely outperform one that has a high branching factor. However, hierarchy selection also affects the goal ordering process, and this can be a non-trivial change. The relationship between hierarchy selection and planning performance is not well understood in general, largely because the factors that control search performance are not well understood or even identified completely. In an attempt to learn something about the relationship between hierarchy selection and planning performance, simple ways of preferring one hierarchy over another are shown, based upon the behaviours of various planning control strategies and properties.

3.3.2.1.1 What is a good hierarchy? A *good* hierarchy is one that allows the planning process to proceed in a top-down manner, from most abstract to

least, and results in as little destruction as possible of work done at high levels when planning at low levels.

3.3.2.1.2 How do we tell a good hierarchy when we see one? I present evidence in Chapter 4 that suggests that the comparison of violations of the monotonic properties while planning indicate the relative benefit of a given hierarchy in a particular domain. This seems to make sense, since violations indicate an attempt to undo planning accomplished at high levels of abstraction by planning at lower levels. The question of identifying "good" hierarchies is dealt with more completely by Craig Knoblock in [Knoblock, 1991].

3.4 Aspects of Nonlinear Planning

3.4.1 Finite Constants

Chapman proves in his "First Undecidability Theorem" that planning is undecidable provided that problems have a potentially infinite initial state, and that the shortest plan to solve a problem is not arbitrarily large. One area Chapman points out that is open to discussion is whether or not planning is undecidable in the case where problems do not have a potentially infinite initial state.

Clearly in any real world example, the initial state (specified at least) must be finite. We know that extending TWEAK's representation (even for finite states) proves to be undecidable, but there still is an open question as to whether TWEAK itself is decidable in its described form for finite states.

At first glance, it would seem a desirable quality to limit the number of constants available in the planning process. Chapman points out that this limitation make

the constraint computations of TWEAK NP-Complete [Chapman, 1987]. A simple example will also show how this limitation causes TWEAK's truth criterion to fail. Consider a domain where only two constants A, and B exist. Any unbound variable in this domain can therefore refer only to A or B in any fully bound plan completion. Consider the unconstrainted variables in the plan shown in Figure 3.17.

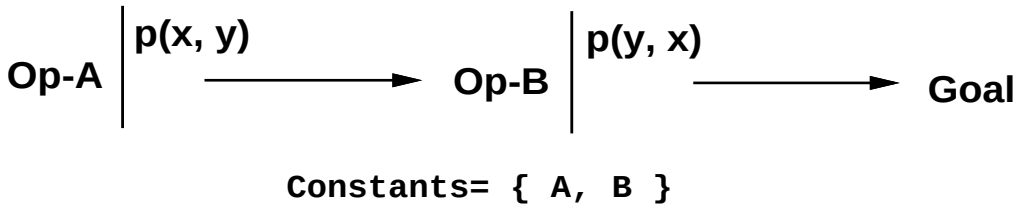


Figure 3.17: Pathological Plan and Finite Constants

This plan does not syntactically *necessarily* assert either $p(A, B)$ or $p(B, A)$. However, semantically, the plan requires that both $p(A, B)$ and $p(B, A)$ would hold just after execution of the second operator as a result of the range and codesignation constraints. In any plan completion, either variable x codesignates with A and y with B, or vice versa. In the former case, the first operator asserts $p(A, B)$ and the second operator asserts $p(B, A)$. In the later case, the first operator asserts $p(B, A)$, and the second $p(A, B)$. So, if MTC in TWEAK were asked to determine the truth value of $p(A, B)$ or $p(B, A)$ just after the second operator, the best it would do is return that they are both *possibly* true, while in fact they are both *necessarily* true. If the planner had been given the conjunctive goal in this way, it would say the plan did not necessarily solve both goals, when in fact it does.

It is tempting to attempt to modify the truth criterion to accommodate this

type of counter-example. While this may be possible for certain domains, it would seem that this general problem is a result of the NP-Complete nature of constraint satisfaction pointed to by Chapman.

3.4.2 Nonlinear Chaining of Operators

A nonlinear, least-commitment approach to planning results in a search space that "keeps its options open". In the Hanoi domain, planning to move a ring from one peg to another can be simple or complex. For example, you can move the ring directly from Peg1 to Peg2, or you can move the ring from Peg1 to some intermediate peg, and then to Peg2. In fact, a true least-commitment approach will leave the intermediary pegs unbound to any real peg if possible, given the domain. The construction of a plan by a goal-driven planner results in plans that tend to add partially bound operators, achieving these goals in a very indefinite manner. For example, a goal like On(Ring1,Peg1) might be achieved by some operator MOVE(ring,from,to), where "ring" indicates which ring to move between "from" and "to" locations. Achieving On(Ring1,Peg1) in least-committed fashion places no restrictions on "from" whatsoever, and so an operator like MOVE(Ring1,$from ,Peg1) might be added, where "$from" is left unbound. A precondition of this MOVE operator would have to be On(Ring1,$from), and this precondition can be achieved by adding another Move operator to the plan, this time "from" some other unspecified location, to the "$from" precondition location. So, while one plan successor might simply bind or constrain the unbound "$from" to some existing Peg value, new operators will also be added in other sibling successors. In this fashion, the act of adding new operators with each successor branching point in the search space (known as "chaining") propagates into a very complex set of plans, each with more and more unsatisfied preconditions, and each potentially competing

with other plan successors with fewer operators. Since the goal selection criteria does not differ from one successor point to the next, it is possible that the selection of an unbound proposition can result in the addition to the plan of an unbound operator to solve this proposition. On the next successor generation, an identical unbound proposition is selected in the new operator, thus causing the addition of yet another identical operator. In this way, the planner "loops" and causes the number of potential plans in the search space to grow dramatically. As the size of the search space grows, so too does the required amount of effort that must be expended in finding a solution, since the cost of exploring fruitless subtree increases. Furthermore, these subtrees do not "go away", but rather continue to propagate themselves into more subtrees, with more and more "looped" operators.

Other planning paradigms also suffer from this operator set growth, however, in nonlinear, least-commitment planning, the problem is intensified. The reason for the additional concern in our nonlinear planning paradigm is that many of these identical "chained" operators will be only partially bound and partially ordered in each plan. These two factors contribute to the total number of *possible* interferences with existing goals, as explained earlier when discussing the effect of nonlinearity on MTC. While this "looping" effect may indeed lead to a correct solution in some domains, it is does not seem to be a common feature of many domains that chained multiple identical operators are required in order to solve problems in these domains.

Additionally, successor generation for propositions will require more declobbering as a result of the increased number of operators *possibly* between the establishing operator and the established-for operator, that have effects which *possibly* deny the established precondition. More declobbering actions result in a larger cartestian product set of plan successors, effectively increasing the branching factor of the search space for *poor* goal selections.

3.4.3 Loop Detection

As mentioned earlier, the A^* graphsearch has been used as the planning search "engine" of my TWEAK and ABTWEAK implementations. A^* is complete, and allows for easy application of plan selection heuristics as a result of its method of maintaining a list of plans on the "Open" frontier. These plans in the "Open" list are essentially leaf nodes in the search space, and the search graph growth is partially controlled by which leaf is selected to be expanded next. A^* also provides for the maintenance of a list of plans known as the "Closed" list, essentially all plans that have been expanded during the search. Many domain-dependent versions of A^* take advantage of the "Open" and "Closed" lists by checking current successor plans as they are expanded to see if they have been visited before. In this way, duplicate search trees are guaranteed not to be explored during search.

