SEARCHING FOR PLANS USING A HIERARCHY OF LEARNED MACROS AND SELECTIVE REUSE

DISSERTATION
Douglas Earl Dyer
Captain, USAF

AFIT/DS/ENG/94J-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/94J-01

SEARCHING FOR PLANS USING A HIERARCHY OF LEARNED MACROS AND SELECTIVE REUSE

DISSERTATION

Douglas Earl Dyer
Captain, USAF

AFIT/DS/ENG/94J-01

AFIT/DS/ENG/94J-01

SEARCHING FOR PLANS USING A HIERARCHY OF LEARNED MACROS AND SELECTIVE REUSE

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Douglas Earl Dyer, B.S.Ch.E., B.S.E.E., M.S.C.S

Captain, USAF

June, 1994

AFIT/DS/ENG/94J-01

SEARCHING FOR PLANS USING A HIERARCHY OF LEARNED MACROS AND SELECTIVE REUSE

Douglas Earl Dyer, B.S.Ch.E., B.S.E.E., M.S.C.S

Captain, USAF

Approved:

George H. Gunsch	25 May 94
Eugene Santos Jr	24 May 94
John Bay	20 May 94
Mark A. Roth	25 May 94

J. S. Przemieniecki 6 June 1994

J. S. PRZEMIENIECKI
Institute Senior Dean and
Scientific Advisor

Acknowledgements

Given the difficulty of any doctoral program, many people must be amazed that anyone would ever pursue a Ph.D. Indeed, it takes a warped system of values to think that doctoral research will lead to happiness. The hard work and numerous setbacks in an increasingly narrow and deep domain are only occasionally offset by some really new and interesting knowledge. Ah, but what a thing this knowledge is, at least if you have the warped values of a researcher! I would like to sincerely thank my friends Dr Nort Fowler and Dr Steve Cross for sufficiently warping my value system during an intense and fruitful search for knowledge at the Rome Laboratory.

I thank my advisor and friend, Dr Gregg Gunsch, whose gentle guidance and firm support have always helped and encouraged me greatly. Dr Gunsch expanded my vision in many ways. Without complaint, he gave me many hours and many ideas; I enjoyed our time together immensely. I am also quite grateful to Dr Frank Brown, whose patience and grace I find admirable. Dr Brown challenged and inspired me to do some of my best work at AFIT. My other committee members, Dr Mark Roth, Dr Gene Santos, and Dr John Borsi, deserve special acknowledgement for their helpful insights and recommendations. I would like to thank Dr Rao Kambhampati and Dr Steve Minton for answering questions at various points in my research. Alex Kilpatrick and Mark Gerken have my gratitude for listening to my ideas, as does Ken Fielding for that and so many other things.

I owe a great debt to my wife, Susan and my daughters, Sarah and Kristen. They never failed to support me, and I would not have succeeded without their support. I am also indebted to the United States Air Force for its investment in my education and for the life my family and I enjoy.

Douglas Earl Dyer

Table of Contents

List of Figures

Abstract

This research presents a new approach to improving the performance of a macro planner: selective reuse. In macro planning, reuse can result in *poorer* performance than when planning with only primitive operators, a phenomenon that has been called the *utility* problem. The utility problem arises because the benefits of reuse are outweighed by the cost of retrieving a macro to reuse and the cost of searching through the larger search space caused by considering reuse candidates. Selective reuse contains the expansion of the search space by limiting the number of reuse candidates considered and limits the search required by considering only those reuse candidates that entail no additional work. Previously, performance improvement in a macro planner has been possible only by selective learning. Unlike selective learning, selective reuse never overlooks a learning opportunity that might have value in future problem solving. This research developed a polynomial-order retrieval method which reduces the cost of retrieving a reuse candidate likely to save search. The retrieval method requires time that varies, at worst, linearly with the increasing size of the library of macros. A macro planner (HINGE) was implemented to explore selective reuse. Because the HINGE planning model can insert operators anywhere in a plan and does not introduce arbitrary ordering constraints, HINGE can solve more problems efficiently through reuse than classical macro planning models. To enhance the probability of beneficial reuse, decreasingly-powerful reuse candidates are iteratively sought using a unique, hierarchically-structured search method which is supported by several problem decomposition techniques and by the planning model used. Performance results with HINGE are consistent with the idea that selective reuse can contain the utility problem without the need for selective learning.

SEARCHING FOR PLANS USING A HIERARCHY OF LEARNED MACROS AND SELECTIVE REUSE

I. Introduction

A plan is a set of actions taken in some order to achieve desired goals, and planning is the process of finding a plan. Planning is thought before action, decision before execution. Planning is a fundamental, essential activity of all intelligent entities because without it, goals may not be achievable or resources may be wasted. As intelligent entities, humans are able to plan quite well, although humans often make mistakes, especially as the complexity of the plan increases.

Under certain conditions, it is possible to write a computer program that plans without making mistakes. Figure 1 shows a block diagram of a basic planning system. The inputs to the planning system define a planning problem and consist of the initial state, the desired goals, and a set of operators for changing state by virtue of the actions they represent. A planning system works by searching for an operator sequence which transforms the initial state into a state in which the problem goals are true. This operator sequence constitutes a plan.[1]

It is desirable for a planning system to always find a plan whenever one exists. To guarantee this property, however, exhaustive search is normally required because chosen search directions may prove to be fruitless. Because exhaustive search algorithms require time that is an exponential function of the length of the path searched, planning systems are often slow when a long plan is required.

[1] An operator *sequence* is not necessarily required. Chapter II describes a model of planning that allows a plan to consist of a *set* of operators and a set of ordering constraints between operators.

Figure 1. A simple planning system that searches using provided operators.

One method of making planning systems plan faster[2] is to use more powerful operators, ones that achieve more goals. Plans made using these operators are shorter (and thus can be found more quickly) because fewer operators are required to achieve the same number of goals. Using powerful operators to speed planning is an appealing approach because it is possible to learn more powerful operators from plans derived during problem solving. Operators learned from plans are called macro operators (or macros); they are defined by the interface characteristics of the plan they represent and they are annotated with the plan so that they can be interpreted by the user of a planning system.

Figure 2 is a block diagram for a planning system that learns macros and reuses them to improve its planning speed. In Figure 2, the original operators supplied as part of the planning problem are called "primitive operators" to distinguish them from the learned macro operators. Planning systems that reuse the plans that macro operators represent are called "macro planners" and the macros considered by the planning system are called "reuse candidates."

[2]In this research, performance refers to the speed with which a planning system can find a plan, rather than the quality of the plan produced.