The implementation of checking successor plans against visited plans is not a simple matter in nonlinear planning. In order for this checking to occur, a determination must be made whether two directed graphs (each representing the operator ordering of a plan) are in fact identical. This graph problem, known as the Minimum Equivalent Graph problem, is known to be NP-complete [Horowitz and Sahni, 1984].

As a result of the complexity of this problem, no duplicate checking is performed in ABTWEAK. However, it should be noted that, based upon observations of the search graphs produced in planning, and upon the goal-driven nature of TWEAK and ABTWEAK duplicate search paths do not occur frequently.

Chapter 4

Experimental Results

4.1 Introduction

The experimental results presented in this section can be categorized as follows:

1. Results which demonstrate the benefit of planning with abstraction in AbTweak, as opposed to planning without abstraction in Tweak.

2. Results which show how goal protection, implemented through the Monotonic Property, can improve abstract planning performance.

3. Results which exhibit how a causal approach to planning in terms of conflict correction by goal ordering outperforms a more rigid approach. This "casualness" is motivated by subgoal independence.

4. Results which lend support to the use of a new complete planning strategy known as Left-Wedge over the use of breadth-first in an abstract, nonlinear, least-commitment planner.

The results will be presented graphically where possible, and tabularly elsewhere, within this framework. The results presented are obtained from the Towers of Hanoi domain with 2 disks, and with 3 disks, from the Blocks World domain as defined by Nilsson [Nilsson, 1980], and from a Robot and Box domain derived from Sacerdoti [Sacerdoti, 1974].

4.2 Domains

4.2.1 Towers of Hanoi

The 3 disk Hanoi domain (Hanoi-3) is formulated as follows. There are three pegs and three disks, of different sizes. A disk can only be placed on a peg, or a disk bigger than itself. Initially, all disks are on `Peg1`. In the goal state, all three disks are on `Peg3`. The predicates used in representing the domain are listed in Table 4.1.

Predicate	Meaning
Ispeg(x)	x is a peg
OnBig(x)	location of the big disk
OnMedium(x)	location of the medium disk
OnSmall(x)	location of the small disk

Table 4.1: Towers of Hanoi

This domain has been extensively tested in the past with linear, abstract planners [Knoblock, 1991]. Most researchers have based their tests on one obvious hierarchy, $crit(ISPEG) = 3, crit(OnBig) = 2, crit(OnMedium) = 1, crit(OnSmall) = 0$.

In complex domains, and even in quite simple ones like Hanoi, it is not obvious what abstraction hierarchy is the best, or even which hierarchies are better than others. The more propositions in a domain, the more difficult this question becomes since there are many possible groupings of propositions at any abstraction level.

In order to fully test the utility of the monotonic property, and search strategy, it would be necessary to test all possible permutations of the hierarchies. While this is not possible because of the incredibly large number of combinations for even simple domains, I have chosen hierarchies that I believe constitute a representative cross-section of the hierarchies that exist.

In the Hanoi domain, for ease of exposition, we use *ibms* to represent the hierarchy of propositions described in Table 4.1, such that Ispeg has the highest criticality (is the most abstract), and OnSmall has the lowest (is the least abstract) . Similarly, *smbi* represents a hierarchy with the reverse order of criticality assignments. Table 4.2 details how the propositions are grouped and labeled by criticality for the purposes of experimentation. Unless otherwise noted, a negation of a proposition has the same criticality placement as the positive version of the proposition.

Group "i"	Group "s"	Group "m"	Group "b"
Ispeg	OnSmall	OnMedium	OnBig

Table 4.2: Tower of Hanoi Criticality Groupings

The operators for moving the disks can be represented as shown in Figure 4.1.

MoveBig (x, y)

Preconditions $\{Ispeg(x), Ispeg(y),$
$\neg OnSmall(x), \neg OnSmall(y),$
$\neg OnMedium(x), \neg OnMedium(y),$
$OnBig(x)\}$

Effects $\{\neg OnBig(x), OnBig(y)\}$

MoveMedium (x, y)

Preconditions $\{Ispeg(x), Ispeg(y),$
$\neg OnSmall(x), \neg OnSmall(y), OnMedium(x)\}$

Effects $\{\neg OnMedium(x), OnMedium(y)\}$

MoveSmall (x, y)

Preconditions $\{Ispeg(x), Ispeg(y), OnSmall(x)\}$

Effects $\{\neg OnSmall(x), OnSmall(y)\}$

Figure 4.1: Towers of Hanoi Operators

4.2.2 Nilsson's Blocks World

The Nilsson's Blocks World domain is formulated in the following manner. There is a table with blocks on it, and a robot arm which can stack and unstack, pickup and putdown these blocks. Each block may be either on the table, on another block, or in the robot's hand. The number of blocks (and their names) is specified in the initial state, and is not part of the domain definition. In the particular case we are interested in, there are three blocks, `A`, `B`, and `C`. Initially, `C` is on `A`, and `B` is on the table. The goal state has `A` on `B`, and `B` on `C`, while `C` is on the table. The predicates used in representing this domain are listed in Table 4.3.

Predicate	Meaning
On(x y)	Block x is on y
Clear(x)	Block x has nothing on it
OnTable(x)	Block x is on the Table
Holding(x)	Robot is holding Block x
Handempty()	Robot has an empty hand

Table 4.3: Nilsson's Blocks World

The hierarchies I have tested group Clear, OnTable, and Holding together, and leave On and Handempty separate. Thus there are three sets of predicates, each of which can have a particular criticality assignment. I test all 6 possible criticality combinations of these. Table 4.4 details how the propositions are grouped by criticality for the purposes of experimentation. For ease of exposition, I will refer to the first grouping as c, the second as o, and the third as h. In this way, a hierarchy ordering the Clear, OnTable, Holding set as criticality above On above Handempty would be referred to as *coh*.

Group "c"	Group "o"	Group "h"
Clear	On	Handempty
OnTable		
Holding		

Table 4.4: Nilsson Domain Criticality Groupings

The operator set used to move the blocks contains the four operators **Stack**, **UnStack**, **Pickup**, and **PutDown**, as detailed in Figure 4.2.

4.2.3 Sacerdoti's Robot World

Results are presented in [Yang *et al.*, 1991] for a more complex domain, a modified version of a domain described by Sacerdoti in [Sacerdoti, 1974], and again in [Knoblock, 1991]. I give a brief description of this domain here in order to demonstrate how search strategies perform in more complex domains.

In Sacerdoti's robot and box domain, there are a number of rooms, a robot that can travel between these rooms, and a number of boxes located at discrete locations in various rooms. The rooms are connected by doors that can be opened or closed. The robot can carry certain blocks, push certain blocks, open and close doors, and move between rooms. This domain is the most complex of the three on which ABTWEAK has been tested on, and in fact has a much larger operator set than the other domains.

The large set of predicates used in this domain are shown in the Table 4.5. A sample of the operator set is shown in Figure 4.3.