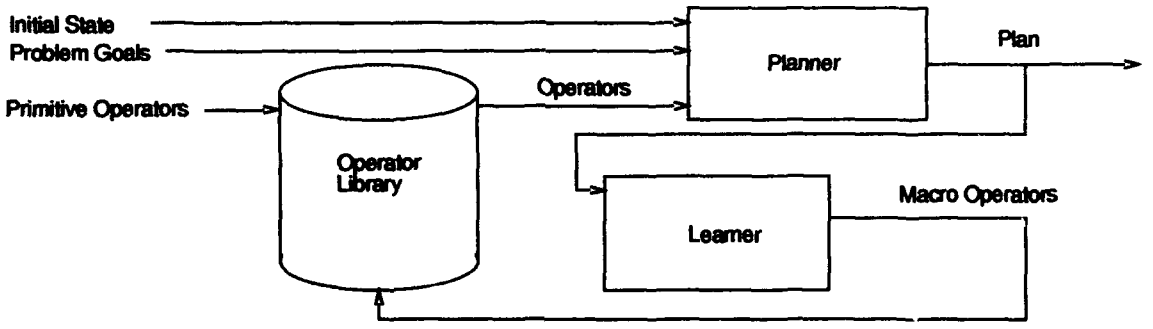


Figure 2. A planning system with a learning element.

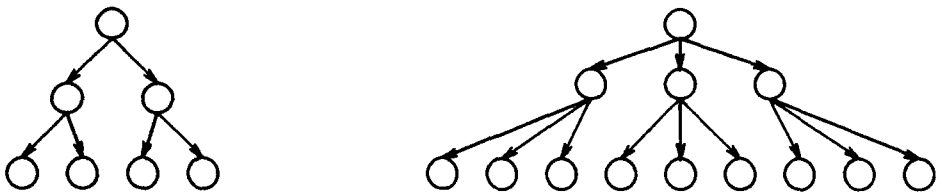


Figure 3. Expansion of a search space caused by considering one additional choice per decision.

In contrast, simple planners like the one shown in Figure 1 are called "generative planners" because they generate a solution using only the primitive operators supplied.

1.1 The Utility Problem

An empirical observation from using macro planning systems such as the one shown in Figure 2 is that for some problems, performance is *worse* than if a simpler planning system like the one shown Figure 1 had been used. This phenomenon has been termed the *utility* problem and there are three factors that cause it. First, there is overhead associated with retrieving a good operator from the library. Second, when the planning system considers macro operators in addition to the primitive operators, the larger number the choices available expands the search space, which thus requires more time to search exhaustively. For example, Figure 3 shows that the additional consideration of 1 operator at each decision point in search increases the size of the search space and the number of possible paths in it dramatically. Third, inserting a macro operator into a plan can impact the amount of future planning necessary more than inserting a primitive operator. A macro operator represents a large step in the search space; if it is a step in the wrong direction, then the planning system has a large amount of future work to do to get back onto a path leading to a goal state. These three factors are costs or potential costs incurred when a planning system reuses learned plans (represented as macro operators). Performance improvement results only if the benefit of using the more powerful macro operators outweighs the costs associated with the utility problem as identified above.

Previous research has addressed the utility problem by limiting the overhead of retrieving a good macro operator or by strictly limiting the expansion of the search space caused by the larger number of operators available for consideration. To limit the overhead of retrieving a good macro operator, one approach is to limit the size of the library by limiting the number of plans that are learned.[3] For example, Minton writes:

> "Searching through the space of stored plans to find one that is best-suited to the current problem may be as expensive as searching through the original search space. This leads to the following paradox: as the system gains experience it gradually becomes swamped by the knowledge it has acquired. In some cases, performance can eventually degrade so dramatically that the system operates even more poorly than a non-learning system. ... To avoid being swamped by too many macro-operators, a problem-solver can endeavor to retain only those macro-operators that are most useful." (Minton, 1985:651)

An important drawback of selective learning is some plan that would be useful on a future problem may not be learned. To limit the expansion of the search space, some planning systems consider only one macro operator for insertion into the plan it is developing. Such planning systems, called case-based planners, retrieve a macro operator which represents a plan that solved a similar problem and modify the operator in some way to make it solve the current problem. An important disadvantage of reusing only one plan is that the planning system can only solve problems that are similar to problems it has previously encountered.

1.2 New Ideas

This research focuses on new techniques for limiting the impact of the utility problem. The utility problem arises not from costs alone, but because costs outweigh the benefits of plan reuse. Therefore, I have treated the utility problem holistically by combining methods that encourage successful reuse with methods for limiting the costs associated with reuse. In particular, I have designed a planning model that explicitly represents features of desirable reuse candidates and improves the flexibility of inserting a macro operator into a developing

[3]Limiting the number of macro operators available also indirectly limits the expansion of the search space because fewer alternatives exist at choice points in the space searched, but this is not the primary motivation for selective learning.

plan. Most previous plan-reusing planning systems have been based on a planning model that limits the insertion of a macro operators and thus precludes reuse in some instances. Also, to encourage successful reuse, I have developed a unique search control strategy that provides multiple opportunities for reuse through problem decomposition. Decomposition controls the search for a reuse candidate and results in generative planning like that implied by Figure 1 when no acceptable reuse candidate is found.

To control the costs associated with the utility problem, I use an efficient, info n-rich retrieval method which limits the retrieval overhead without the need for selective le ng. To limit the search space expansion and search itself, my approach requires that only one reuse candidate can be considered to solve a problem or sub-problem and that reuse candidate must be on a solution path and must entail no future work. I call this method *selective reuse*, and it is supported by the ability of my retrieval method to find very specific reuse candidates. Previous approaches have encouraged retrieval of a reuse candidate that is likely to be on a solution path or likely to entail no future work. A logical extension of these approaches, selective reuse contains the utility problem better than other methods.[4] Selective reuse depends on having complete information about a problem. When a problem is specified incompletely, selective reuse must be relaxed to allow reuse. The relaxed form of selective reuse considers only a few reuse candidates that are likely to entail no future work. Although the relaxed form of selective reuse is more susceptible to the utility problem than strict selective reuse, it contains the utility problem better than any other reuse policy, given the amount of information known about the problem.

To explore approaches to containing the utility problem, I designed and implemented a planning system, HINGE, based on the planning model, search control, retrieval method, and policy of selective reuse I developed. (Chapter III, IV). By virtue of these attributes, HINGE controls the utility problem extremely well.

The specific contributions made by this research are:

[4] Selective reuse often results in less reuse and thus, sometimes, worse overall performance than other methods, but the aim of this research is to find techniques of reducing the utility problem.

1. *A model of planning and reuse that explicitly represents features of desirable macro operators and supports more flexible insertion of macro operators into the developing plan.* I have developed a planning model which represents distinguished subsets of the outstanding goals as abstract operators in a plan. These subsets of goals are used as an index for finding macro operators which, when retrieved, may be inserted anywhere in the developing plan. In contrast with other planning models, these features potentially improve search control and allow reuse that would otherwise be precluded. (Sections 2.4, 3.2, 3.9).

2. *A unique hierarchically-structured search control mechanism which encourages sub-problem learning and improves opportunities for reuse while ensuring that a plan will be found when one exists.* Unlike other planning systems, HINGE structures its goal-satisfaction search hierarchically in terms of the number of goals; this structure promotes both learning and reuse of generalized plans or sub-plans. During planning, HINGE's hierarchically-structured search control prefers operators that solve larger portions of the problem before other operators. HINGE can also decompose a set of goals that cannot be solved simultaneously. HINGE does not differentiate between macro operators learned from previous planning episodes and the primitive operators that are supplied as part of the planning problem description. The effect of these choices during planning is to facilitate opportunistic reuse of whole-problem solutions, flexible insertion of subproblem solutions when no generalized plan solves the whole problem, and graceful degradation to planning using only primitive operators when reuse is not possible. If reuse fails, HINGE solves the problem using only primitive operators and a complete search strategy; HINGE always finds a solution to a planning problem if a solution exists. When learning, all sub-parts of the hierarchical search tree corresponding to sub-problems that can benefit from reuse are learned by HINGE's learning component. (Sections 3.4, 3.6).

3. *A polynomial-order method for retrieving appropriate (or probably-appropriate) macros from a library.* An important result of this research is identification of a polynomial

method for retrieving macros that are guaranteed to be on a path to a solution and to entail no future planning work. To find such macros, the retrieval method requires complete information which generally exists only at the beginning of a planning problem. However, even if the problem must be decomposed, information often exists to find macros that are at least likely, if not guaranteed, to be on a path to a solution. A related result is that this same retrieval method requires, in the worst case, only linear-order time in the size of the library and is not a function of the size of the macros in the library. (Section 4.4.3).

4. *A policy of selective reuse which limits the number of macros considered and considers only those macros that are on a solution path and entail no future work.* This specific search control policy is designed to improve reuse performance by strictly limiting both the size of the search space and the search required in it. Limiting the number of reuse candidates considered limits the expansion of the search space caused by considering any operators in addition to primitive operators. If macros retrieved are guaranteed to be on a solution path, then it is not necessary to consider more than one of them. Moreover, if macros are on a solution path, then no backtracking is required, and if macros do not require the planner to do any future work on their behalf, then no search is required to make sure they are applicable. In contrast, if a macro requires the planner to achieve additional goals, then the amount of additional search (planning work) required in the future is unknown and unconstrained. Without some external source of knowledge, the impact of inserting a macro into a plan can only be quantified when no additional work is required (and then the quantity is zero). Selective reuse is therefore the only domain-independent method of limiting the future work entailed by inserting a macro operator. For domain-independent planning, any other reuse policy re-introduces the utility problem by expanding the search space and requiring additional search. Selective reuse depends on having a complete specification of the planning problem. If the planning problem is not completely specified, as it is not when solving a sub-problem as if it were independent of other parts of the problem, then a

relaxed form of selective reuse often gives good results in HINGE. The relaxed form of selective reuse considers only those macros that are *likely* to be on a solution path and *likely* to entail no future work by the planner. Both forms of selective reuse contain the utility problem better than alternative policies. (Sections 4.3, 4.5).

HINGE is the first macro planner that contains the utility problem by selective reuse, rather than selective learning. Furthermore, HINGE is the first planner that is specifically designed to increase the opportunities for reuse through hierarchical problem decomposition. Finally, HINGE's retrieval method lowers the cost of reuse, while its planning model facilitates search control and increases the probability of effective reuse.

1.3 Dissertation Organization

This dissertation is organized into six main chapters. The following chapter presents background material that serves as a foundation for this research. The general topics of planning, macro learning, reuse, and the utility problem are described. To show the relationship of this research with other work, four other planning systems are outlined and compared to HINGE in terms of the types of problems solved, the flexibility of reuse, and management of costs associated with the utility problem. Chapter III describes the HINGE planner and the features of its model that promote reuse. HINGE's hierarchically-structured search method is described as providing multiple opportunities for reuse, trying the best candidates first. HINGE's ability to flexibly reuse solutions is reiterated and shown to stem from its plan-space planning model. Chapter IV analyzes the utility problem and defines two methods to combat it: a polynomial retrieval method and the policy of selective reuse for macros. An example is presented to support the claim that, under restrictive conditions, it is possible to build a polynomial-order macro retrieval procedure that retrieves only macros that are guaranteed to be on a solution path. Chapter IV also shows, by example, how a non-selective policy on insertion of reuse candidates expands the search space and requires additional search, characteristic symptoms of the utility problem. Chapter V describes empirical results which are consistent with claims made about the developed retrieval method and selective reuse. The

final chapter of the document provides conclusions, reviews specific research contributions, and suggests promising avenues for future work.

II. Background

2.1 Introduction

Automated planning systems can be applied to solve many real-world problems, but their inefficiency prevents their widespread use. One method of improving efficiency is to avoid search by reusing previously derived plans as much as possible. This strategy does not always work, however: an observation which defines the utility problem. To avoid the utility problem, some mechanism must be employed to promote plan reuse while limiting search. To encourage plan reuse, it is useful to identify and retrieve good reuse candidates, to be flexible when inserting a selected candidate, and to provide additional opportunities for reuse if initial efforts fail. This research improves upon previous efforts in these areas. To limit search without outside information to guide search, the number of reuse candidates considered must be limited somehow. Previous methods for limiting search have included selective learning of plans for macro planners and single-plan reuse with modification for case-based planners. The restriction of reusing a single plan in case-based planning is an example of a more general method for controlling the expansion of the search space: selective reuse. Chapters III and IV describe a macro planning system (HINGE) which also contains the search space expansion by selective reuse. This chapter presents the the background information that facilitates discussions in Chapter III and IV.

As an overview, this chapter begins by defining terms required to describe HINGE as well as other planners and then uses these terms to discuss state- and plan-space search and related issues. Section 2.2.4 describes a non-deterministic, plan-space-searching planning model which HINGE's planning model extends. Three formal properties that characterize HINGE and some other planners are defined. Section 2.3 describes learning, reuse, and the utility problem. Finally Section 2.4 presents four other planners that reuse previous solutions and compares HINGE with them in terms of their management of the utility problem.

2.2 *Planning Terms and Models*

In Chapter I, planning systems were characterized as algorithms that search for a sequence of operators that transform a given initial state into a state in which problem goals are true. This section formalizes that description and extends it to eliminate unnecessary ordering constraints on operators.

2.2.1 Representing Domain Objects and Operators. Many problems encountered daily in business, industry, and the military can be cast as planning problems and solved using planning systems. To do so, the problem must first be mapped into a formal representation assumed by the planning system and the output from the planning system must be interpreted in terms that make sense to the user. In this research, it is assumed that domain objects and operators are represented by propositions. Although alternative representations exist, object attributes, relationships between objects, and changes in state can most easily be represented by propositions. The syntax and semantics of propositions can be found in (Gries, 1985).

Operators are identified with names. It is assumed that there is an mapping between names of operators and actions so that the output of a planning system may be interpreted by the user.

2.2.2 Definitions. To specify a planning problem, definitions are required for the fundamental concepts of state and problem goal and for the meaning of change caused by action.

> **Definition: State** A state is a unique set of non-negated propositions that describes all objects of interest in a domain in terms of pertinent relationships between objects and attribute values for each object.