The propositions which define the domain objects or characteristics (such as

Pickup (x)

Preconditions $\{OnTable(x), Clear(x),$
$Handempty()\}$

Effects $\{Holding(x), \neg OnTable(x)$
$\neg Clear(x), \neg Handempty()\}$

Putdown (x)

Preconditions $\{Holding(x)\}$

Effects $\{OnTable(x), Clear(x)$
$Handempy(), \neg Holding(x)\}$

Stack $(x\ y)$

Preconditions $\{Holding(x), Clear(x)\}$

Effects $\{On(xy), Clear(x),$
$Handempty(), \neg Holding(x),$
$\neg Clear(y)\}$

UnStack $(x\ y)$

Preconditions $\{On(xy), Clear(x),$
$Handempty()\}$

Effects $\{Clear(y), \neg Clear(x),$
$\neg Handempty(), Holding(x), \}$
$\neg On(xy)\}$

Figure 4.2: Nilsson's Blocks World Operators

Predicate	Meaning
Armempty()	is the robot holding something?
Holding(object)	object currently in the robot arm
Open(door)	is the door open?
Robot-At(location)	location the robot is at
Object-At(object, location)	location an object is at
Robot-Inroom(room)	room the robot is in
Object-Inroom(object, room)	room the object is in
Is-Location(location)	this entity is of type "location"
Location-Inroom(location, room)	this location is in which room
Is-Door(door,room1,room2,r1-loc,r2-loc)	this door is between what rooms and locs?
Pushable(object)	this object can be pushed
Carriable(object)	this object can be carried

Table 4.5: Robot Domain

IsPeg in Hanoi), and which cannot be satisfied by operator addition are grouped into in the same criticality category. These propositions which be seen as the most "difficult" to achieve in this domain since they are only achievable through one operator, the initial state operator. Moving the robot or objects into rooms is more difficult than simply moving within a room, since the former requires a more complex operator subset than the later. Accordingly, achieving goals such as holding an object, or opening a door require a quite simple operator subset to be added to the plan.

Some examples of operators that are used to achieve the aforementioned goal propositions are shown in Figure 4.3 on page 112.

In addition to the operators shown in Figure 4.3, there are also 8 others: **Pickup-Object**, **Putdown-Object**, **Goto-Room-Location**, **Carry-Object**, **Push-Thru-Door**, **Carry-Thru-Door**, **Open-Door**, and **Close-Door**.

4.3 Explanation of the Figures

The experimental results can be divided into three main categories:

1. Results which demonstrate the usefulness of monotonic properties in restricting search.

2. Results which identify the benefit of applying the LEFT-WEDGE search strategy in place of breadth-first.

3. Results which show how the application of certain goal ordering heuristics can dramatically improve search performance by taking advantage of the tendency of many domains to exhibit subgoal independence.

Push-Object $(object, room, fromLoc, toLoc)$

Preconditions $\{Pushable(object)$
$LocationInRoom(toLoc, room)$
$LocationInRoom(fromLoc, room)$
$ObjectInRoom(object, room)$
$RobotInRoom(room)$
$ObjectAt(object, fromLoc)$
$RobotAt(fromLoc)\}$

Effects $\{\neg RobotAt(fromLoc)$
$\neg ObjectAt(object, fromLoc)$
$RobotAt(toLoc)$
$ObjectAt(object, toLoc)\}$

Go-Thru-Door $(door, fromRoom, toRoom, fromLoc, toLoc)$

Preconditions $\{IsDoor(door, fromRoom, toRoom, fromLoc, toLoc),$
$RobotInRoom(fromRoom),$
$RobotAt(fromLoc),$
$Open(door),$
$Armempty()\}$

Effects $\{\neg RobotAt(fromLoc)$
$\neg RobotInRoom(fromRoom),$
$RobotAt(toLoc),$
$RobotInRoom(toRoom)\}$

Figure 4.3: Sacerdoti's (modified) Blocks World sample operators

Results detailing the first and second categories in the Towers of Hanoi domain are presented in Figures 4.4 through 4.13. This set of tests assumes that the assignment of criticalities for negative and positive values of a given proposition is the same. Figures 4.4 to 4.7 and 4.10 to 4.13 show the number of state expansions required by AbTweak to find a solution to Hanoi-3 as a function of the abstraction hierarchy used. Figures 4.4 to 4.7 show the number of expansions required by breadth-first search (1) without any monotonic property, (2) with P-WMP, and (3) with N-WMP. Figures 4.10 to 4.13 contrast breadth-first and Left-Wedge strategies without using monotonic properties. Figure 4.8 shows the number of nodes expanded as a function of the number of monotonic violations, for both N-WMP and P-WMP. Figure 4.9 compares breadth-first and Left-Wedge with and without monotonic properties.

Figures 4.16 through 4.18 show results obtained in the Hanoi domain with a slightly different criticality assumption. These figures detail a set of tests which assign the negative versions of propositions a higher (more abstract) criticality than the positive version. Figure 4.16, like Figures 4.5 through 4.7, shows how application of the monotonic property affects the number of expansions required to solve Hanoi-3 while varying the criticality of the IsPeg proposition. Figure 4.16 also shows search performance for these problems using breadth-first and Left-Wedge strategies, similar to Figures 4.10 through 4.13. Figure 4.17 describes the expansions for various hierarchies, keeping IsPeg fixed at one criticality, as is the case in Figure 4.4. Figure 4.18, like Figure 4.8, shows the number of nodes expanded as a function of the number of monotonic violations for P-WMP.

Figures 4.14 and 4.15 detail Tweak and AbTweak results in Nilsson's domain when using Stack and Tree goal ordering. Figure 4.14 shows the results of applying the two goal ordering heuristics for problem instances with solutions of various sizes.

Figure 4.15 describes the result of applying the monotonic property to abstract search for the Sussman Anomaly problem.

4.4 The Utility of The Monotonic Properties

We can summarize our experimental results supporting the application of the monotonic property in abstract planning as follows:

1. Using the monotonic property to constrain search is almost always advantageous, for hierarchies that are intuitively good.

   An intuitively good hierarchy solves a difficult problem, such as `OnBig`, first, and the easy parts like `OnSmall` the last. The better performance can be observed from Figures 4.4 to 4.7. For example, in Figure 4.4, breadth-first search using either N-WMP or P-WMP outperforms one without MP pruning in five of six cases: (*ibms ibsm imbs imsb isbm*, excluding *ismb*). This can be explained in two ways. First, using the MP reduces the branching factor of search. Second, when pruning is done, it is unlikely that a branch leading to a solution will be cut off during breadth-first search. The reason for the second justification is that for most problems, it is more straightforward to find a base-level solution from an abstract one, without violating any abstract causal relations.

2. Breadth-first search is more efficient with P-WMP than with N-WMP (see Figures 4.4 to 4.7).

   We anticipated that P-WMP would perform much better than N-WMP, because the former cuts a subtree of the search space off much earlier than the

latter. However, the experiments show that although there is some difference between the two, the improvement is not as great as we expected.

3. Without using MP, the criticality assigned to object-type predicates such as `IsPeg` is crucial in search efficiency (see Figures 4.5 to 4.7, and 4.16). Overall, it seems better to place these types of predicates at the highest level of abstraction. However, MP tends to stabilize the effect of criticalities on search (see Figures 4.5 to 4.7).

4. We can see a general rule emerging from the results in Figure 4.8 and Figure 4.18, that the fewer the number of monotonic violations, the better the performance in search with an abstraction hierarchy. In a sense, MP measures the number of attempts we make at a particular level of abstraction to undo the work done at the previous level. Exceptions to this general rule can occur when operators added into a plan have beneficial sideeffects, or in instances where a simple solution can be found more quickly generating a successor which violates the monotonic property.

5. Figure 4.9 summarizes the results of tests with and without the use of the MP. The first column lists six hierarchies, and the rest the number of states expanded under a particular search strategy.

   From this figure, it is clear that *ibms*, *ibsm* and *imbs* are three hierarchies that demonstrate better performances than the rest. This phenomenon can be easily explained by the fact that, for example, in *ibms*, the abstract solution at `OnBig` level corresponds the *first* abstract solution. However, in *ismb*, the abstract solution refineable, at the `OnSmall` level, to the base-level one actually corresponds to the fourth alternative solution at that level. As a result, the search space with *ibms* has a much smaller branching factor than *ismb*.

Thus, intuitively, *ibms* is a good abstraction hierarchy. Similarly, hierarchies *ibsm* and *imbs* are close to be good ones. On the other hand, the other three are far from optimal. This is because they all solve easier problems (involving the placement of the small disk) first, and solve the harder ones (involving the placement of medium or big disks) the last.

The following conclusions can be drawn from Figure 4.9:

(a) For good abstraction hierarchies, using the MP is clearly better than without using the MP on average cases. For example, the average number of expansions using *ibms*, *ibsm*, and *imbs* under breadth-first and P-WMP is 450. The average for these three hierarchies under breadth-first, and without using MP is 711.

(b) For good abstraction hierarchies, using LEFT-WEDGE strategy and P-WMP is an overall winner. For example, LEFT-WEDGE plus P-WMP for the first three hierarchies gives an average of 222 number of expansions, as opposed to 711 for breadth-first.

(c) However, for bad hierarchies, the use of MP with either breadth-first search or LEFT-WEDGE dramatically decreases search efficiency. For example, for *ismb*, either breadth-first or LEFT-WEDGE plus MP couldn't even finish in 6000 expansions.

Thus, goal-protection in abstract planning improves efficiency only when the hierarchy used is intuitively good.

4.5 Left Wedge Refinement Utility

We can summarize our experimental results supporting the use of our LEFT-WEDGE search strategy in abstract planning as follows:

1. The results of planning using the LEFT-WEDGE approach (Figures 4.9 to 4.13, Figure 4.16 and Figure 4.17) indicate quite a dramatic improvement over breadth-first for certain criticality assignments. In Figure 4.9, this can be observed under LEFT-WEDGE without using MP for hierarchies *isbm*, *ismb*, *ibms* and *ibsm*. However, no improvement, or even a decrease in performance is seen for certain other criticality assignments, notably *imbs* and *imsb*.

   It is also interesting to note that, for hierarchies (*isbm* and *ismb*, Figures 4.12 and 4.13) that perform very poorly with breadth-first, the LEFT-WEDGE strategy achieves good results.

2. LEFT-WEDGE tends to stabilize the effect of criticalities on search (see Figures 4.10, 4.12, 4.13 and 4.16). Although this effect can be easily explained for intuitively good hierarchies (i.e., *bms*), it is still not clear why the stability exists in bad hierarchies such as *sbm* and *smb*.

4.6 Goal Ordering Results

As explained earlier, the manner in which goals are selected when generating plan successors can dramatically affect search performance. Two different approaches have been tested in order to demonstrate this behaviour. The primary goal of this experimentation is to discover a reasonable, efficient, and justifiable domain-independent approach to goal ordering.

The two approaches are known as Stack and Tree. Stack and Tree represent two attempts at a coherent strategy for selection. It is important to note that the "best" goal to select is not something that can be exactly computed, at any cost, since the factors that would determine the "best" selection are not well known. Some of the factors in goal selection that have been observed to affect planning search performance include:

1. Selection of a particular goal determines the branching factor at a particular point in the search space, since different goals are establishable different ways, depending on the domain definition (new operator additions), and the current plan (existing operator establishments).

2. The degree of commitment to certain codesignations that results from a certain goal selection affects future planning. Non-commitment can result in larger branching factors in future goal selections as a result of more possible establishers existing in the plan for unbound propositions than for fully bound ones.

The experimental results for Stack and Tree ordering show the following:

1. Figure 4.14 demonstrates the utility of Stack and Tree for a set of problems in the Nilsson Blocks World domain. As the complexity of the solution increases, the number of expansions increases dramatically when using the Tree goal ordering method.

   The Stack ordering approach experiences a much slower growth curve as complexity increased. This result can be explained by the ability of Stack to take advantage of the level of independence each subgoal in this domain has over previously satisfied subgoals. While Tree pays close attention to "detailed"

interactions by constantly repairing any potential conflicts with previously established subgoals, Stack makes the assumption that these conflicts will "go away" on their own accord as a result of future planning. Subsequently, Stack causes operators to be added to a plan in a very "linear" manner, where some operator A is added to solve a precondition of the goal state, other preconditions of the goal state are ignored while the first precondition of A is selected and satisfied by a new operator B,etc. This "chaining" of operator addition allows a plan to grow quickly, and if sideeffect establishments are rare, Stack works well.

2. Stack has more of an advantage over Tree for problems solved by TWEAK than by ABTWEAK.

   Figure 4.15 shows (for solutions of the Sussman Anomaly) that in ABTWEAK, Stack outperforms Tree ordering. However, the difference seems to be only a constant factor, rather than the dramatic difference for TWEAK seen in Figure4.14. This result can be explained in at least one way. If we consider the difference between abstract planning and non-abstract planning, we see that TWEAK performs all of its planning at a single level. Each operator in the plan essentially exists at the lowest level of abstraction. ABTWEAK plans at various levels, creating different portions of the eventual concrete plan at different abstract levels. As a result of this "splintering" of the creation of a plan, ABTWEAK cannot always add operators in the "chaining" fashion that benefits TWEAK with Stack goal ordering. Essentially, ABTWEAKś abstraction-based approach is diametrically opposed to the "linear" goal addition of Tree in TWEAK, and Tree in ABTWEAK only performs well at each level, not across the whole planning process.

3. Experiments with random goal selection show that either Tree or Stack are reasonable in that they tend to give solutions within a relatively stable number of expansions.

   It has been observed that search performance with random goal selection is very unstable in that performance for simple problems can be very good or extremely bad. Solutions to simple problems are found more efficiently than Stack or Tree in only a small percentage of test cases. These simple problems have only a few goal selections required in the entire planning process, and few conflicts at any point from which to chose a goal. Solutions to more complex problems (such as the Sussman Anomaly) do not seem to be solvable in a reasonable amount of expansions at all. It would appear that as far as goal selection is concerned, no strategy is a bad strategy. These results can be partially explained by the fact that repeated selections of goals with many unbound propositions causes the addition to a plan of more operators that are largely unbound, and this addition increases the number of conflicts in a plan that are themselves largely unbound, and hence the future branching factor of search.

4.7 Comparing Tweak and AbTweak

In order to more fully understand abstraction by precondition elimination as implemented in AbTweak, it is important to compare abstract search and the non-abstract search of Tweak. Some experimental results indicate that abstraction as implemented in AbTweak is not always superior to Tweak.