> **Definition: Problem Goal** In the context of a planning problem, a problem goal is a proposition that indicates a desired value for a particular relationship between objects or an attribute value for an object.

Because states are described by unique sets of propositions, each state is distinguishable from all other states. A set of problem goals may be included in multiple states; any state that includes all the problem goals is referred to as a "goal state."

An operator is a data structure used by the planner that has the interpretation of an action. An operator captures the important attributes of action: the changes in state caused by the action (effects) and the propositions that must be true before the action is feasible (preconditions). Applying an operator has the same implications as performing an action.

> **Definition: Operator** An operator consists of an operator name, a set of preconditions, and a set of effects. Each precondition and effect is a proposition which may be negated or non-negated.

> **Definition: Applying an Operator** An operator may be applied in any state which includes all of its preconditions. When applied in a *State_1*, an operator causes a transition to a new state, *State_2*, which is found by deleting all negated operator effects from *State_1* and adding all non-negated operator effects. More generally, an operator may be applied whenever its preconditions can be shown to hold.

To save storage space, practical planning systems take operator schemas, rather than operators, as inputs. An operator schema includes variables in its name and as operands in its preconditions or effects. For example, consider an operator which includes the proposition *SOME-RELATIONSHIP(A, B, C)* in either its preconditions or effects. Rather than require n^3 operators that are necessary to represent a domain of n objects such as *A*, *B*, and *C*, a planning system that takes operator schemas as inputs requires just one operator schema with a predicate of the form *SOME-RELATIONSHIP(?-X, ?-Y, ?-Z)*, where *?-X*, *?-Y*, and *?-Z* are variables. Such a planning system finds operators by instantiating all variables in useful operator schemas, unifying predicates in each operator schema with propositions in a state or a set of goals. To guide the unification process, an operator schema must also specify constraints on the values of variables. These constraints are used to keep different variables from unifying with the same constant (variable separation) or to represent domain theory required to define the operator schema.

Definition: Operator Schema An operator schema consists of an operator name which possibly contains variables; a set of preconditions, each of which is a negated or non-negated predicate; a set of effects, each of which is either a negated or non-negated predicate; and a set of constraints on the values for variables. An operator schema must contain at least one variable as an operand to a precondition or effect.

A plan is often thought of as a *sequence* of operators, but a total ordering on all operators is not always required.

Definition: Plan A plan is a set of operators and a set of ordering constraints between operators, where an ordering constraint is denoted by $O1 \prec O2$, meaning that the operator $O1$ necessarily precedes the operator $O2$ in the plan.

Note that the definition above does not require a plan to be able to solve a problem or even to be executable from an initial state; instead, a plan is the current product of a planning system.

If a planning algorithm searches in a space of states, then it is easy to determine if an operator can be applied by examining the current state. However, as discussed in Section 2.2.3, searching in a space of states with a simple search strategy is inherently less flexible than searching in a space of plans. When searching in a space of plans, some data structure is required to ensure that operators can be applied. For planners that search in a space of plans, two important concepts useful in this regard are establishment and clobberer. An establishment is a data structure used show that preconditions hold. A clobberer is an operator that "threatens" or possibly invalidates an establishment.

Definition: Establishment With respect to a particular plan, an establishment is a triple $< E, p, C >$ where p is a proposition, E is an operator that includes p in its effects, C is an operator that includes p in its preconditions, and the set of ordering constraints in the plan includes $E \prec C$.

Definition: Clobberer With respect to a particular plan and an establishment $< E, p, C >$, a clobberer is an operator $Clob$ whose effects include the negation of p and neither $Clob \prec E$ nor $C \prec Clob$ are in the ordering constraints of the plan.

If an establishment $< E, p, C >$ exists, then the operator E is called an establisher, while the operator C is referred to as the consumer. E is said to establish p. Finding an establishment for

a particular precondition is called establishing the precondition. Furthermore, if a clobberer $Clob$ exists with respect to a plan and an establishment $< E,\ p,\ C >$, then $Clob$ is said to clobber the establishment.

Chapman's modal truth criterion defines a method for determining if a given proposition is true in a given state or in the context of a particular plan (Chapman, 1987:340). The definitions below captures the essence of the modal truth criterion and are useful for searching in a space of plans.

> **Definition: Operator Applicability** An operator is applicable in a plan if all of its preconditions are true in the state in which the operator is applied. For searching in a state of plans, an operator C is applicable if every precondition of C has an establishment and there is no clobberer of the establishment in the plan.

> **Definition: Plan Applicability** A plan is applicable if every operator in the plan is applicable.

The input to a planning system is a planning problem which includes an initial state, a set of problem goals, and a set of primitive operators or primitive operator schemas. For searching in a space of plans, it is convenient to represent the initial state and set of problem goals as an initial plan. A plan that solves a planning problem is called a solution and can be defined in terms of applicability and the initial plan.

> **Definition: Planning Problem** A planning problem is a 3-tuple $< I, G, O >$, where I is a set of propositions that are all true initially, G is a set of propositions that must be true in the final state, and O is a set of operators or operator schemas. I is called the initial state. Elements of G are called problem goals. Elements of O are called primitive operators or primitive operator schemas to distinguish them from macro operators or macro operator schemas that are learned from plans during the course of problem solving.

> **Definition: Initial Plan** An initial plan of a particular planning problem is a plan that consists of two operators, *init* and *final*, and an ordering constraint $init \prec final$; *init* has no preconditions, and its effects are equal to I, while *final* has preconditions equal to G and has no effects. All other operators in the plan are constrained to come after *init* and before *final*.

Definition: Solution A plan is a solution of a particular planning problem if it is applicable and includes the initial plan of the planning problem.

For a given set of operators, there is a one-to-one correspondence between a planning problem and its initial plan. A solution always includes *init* and *final*; these operators should be eliminated when the solution is interpreted.

Once derived, a solution can be converted into a macro operator as described in Sections 2.3.1 and 3.8.1.

Definition: Macro operator A macro operator (or macro) is an operator that represents a solution to a planning problem and is annotated with the solution so that the macro operator may be interpreted by the user of a planning system.

In HINGE and most other practical planning systems, learned plans are stored as macro operator schemas both to save space and to make learned plans useful for solving many more problems. In the context of planning with reuse, the macro operator schemas (or macros that arise from them) under consideration by the planner are called reuse candidates and represent generalized solutions (or solutions) to previously learned planning problems.