The following results have been observed in experiments:

1. TWEAK expanded 379 nodes when solving the Hanoi-3 problem, while ABTWEAK without application of the monotonic property performs worse. For example, when using hierarchy *ibms*, ABTWEAK expanded 471 nodes.

2. The addition of P-WMP to ABTWEAK while using the hierarchy *imbs* results in only 149 node expansions, more than twice as efficient as TWEAK.

3. When compared with ABTWEAK using LEFT-WEDGE the difference is more dramatic. With good hierarchies, e.g., *ibms*, the number of nodes expanded is only 57, more than six times more efficient than TWEAK.

4. When taking all 24 hierarchical combinations of the IsPeg, OnBig, OnSmall, and OnMedium predicates in the Hanoi domain, the LEFT-WEDGE hierarchy alone performed best in 12 of the hierarchies, ABTWEAK with P-WMP performed best in 11, and ABTWEAK alone in none. With these combinations, LEFT-WEDGE was worst in 7 cases, ABTWEAK with P-WMP in 1, and ABTWEAK alone in 10.

5. 11 hierarchical combinations were tested separating the negative and positive versions of the On predicates in Hanoi. In these cases, LEFT-WEDGE performed best in 5 cases, ABTWEAK with P-WMP best in 5, and ABTWEAK alone in 1. With these combinations, LEFT-WEDGE was worst in 1 case, ABTWEAK with P-WMP in none, and ABTWEAK alone in 9.

We have seen quite clearly in all of our results that abstract planning does not always outperform non-abstract planning. Hierarchy selection is critical to abstract planning, and the best hierarchy is not always easy to predict. At least one simple and efficient goal selection method seems to better suit non-abstract planning, and in fact constitutes one explanation of the relatively "good" behaviour of TWEAK.

Nonetheless, abstract planning allows for the use of certain properties such as the monotonic property, and search strategies such as LEFT-WEDGE which give the abstract approach of ABTWEAK the capability to dramatically outperform the non-abstract approach of TWEAK.

4.8 Graphical Results

These figures refer to experiments performed using the Towers of Hanoi domain for three pegs.

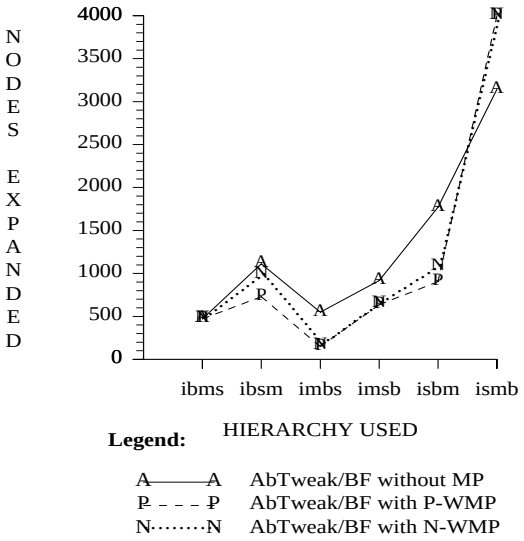


Figure 4.4: Expansions, vary hierarchy

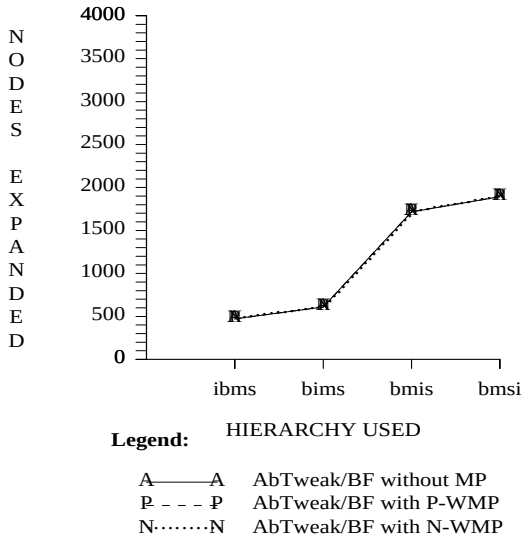


Figure 4.5: BMS expansions, vary IsPeg

Figures 4.14 and 4.15 refer to experiments performed using the Nilsson Blocks World domain.

Figures 4.16, 4.17 and 4.18 refer to experiments performed using the Towers of Hanoi domain with 3 pegs, assigning separate criticality values for the negative and

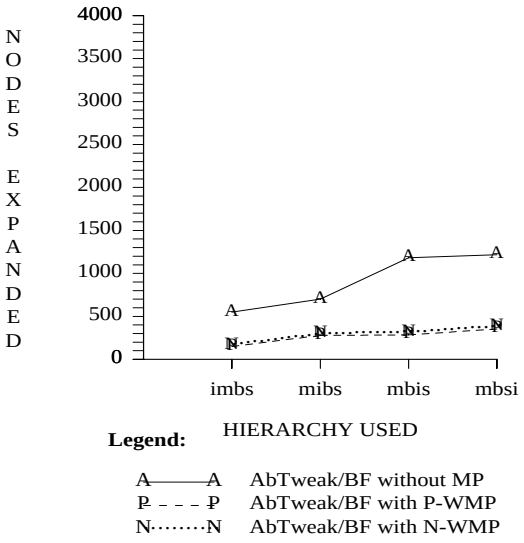


Figure 4.6: MBS expansions, vary IsPeg

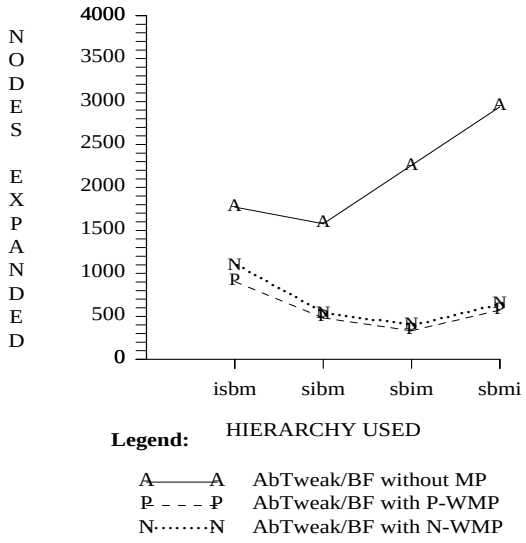


Figure 4.7: SBM expansions, vary Ispeg

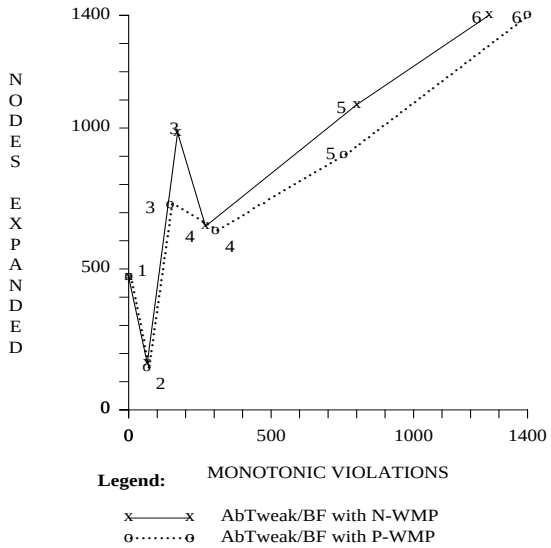


Figure 4.8: BF Expansions : 1-IBMS, 2-IMBS, 3-IBSM, 4-IMSB, 5-ISBM, 6-ISMB

	Breadth	Breadth P-WMSP	Left-Wedge	Left-Wedge P-WMSP
IBMS	471	471	57	57
IBSM	1112	729	828	531
IMBS	550	149	1009	78
IMSB	918	636	5170	2672
ISBM	1771	904	168	5232
ISMB	3142	>6000	963	>6000

Figure 4.9: Summary of BF and LW expansions

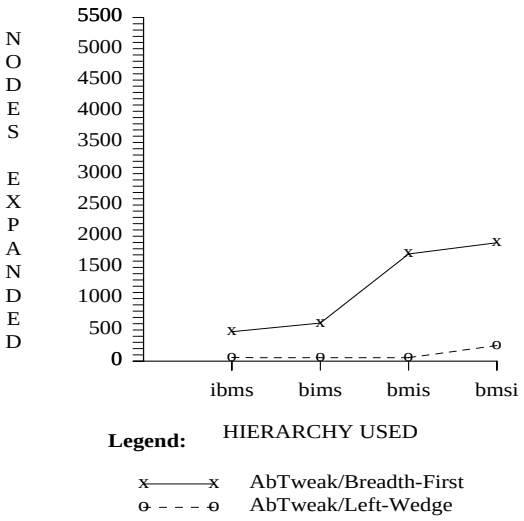


Figure 4.10: BMS expansions: BF,LW

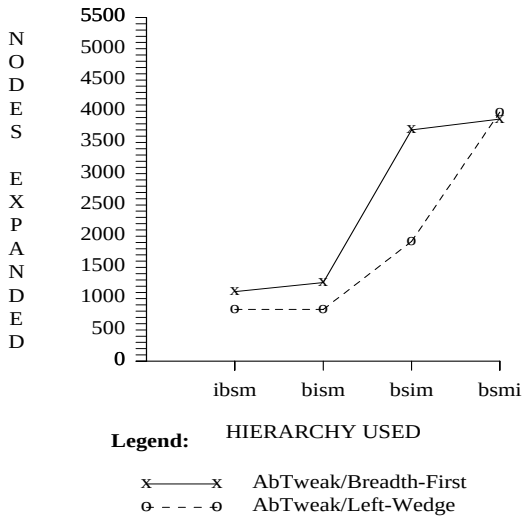


Figure 4.11: BSM expansions: BF,LW

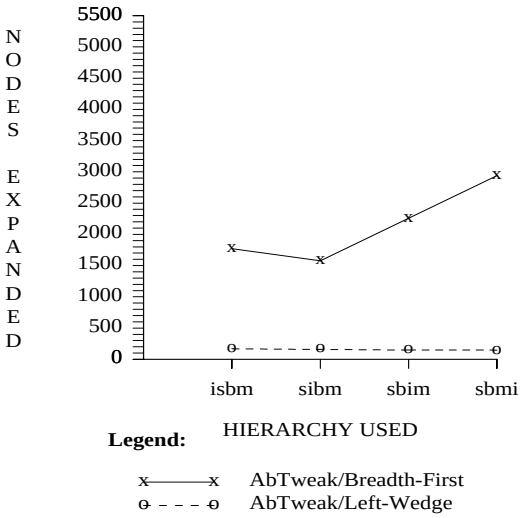


Figure 4.12: SBM expansions: BF, LW

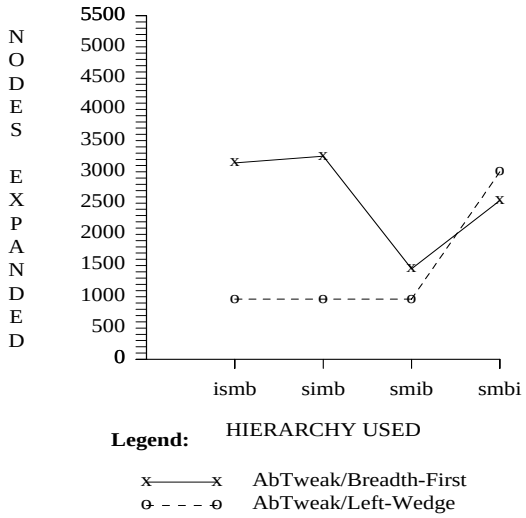


Figure 4.13: SMB expansions: BF, LW

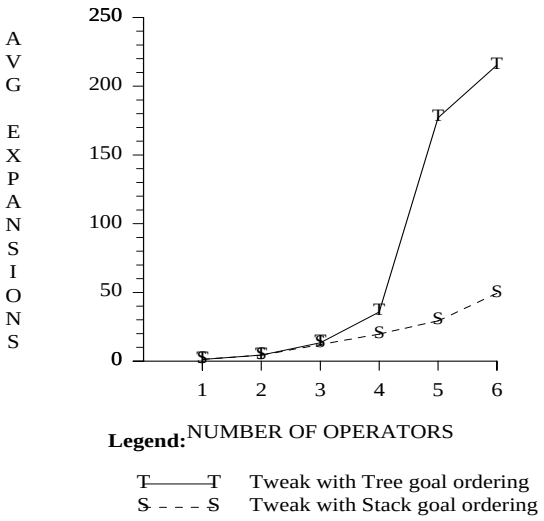


Figure 4.14: Tweak goal ordering

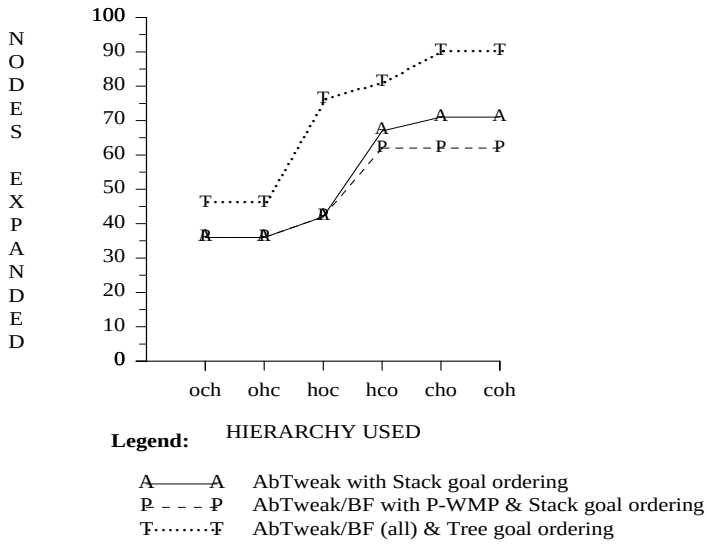


Figure 4.15: ABTWEAK, vary hierarchy

positive versions of propositions. References to criticality orderings in Figures 4.16 through 4.18 are detailed in Table 4.6.