2.2.3 Search Space: States or Plans. Planning with simple search strategies in a state-space has important implications regarding ordering constraints between operators and how operators are inserted into plans. Using a simple search method like depth-first search, every operator in the current plan has an ordering constraint with respect to every other operator, and operators in the plan are said to be *totally-ordered.* Some of these ordering constraints are required to establish preconditions or eliminate clobberers, but other ordering constraints are imposed by the search process itself. For example, consider the state space shown in Figure 4 and operators defined by Figure 5. Neither operator has any preconditions. Suppose the initial state is *S1* (the empty set) and the problem goals are *A* and *B* so that *S4* is a goal state. There are two different paths from S1 to S4. However, only one path will be found if depth-first search or another simple search strategy is used. The path found will depend on the planner's choice for the first goal to achieve, but either path leads to a plan in which operators are totally-ordered. Either plan includes an unnecessary ordering constraint between

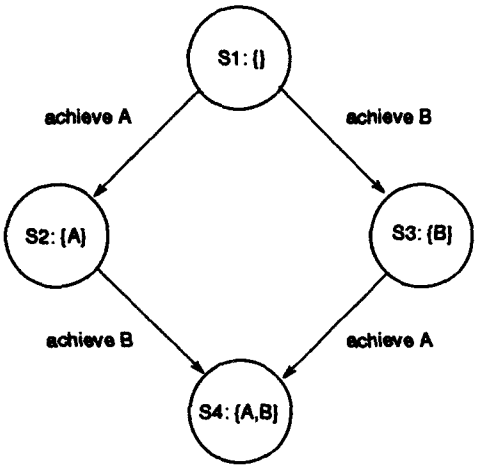


Figure 4. A state space with multiple possible paths.

Operator Definitions

name:	achieve B	achieve A
preconditions:		
effects	B	A

Figure 5. The definitions of operators of the state space shown in Figure 4.

operators. Adding this unnecessary ordering constraint is a form of over-commitment by the planning system.

Importantly, searching in a state-space does not imply a total-order on operators for every search strategy; it is the combination of search strategy and search space which determines this property. For example, a more sophisticated search method could be devised for finding both paths shown in Figure 4 and recognizing that no ordering constraint is required in the plan. Generally, such a strategy requires look-ahead and some analysis. For simple search strategies that lack these characteristics, the choice of a state-space implies totally-ordered operators. I call state-space-searching planning systems that use simple search strategies "state-space planners."

An alternative to searching in a state-space for a state that includes problem goals is to search in a space of plans until a solution is found. In a space of plans, the nodes are

plans (operators and ordering constraints on them) and for refinement planners[1], the transitions between nodes are the result of adding an operator or adding an ordering constraint to establish a precondition or eliminate a clobberer. I call such a planner a "plan-space planner[2]." Even with simple search strategies, plan-space planners do not imply a total ordering on operators and do not add arbitrary ordering constraints for the convenience of the search strategy.

It is commonly understood that committing to planning decisions is inappropriate whenever delaying the decision leads to discovering information useful for making a better decision; this strategy is called "least commitment." Many researchers in the planning community believe that searching in a space of plans is more flexible and supports a strategy of least commitment better than searching in a space of states (Kambhampati, 1992, McAllester and Rosenblitt, 1991, Tate, 1977, Wilkins, 1988).

2.2.3.1 Implications for Inserting Operators and Reuse. The total ordering of operators characteristic of state-space planning models requires that new operators can only be added at one end or the other of the sequence of operators in the current plan. If the planner searches forward from the initial state, then operators are added to the end of the current plan; it is also possible to find one or more goals states and search backward to the initial state.

Because transitions in a plan-space only involve adding operators and ordering constraints, new operators may be inserted anywhere in the current plan (Kambhampati and Chen, 1993:515). For example, Figure 6 shows paths to different op-

[1]Refinement planners only add operators or ordering constraints, but in general, transitions in a plan-space could also result from deletion of operators or ordering constraints. In this document, plan-space planners are assumed to be refinement planners.

[2]Planning models which search in a space of plans are often called "partial-order" planning models because they do not require a total-ordering on operators in a plan. The name "partial-order" distinguishes such models from other models which insert arbitrary ordering constraints purely as a convenience for the planning model. However, Frank Markham Brown points out that "partial-order" is a established term in logic with a meaning inconsistent with its usage in the planning literature (Brown, 1990). In logic, a partial-order is a reflexive, antisymmetric, transitive relation; in contrast, the relation of ordering constraints on operators in a plan is irreflexive, antisymmetric, and transitive (a quasi-order, in logic parlance). The "partial" part of the planning term refers to the fact that the relation of ordering constraints on operators is a partial relation (every operator is not necessarily ordered with respect to every other operator). To avoid confusion, I call planners that search in a space of plans "plan-space planners."

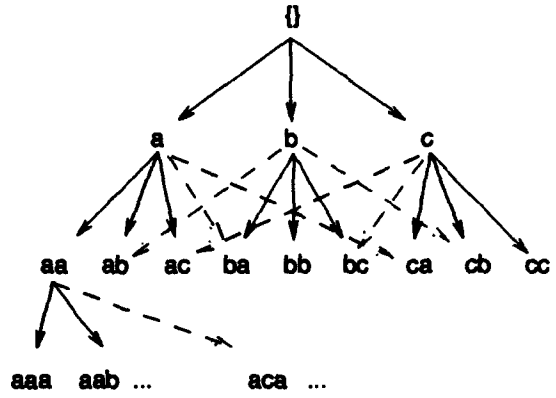


Figure 6. Allowed refinements of an operator sequence for operators a, b, and c (Kambhampati and Chen, 1993:515).

erator sequences. All paths shown are possible refinements if a plan-space is searched, but if a state-space is chosen instead, those paths that include dashed lines are not possible with simple search strategies. For exploiting plan reuse, Kambhampati and Chen have recently pointed out the potential usefulness of being able to insert operators at any point in the current plan (as discussed in Section 2.4.3):

> "...we argue that the real utility of using partial order [plan-space] planning ... is that it provides a flexible and efficient ability to interleave the stored plans with new operators, thereby significantly increasing the planner's ability to exploit stored plans." (Kambhampati and Chen, 1993:515)

By allowing macro insertion anywhere in a plan, a plan-space planner allows reuse precluded by state-space planners.

2.2.3.2 Implications for Nonlinear Problems. The choice of search space can affect the algorithmic requirements for solving certain problems and the length of the resulting solutions, but searching in either type of space can be effective for finding solutions, even for *nonlinear* planning problems. Nonlinear problems are those which cannot be solved by achieving problem goals sequentially, even when goals are considered in all possible orders. Nonlinear problems are caused by goal interaction: a planning system achieves a particular problem goal, but subsequently the goal is negated by an operator used to achieve another problem goal. As an example, the nonlinear block-stacking problem known as

Sussman's Anomaly is shown in Figure 7, and its state-space is shown in Figure 8. In Sussman's Anomaly, achieving either of the problem goals first leads to a state in which further progress depends on negating the achieved goal. For example, in Figure 8, if *on(A, B)* is achieved first, *state X* results and *on(A, B)* must be negated to reach the goal state. To find a solution to a nonlinear problem like Sussman's Anomaly, the planner must either protect the goals it achieves from being negated or re-achieve problem goals that have been negated.[3] While plan-space planners can use either of these methods with simple search strategies, state-space planners cannot protect goals without some form of look-ahead search and search-space analysis. For example, if a simple state-space planner achieves *on(A, B)* by reaching *state X* in Figure 8, the planner could protect this goal by refusing to move to a state that negates it, but doing so implies that the planner will not find the goal state. The only option for such a state-space planner is to re-achieve goals that are negated. For example, in Figure 8, if *on(A, B)* is achieved first, the most straightforward path goes from the initial state to *state X*, then to *state Z*, and then to the goal state. The drawback with goal re-achievement is that the resulting plan is longer than necessary. Of course, it is possible to overcome this drawback by writing a program which analyzes the plan and shortens it, or by using a more sophisticated search strategy using look-ahead search with analysis to avoid the drawback. These methods address the fundamental disadvantage of using simple search methods in a state space: over-commitment to unnecessary ordering constraints between operators. Rather than use a complicated planning algorithm to search in a space of states, some planners including HINGE are based on a simple algorithm that searches in a space of plans (Kambhampati, 1992, McAllester and Rosenblitt, 1991, Tate, 1977, Wilkins, 1988).