Criticality Ordering	Label
$IB\neg MM\neg SS$	I1, O1
$BI\neg MM\neg SS$	I2
$B\neg MIM\neg SS$	I3
$B\neg MMI\neg SS$	I4
$B\neg MM\neg SIS$	I5
$B\neg MM\neg SSI$	I6
$IB\neg SS\neg MM$	O2
$I\neg MMB\neg SS$	O3
$I\neg MM\neg SSB$	O4
$I\neg SSB\neg MM$	O5
$I\neg SS\neg MMB$	O6

Table 4.6: Hanoi Domain, Positive, Negative Criticality Labels

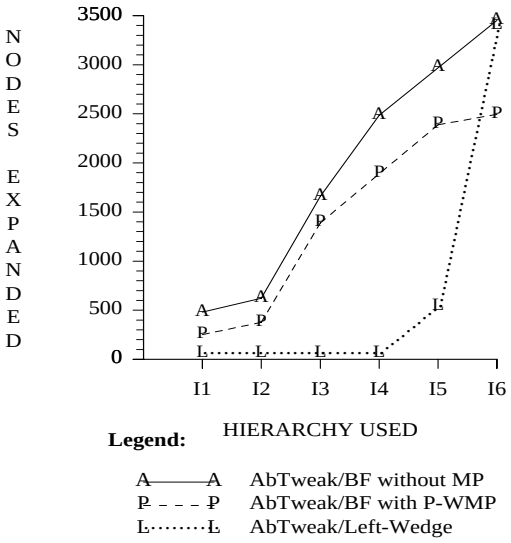


Figure 4.16: Expansions, vary IsPeg

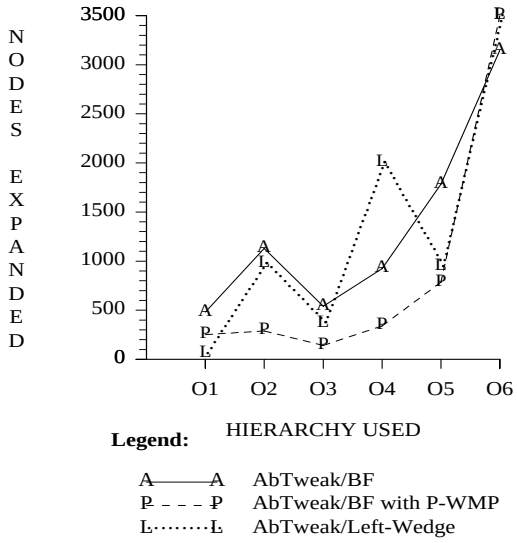


Figure 4.17: Expansions, vary hierarchy

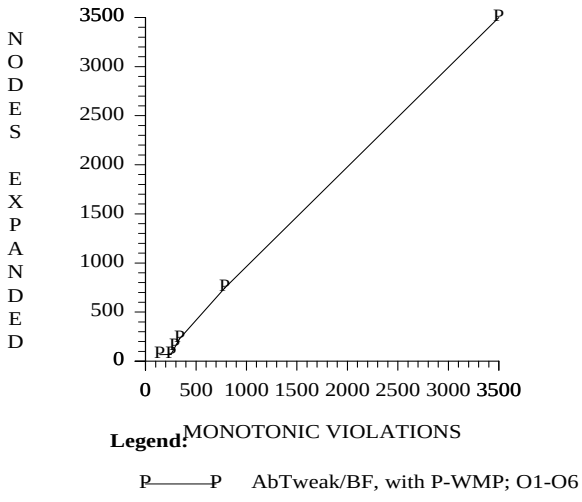


Figure 4.18: BF Expansions & Violations, varying hierarchy

Chapter 5

Summary

5.1 Conclusions from Experiments

The experiences and lessons learned from our experiments can be summarized as follows:

1. The application of the monotonic property is a form of goal protection in abstract planning. However, if one protects too much, as in the case of SMP, completeness is not preserved. The use of the other versions of the WMP , including both P-WMP and N-WMP, in abstract planning with good hierarchies and using a breadth-first search strategy, tend to outperform breadth-first without the application of MP. This is true for most criticality assignments, with only 2 exceptions out of 24 tests. Furthermore, with a slightly domain-dependent implementation of WMP, i.e. P-WMP, AbTweak always outperforms the domain-independent version of WMP, i.e. N-WMP.

2. When abstract planning takes on a depth-first flavor, as seen in the Left-Wedge strategy, the application of MP is useful only when utilizing *good* ab-

straction hierarchies. On the other hand, with *bad* hierarchies, LEFT-WEDGE searches more than with a breadth-first strategy.

3. The number of monotonic violations is an indication of the relative *goodness* of a hierarchy. A hierarchy is generally indicated to be *better* in terms of predicted search performance if it demonstrates fewer monotonic violations when planning under a complete search strategy.

4. Taking a range of hierarchies into account, LEFT-WEDGE performs best the most often of all strategies (17), but also performed worst quite often (8). ABTWEAK with P-WMP performed best next most often (16), and worst only (1). ABTWEAK alone performed best in only (1) case, while performing worst in (19). It would appear that LEFT-WEDGE is potentially the most efficient, but is unstable depending upon hierarchy. ABTWEAK with P-WMP performs best overall almost as often as LEFT-WEDGE and is extremely stable over various hierarchies.

5. A goal ordering approach in non-abstract planning which tends to be casual in terms of protection of existing plan establishments tends to outperform approaches which rigidly try to repair existing establishments.

6. In ABTWEAK, the advantage of a "casual" goal ordering approach is largely restricted to individual abstraction layers, and in fact is not as pronounced as for non-abstract planning.

7. Non-abstract planning is not always outperformed by abstract planning. However, the use of abstraction allows for the application of certain properties which restrict the possible search space, and of certain search strategies which work well with abstraction for well understood hierarchies.

AbTweak represents an approach to classical planning that demonstrates definite performance improvement over Tweak. Application of the Monotonic Property when planning with AbTweak further improved performance. A simple search strategy that attempts to take advantage of the nature of hierarchical plans to some extent dramatically outperforms a traditional breadth-first approach. All of these improvements represent domain-independent heuristics for improving the performance of nonlinear, conjunctive goal planning. Future insights into the nature of plans themselves, planning hierarchies, plan construction, and constraint maintenance will certainly suggest further heuristics which can improve planning performance even further. While the application of new and improved heuristics may not change the fact that planning itself is undecidable, they may yet provide the framework for a classical planning system, efficient and predictable enough for real-world problems in a variety of domains.

5.2 Future Work

1. Control Heuristics

   Criticality assignments to precondition literals represents one type of goal ordering heuristic for nonlinear planning known as abstraction. Experiments show that other factors also affect planning performance, such as the selection of preconditions within each layer of abstraction. While the results of application of each of these heuristics have been examined experimentally, a formal explanation of the relative utility of each approach has not been given. Further investigation into defining hierarchies for nonlinear planners, and for justifying various goal selection strategies would add to the general understanding of nonlinear planning search control.

2. Extended Search Control Strategies

   While AbTweak as described in this paper represents one visualization of an abstract, nonlinear planning control strategy, others surely exist. AbTweak plans in two layers simultaneously in its search for a solution plan, the abstract search control layer which determines which abstract solution to explore, and the individual layer, which determines how to search for more concrete solutions under a given abstract solution. Since AbTweak is only partially committed to a given abstract solution, and is in fact expanding many abstract solution simultaneously, a more parallel search strategy is suggested. A parallel search beneath multiple abstract solutions will not decrease the number of node expansions required to find a goal, however, the overall time could potentially be reduced.

3. "Soup" approach

   One simple approach to improving search performance, suggested by Qiang Yang, is to look at all of the goals that need to be satisfied at one time, create a set of plans such that all combinations of operators that can satisfy these goals are considered. These operators are added with no ordering constraints with respect to one another, and once added, search progresses as before, with successors generated as a result of declobbering, except new operators are not added for the goals that motivated the "soup". While this approach benefits in reducing the depth of a potential solution in the search space by adding more than one operator in a step, the cost in terms of potential plan conflicts is likely to be high.

4. Explanation Closure

The idea of Explanation Closure (EC) [Schubert, in press] is basically a refinement of the "Soup" approach, where the operators added are "motivated" in terms of the initial set of problem goals. In addition, the order of the initial operators is constrained more fully than is the case with the "Soup" approach. A certain amount of inference is performed on the operators, thus driving only "sensible" orderings. This method has the potential to greatly improve search performance, once again by avoiding much of the search space composed of "building" the plan via operator addition, however, the exact cost of performing this explanation inferencing is unclear.

5. Plan Reuse via Macro Operators

Once a plan has been successfully built in any domain, the plan itself can be viewed as a macro-operator which essentially changes the state of the domain from the initial domain conditions (macro-operator preconditions) to the goal conditions (macro-operator effects). Others have investigated the application of macro-operators [Fikes *et al.*, 1972] and plan-reuse [Kambhampati, 1989] for other forms of planners. The addition of macro-operators to the planner's operator library can potentially reduce subsequent search for similar or even identical problems. For example, in the Tower of Hanoi domain, progressively more difficult hanoi problems cause search using a single operator set to grow very fast. In fact, while a solution for 2 and 3 rings may be found quite efficiently, a solution for 4 or more rings can not be found in the space limitations of my own implementation. However, if a solution for the 2 ring problem is made into a macro-operator, the "improved" planner can solve the 3 ring problem with the macro-operator and only 2 additional operators. In fact, it is well-known that the hanoi problem in general has problems that can be solved for N rings using the solution for N-1 rings plus two other operators.

There are difficulties with this approach, however. One such problem is determining which macro-operators should remain in the operator set, and which should not. The operator set size affects the branching factor of the search directly, and so must not be allowed to grow indefinitely. Heuristic approaches to solving this problem could include preferring frequently used macro-operators, or other, more domain-dependent methods.

6. Problem Decomposition

   If we extend our assumption of subgoal independence somewhat, so that we "pretend" that it is possible to solve each initial goal independently, we could simply create one plan for each of the high level goals, and attempt to merge these solutions by declobbering via MTC goal achievement. Once again, the concept is to skip the initial operator-addition portions of the search space. While this method is attractive intuitively, especially in domains with goals that are very independent (such as where a domain possess many separate objects, each of which needs to be affected independently), a very real problem exists in keeping the search complete. Many solution potentially exist for each subgoal. In fact, an infinite number of solution can possibly exist for a particular goal. For example, moving a ring between two pegs allows the possibility that the ring can have any number of intermediate locations.

   A planner described by Yang in [Yang, 1991] called WATPLAN works in a similar manner. The problem of infinite solutions is handled heuristically, where each of M subgoals is actually satisfied some arbitrary number (N) ways, and then the M subgoals * N solutions are merged, giving a set of plans. Search through this space of plans is completed via declobbering of the existing conflicts as outlined by MTC and by the conflict algebra mentioned in [Yang, 1990a]. In addition, the search space is reduced using known constraint

satisfaction techniques [Dechter and Pearl, 1987, Dechter, 1990]. A complete discussion of this approach to nonlinear planning can be found in [Yang, 1991].

Bibliography

[Agre and Chapman, 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the 6th AAAI*, pages 268–272, 1987.

[Carberry, 1990] Sandra Carberry. A new look at plan recognition in natural language dialogue. Technical Report 90-08, University of Delaware, 1990.

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

[Christensen, 1990] Jens Christensen. Pablo: A hierarchical planner that generates its own abstraction hieararchies. *Submitted for Publication, Stanford University*, 1990.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34, 1987.

[Dechter, 1990] R. Dechter. From local to global consistency. In *Eighth Canadian Conference on Artificial Intelligence*, 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.

[Fikes *et al.*, 1972] Richard Fikes, Peter Hart, and Nils Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Hart *et al.*, 1968] P. Hart, N. Nilsson, and B. Rapahael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hertzberg and Horz, 1989] Hertzberg and Horz. Towards a theory of conflict detection and resolution in nonlinear plans. In *Proceedings of the 11th IJCAI*, pages 937–942, Detroit, Michigan, 1989.

[Horowitz and Sahni, 1984] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algortihms.* Computer Science Press, Rockville, Maryland, 1984.

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

[Kautz, 1987] Henry Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York, 1987.

[Knoblock, 1989] Craig A. Knoblock. A theory of abstraction for hierarchical planning. In Paul Benjamin, editor, *Proceedings of the Workshop on Change of Representation and Inductive Bias*, Boston, MA, 1989. Kluwer.

[Knoblock, 1990] Craig A. Knoblock. Learning effective abstraction hierarchies. In *Proceedings of Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Knoblock, 1991] Craig A. Knoblock. Automatically generating abstractions for problem solving. Technical Report CMU-CS-91-120, Carnegie Mellon University, 1991.

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

[McDermott, 1978] D.V. McDermott. Planning and action. *Cognitive Science*, 2, 1978.

[Minton, 1989] S. Minton. Explanation-based learning: A problem-solving perspective. Technical Report CMU-CS-89-103, Carnegie Mellon University, 1989.

[Nau, 1987] Dana Nau. Hierarchical abstraction for process planning. In *Proceedings of Second International Conference in Applications of Artificial Intelligence in Engineering*, 1987.

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

[Plaisted, 1981] D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:46–108, 1981.

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

[Sacerdoti, 1975] E. D. Sacerdoti. The nonlinear nature of plans. *Advance Papers, IJCAI*, 1:206–214, 1975.

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

[Schubert, in press] L. Schubert. Monotonic solution of the frame problem in the situation calculus. In Kyburg, Loui, and Carlson, editors, *Knowledge Representation and Defeasible Reasoning*. Kluwer, in press.

[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. Technical Memo AI-TR-287, M.I.T. AI Lab, 1973.

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

[Tenenberg, 1988] Josh Tenenberg. *Abstraction in Planning*. PhD thesis, University of Rochester, Dept. of Computer Science, Rochester, NY, May 1988.

[Wilkins and Robinson, 1981] D.E. Wilkins and A.E. Robinson. An interactive planning system. SRI Technical Note 245, Stanford Research Insitute, 1981.

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

[Yang and Tenenberg, 1990] Qiang Yang and Josh Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the 8th AAAI*, Boston, MA, August 1990.

[Yang *et al.*, 1991] Qiang Yang, Josh Tenenberg, and Steven Woods. Abtweak: Abstracting a nonlinear, least commitment planner. Submitted for publication, 1991.

[Yang, 1990a] Qiang Yang. An algebraic approach to conflict resolution in planning. In *Proceedings of the 8th AAAI*, Boston, MA, August 1990.

[Yang, 1990b] Qiang Yang. Reasoning about conflicts in least-commitment planning. Technical Report CS-90-23, University of Waterloo, 1990.

[Yang, 1991] Qiang Yang. Understanding the essence of nonlinear, least-commitment planning. Submitted for Publication, 1991.