2.2.4 A Plan-space Planning Model. McAllester and Rosenblitt have defined a planning model that searches in a space of plans and only adds operator ordering constraints that are necessary for establishing preconditions or eliminating clobberers. The algorithm for McAllester and Rosenblitt's planning model is non-deterministic because it leaves the

[3]Some early planners such as STRIPS did not use either of these methods and could not solve nonlinear problems (Fikes and Nilsson, 1971).

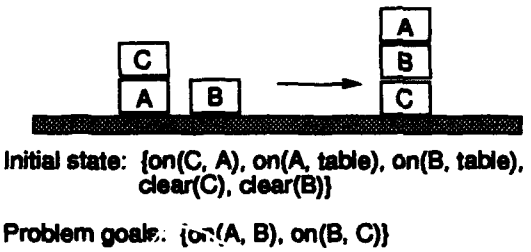


Figure 7. A block-stacking planning problem called "Sussman's Anomaly."

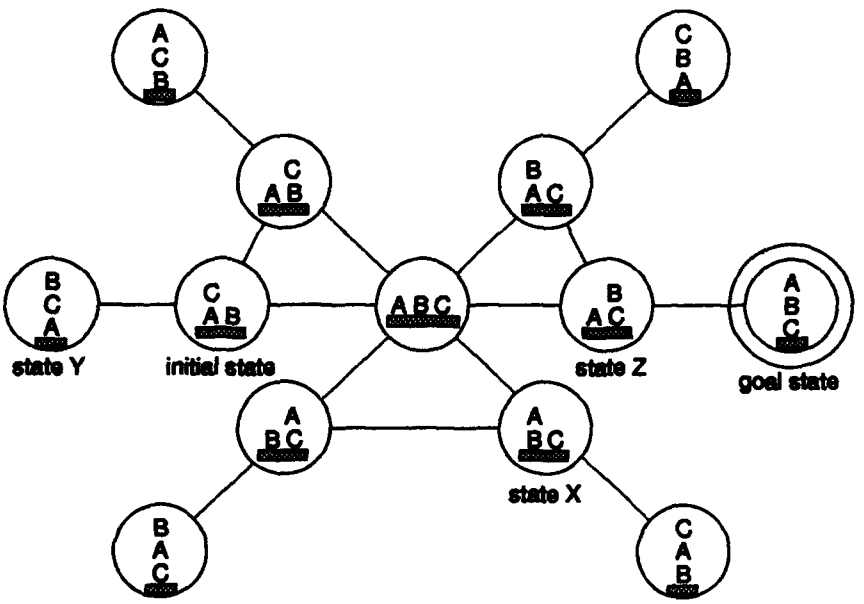


Figure 8. The state space associated with Sussman's Anomaly.

goal satisfaction algorithm unspecified (McAllester and Rosenblitt, 1991). McAllester and Rosenblitt's algorithm has served as the basis for several recent planners (Barrett *et al.*, 1991, Hanks and Weld, 1992), and HINGE is loosely based on this algorithm. A modified form of the algorithm is given by the following procedure FIND-SOLUTION whose arguments are a plan, a set of outstanding goals, and a set of establishments. FIND-SOLUTION is defined as:

1. If the current plan solves the planning problem, return the current plan.
2. Suppose C is an operator in the current plan, p is a precondition of C, and the set of establishments includes $< E,\ p,\ C >$. If there is a clobberer $Clob$ of $< E,\ p,\ C >$ in the current plan, then make a new plan by adding one of the following ordering constraints. Recursively call FIND-SOLUTION with the new plan and the current set of establishments.
   (a) $Clob \prec E$
   (b) $C \prec Clob$
3. Now there must be some operator C in the plan that has some precondition p with no associated establishment. If possible, do one of the following and recursively call FIND-SOLUTION with the new plan and new establishments that result. Otherwise, backtrack.
   (a) If the plan includes an existing operator $E1$ that includes p in its effects and does not include $C \prec E1$ in its ordering constraints, add the ordering constraint $E1 \prec C$ to the plan. Add the establishment $< E1,\ p,\ C >$ to the set of establishments.
   (b) Add to the plan a new operator $E2$ that includes p in its effects and the new ordering constraint $E2 \prec C$. Add the establishment $< E2,\ p,\ C >$ to the set of establishments.

The modified algorithm above differs from McAllester and Rosenblitt's original algorithm in that the original algorithm uses a stricter definition of "clobberer" than the one

in this chapter (McAllester and Rosenblitt, 1991:636). This chapter defines "clobberer" as Tate does (Tate, 1977). McAllester and Rosenblitt state their believe that the modified algorithm above works as well or better than their original algorithm in practice[4] (McAllester and Rosenblitt, 1991:638).

The algorithm above may be altered for planning systems that take operator schemas as inputs , rather than operators. To accommodate operator schemas, two alterations involving unification are required. First, instead of adding a new operator in step 3(b), an operator schema, instantiated so that it includes p, is added. Second, in a later call to FIND-SOLUTION, some of the remaining variables in the operator schema may be instantiated either to establish preconditions (in step 3(a) or 3(b)) or to eliminate clobberers in the plan (in step 2). Instantiating a variable in an operator schema is a planning decision, and, like other planning decisions, it is left unspecified by the algorithm. (McAllester and Rosenblitt, 1991:638)

2.2.4.1 Complete Planning Systems. FIND-SOLUTION is a non-deterministic algorithm. To build a planning system based on FIND-SOLUTION, a search strategy must be specified. If the search strategy is systematic and exhaustive, then the resulting planning system will always return a solution whenever one exists, and the planning system is said to be *complete*. Completeness is clearly a desirable characteristic from the standpoint of a user of a planning system, but often the exhaustive search required to make planners complete also makes them undesirably slow. Completeness has only recently become commonplace in planning systems, and it is not a property of some planners that have been shown to perform well in certain domains (Wilkins, 1988).

2.2.4.2 Plan Quality vs. Planning Speed. This research does not address the quality of plans (solutions) produced by a planning system. Plans are characterized in many ways including their feasibility, cost, flexibility, and relative demand for particular resources.

[4]Kambhampati has characterized the differences between the modified and unmodified algorithm in terms of a trade-off between search-space redundancy and over-commitment (Kambhampati, 1993). Kambhampati's results are consistent with McAllester and Rosenblitt's intuition.

The search strategy chosen for a planning system greatly affects the plans returned. However, this research does not address plan characteristics; instead, it focuses only on the speed of returning a solution, particularly in light of the utility problem.

2.3 Learning, Reuse, and the Utility problem

Except under very restrictive and impractical conditions, planning systems require an amount of time which varies exponentially with the number of operators in the plan (Bylander, 1991, Bylander, 1992). Therefore, building a fast planning system depends on finding a way to make short plans, even for complex problems. The most obvious method of making shorter plans is to use more powerful operators which achieve more goals. The easiest method of finding such operators is to learn them from planning experience, as macro planners do.

2.3.1 Explanation-based Learning (EBL). Forming a macro operator schema from a plan is a form of learning called *explanation-based learning* (EBL). There are many other types of learning that are also considered to be EBL (Ellman, 1989). However, all methods learn a generalized goal concept from a single training example using domain theory to guide learning. The "explanation" is a justification that the training example describes the goal concept. Once the generalized concept is learned, the concept must be represented in such a way as to be recognizable to the problem solver, a process called "operationalization." (Dejong and Mooney, 1986)

As applied to a plan, EBL results in a macro operator schema that represents a generalized plan. The goal concept is a plan for achieving some goals given some initial state. The training example is a specific problem together with its solution. The domain theory required is knowledge about establishments, clobberers, and plan applicability, as well as knowledge about domain objects and their generalization. Knowledge about establishments, clobberers, and plan applicability forms a causal model for explaining how the plan achieves problem

goals and what propositions of the initial state are required for the plan to be applicable. Knowledge about domain objects helps to determine how a plan should be generalized.

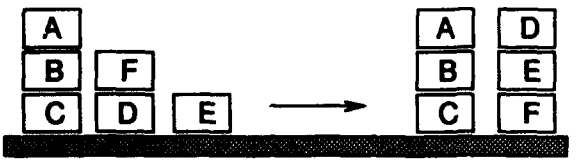
As an example, suppose an EBL system is to learn a plan for stacking blocks. Consider the block-stacking training example shown in Figure 9 and the operator definitions shown in Figure 10. Using this training example and domain knowledge, the EBL system can prove that the plan achieves problem goals and can identify blocks *A*, *B*, and *C* as irrelevant parts of the initial state. Knowledge of object properties allows the EBL system to avoid variablizing the table. Using domain knowledge, the EBL system can generalize the training example into a plan that solves the generalized Sussman's Anomaly as shown in Figure 11. Finally, this generalized plan must be operationalized so that it can be recognized and reused by the planner. The EBL system uses domain knowledge again to operationalize the generalized plan, resulting in the macro operator schema shown in Figure 12.

2.3.2 The Utility Problem. The primary motivation for learning and reusing learned plans acquired from explanation-based learning (EBL) is to enhance planning speed over that of a generative planner. Ironically, reusing plans *decreases* planning speed in some cases, a phenomenon called the *utility* problem. In Chapter I, the utility problem was introduced as an empirical observation on planning systems that reuse plans. Actually, the utility problem affects any system that reuses learned concepts.

> **Definition: Utility Problem** For systems that seek to enhance performance by reusing learned concepts, the utility problem is the possibility that the system's performance will degrade with additional learning. For macro planners, the utility problem is the possibility that the planner will plan more slowly after learning additional plans.[5]

Previously, the utility problem has been considered only in terms of costs, rather than in terms of both costs and benefits. No one has developed a planning model and algorithm designed to overcome the utility problem by increasing the likelihood of reuse without future

[5] A generative planner learns no plans which is certainly fewer than a macro planner. Therefore, another description of the utility problem is that a macro planner sometimes performs worse than a generative planner which is based on the same planning model.



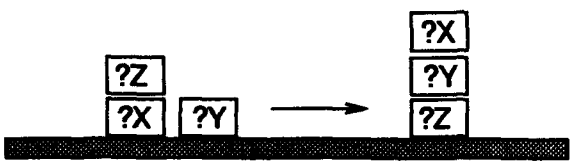
Problem goals: {on(D, E), on(E, F)}

Solution:
1. unstack(F)
2. stack(E, F)
3. stack(D, E)

Figure 9. A block-stacking problem with a solution.

unstack(F)	stack(E, F)	stack(D, E)
clear(F) on(F, D)	on(E, table) clear(E) clear(F)	on(D, table) clear(D) clear(E)
on(F, table) ~on(F, D) clear(D)	on(E, F) ~on(E, table) ~clear(F)	on(D, E) ~on(D, table) ~clear(E)

Figure 10. Operator definitions corresponding to the block-stacking problem solution in Figure 9.



Problem Goals: {on(?X, ?Y), on(?Y, ?Z)}

Solution:
1. unstack(?Z)
2. stack(?Y, ?Z)
3. stack(?X, ?Y)

Figure 11. The generalized Sussman's Anomaly and a generalized solution.

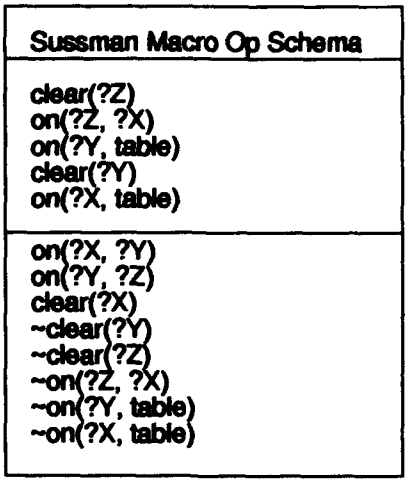


Figure 12. A generalized plan for Figure 11 operationalized by representation as an operator schema.

impact. Furthermore, previous research has focused on the cost of retrieving a good reuse candidate, rather than on the implications of inserting it into a plan.

Retrieving a reuse candidate was previously thought to require search, and the speed of retrieval has been thought to be a nonlinear function of the size of reuse candidates or the number of them in a library despite any indexing method used:

> "It is sometimes claimed that the utility problem will be "solved" by the development of highly parallel hardware and/or powerful indexing schemes. This opinion is based on the belief that either of these developments would make matching extremely inexpensive. If matching were inexpensive, this would largely eliminate any question of EBL's utility, since EBL, in effect, converts a tree search problem into a matching problem. However, this argument ignores two important points. First, the descriptions learned using EBL are neither bounded in number nor size. Secondly, the computational cost of matching grows rapidly with the number and size of the learned descriptions. Thus, the matching process requires search also (sometimes referred to as *knowledge* search)." (Minton, 1990:369).

Because retrieval has been thought to require general search, many researchers believe the utility problem can only be limited with selective learning (Greiner, 1989, Markovitch and Scott, 1989, Minton, 1985, Minton, 1990). In selective learning, derived concepts (plans) are evaluated using a heuristic function that predicts the value of the concept in future problem solving. Concepts without sufficient predicted future value are not learned.

Importantly, retrieval does not always require search. Furthermore, when search is required, normally a specific method of search can be used; this specific method may be a polynomial algorithm, rather than an exponential algorithm required for general search. The computational requirements for retrieval depend on the amount of information available; if complete information exists, then search is avoidable or if partial information exists, then general search may be avoided. Recently, Veloso has addressed the utility problem using an efficient retrieval method to find reuse candidates that are likely to save search in a case-based planner (Veloso, 1992). Veloso's approach is discussed in Section 2.4.4.

While focusing on the costs of retrieval, past research has neglected other costs that contribute to the utility problem. These other costs are associated with testing reuse candidates to see if they make planning faster. Previously, no one has found a way to guarantee that reusing a retrieved candidate will lead to a solution. Furthermore, there has been no attempt to quantify the amount of work entailed by inserting a reuse candidate into a plan. Instead, previous methods have been able to increase the probability that a reuse candidate will lead to a solution and increase the likelihood that inserting a reuse candidate will cause little or no planning work in the future. A useful definition that characterizes the "goodness" of a reuse candidate is:

> **Definition: Appropriateness** A reuse candidate is *appropriate* if the decision to insert it into a plan leads to a problem solution and does not have to be retracted before a solution is found.

2.4 Related Research

Approaches to limit the utility problem while planning with reuse have included

1. selective learning and unconstrained reuse of macro operators in a state-space planner;
2. reuse of one past plan with modification in a case-based planner;
3. unconstrained learning and reuse of macro operators in a plan-space planner; and
4. unconstrained learning and arbitrarily constrained reuse of plans in a state-space planner.

None of these methods promotes reuse while limiting the costs associated with the utility problem as well as the methods used in HINGE. Selective learning can preclude reuse that is possible if all plans are learned. Arbitrarily limiting the number of reuse candidates considered, as the second and fourth methods above do, also precludes reuse in some cases because a good reuse candidate may be known, but not be found. State-space planning implies that reuse candidates cannot be inserted except at one end of the sequence of operators that represent the current plan. Hence, a state-space planner precludes reuse that depends on interleaving operators, a capability allowed by a plan-space planner.

In contrast to these methods, HINGE is a plan-space planner that uses unconstrained learning and *selective reuse*. Selective reuse considers only appropriate reuse candidates; because the candidates are appropriate, limiting the number of reuse candidates considered never precludes reuse. When the available information is insufficient for selective reuse, HINGE uses a relaxed form of selective reuse which also uses appropriateness as a foundation for constraining the number of reuse candidates considered. Selective reuse is described in Section 4.5. In contrast, the second and fourth methods above use weaker methods to identify good reuse candidates.

2.4.1 MORRIS: Selective Macro Learning. In 1985, Minton studied reuse in STRIPS, an early state-space macro planner that suffered greatly from the utility problem (Fikes and Nilsson, 1971, Fikes *et al.*, 1972). From his analysis, Minton decided that STRIPS was performing poorly because it learned too many worthless plans. Minton felt that STRIPS would produce plans faster if only it learned fewer plans, particularly those with more value to future problem solving. To test his idea, Minton built a STRIPS-based planner called MORRIS that learned only two types of macros: those that were most commonly used in the experience of the planner and those that represented "non-obvious" solutions, implying that they saved a large amount of search. Minton compared the performance of MORRIS with that of a generative planner and a STRIPS-like planner that uses a non-selective EBL system. His results showed the value of selective learning for reducing the effect of the utility problem and improving reuse-related performance. (Minton, 1985)

The retrieval methods used in STRIPS and MORRIS search for macros linearly. Each macro in the library is a potential reuse candidate that must be tested to see if it achieves desired goals. In MORRIS, the utility problem is controlled because of two attributes of selective learning:

1. limiting the number of macros learned keeps the library small and reduces the time required to retrieve a good reuse candidate and
2. learning non-obvious solutions results in powerful macros which enhance the probability that reuse will dramatically improve performance.

Selective learning also contains the expansion of the search space to some extent because the smaller library that results means fewer reuse candidates are available to consider. However, this effect is coincidental and is not the primary focus of selective learning. With its policy of selective reuse, HINGE considers only a small fraction of the macros in the library and thus limits the search-space expansion much better than MORRIS does. In contrast to selective reuse, selective learning neglects the impact of inserting a particular macro into a plan and thus does not limit the future work the planner will have to do.

The biggest disadvantage of selective learning is that some concept that would be useful on a future problem may not be learned. It is difficult to accurately predict the needs of future problems in some domains. MORRIS uses a heuristic metric to determine if a macro or rule should be learned, but the heuristic may fail. In contrast to selective learning, selective reuse does not attempt to predict future value of a plan; instead, it learns all plans and filters reuse candidates when the future "arrives," along with problem information useful for the filtering process. By learning indiscriminately, HINGE avoids the work of predicting future utility and never fails to learn a concept that is useful on a future problem.

Like STRIPS, MORRIS is a state-space planner. As described in Section 2.2.3, a state-space planner must add operators to one end of a sequence of operators that constitutes the current plan, while a plan-space planner can insert operators at any point in the plan. Therefore, because of the order in which it achieves goals, MORRIS precludes reuse in some

cases that HINGE (a plan-space planner) does not. Minton has applied selective learning to another problem-solver, PRODIGY/EBL, with the same success (Minton, 1990). PRODIGY/EBL selectively learns control rules that can describe a broad range of concepts and learns from both successes and failures. Like MORRIS, PRODIGY/EBL is a state-space planner and is inherently less flexible in allowing reuse than HINGE.

Primarily because they have no special mechanism to do so, neither MORRIS nor PRODIGY/EBL can solve nonlinear problems. NoLimit, an algorithm that extends the search algorithm of PRODIGY/EBL, can solve nonlinear problems, but NoLimit is a state-space planner, and thus precludes reuse allowed by HINGE (Kambhampati and Chen, 1993, Veloso, 1992). HINGE is a plan-space planner and can solve nonlinear problems without any special mechanism.

2.4.2 CHEF: Heuristic Case-based Planning. Case-based reasoning is an effective method of improving performance through reuse that became popular during the mid-1980s. In the context of case-based planning, a "case" is a plan or generalized plan along with the problem it solves and a representation of the planning process used to produce the solution. Many researchers have developed case-based reasoning systems (Kolodner, 1983, Kolodner *et al.*, 1985, Koton, 1988), and a wealth of research foundations have been laid for memory-based problem solving by others (Carbonell, 1983, Schank, 1977). There are differences between case-based systems, but Hammond's CHEF is representative of many case-based planning systems.

In contrast to MORRIS, CHEF and other case-based reasoners do not regulate learning; instead, they work by very selective reuse. When CHEF is assigned a new problem, features of the problem and a heuristic retrieval procedure are used to retrieve the solution for a very similar previous problem encountered. Once retrieved, the past problem is analyzed by a set of heuristic critics to identify changes that are required to make the old solution fit the current

problem. After modifications are identified, a set of heuristic repair procedures is unleashed to perform them.[6] (Hammond, 1990)

It is interesting to note that the granularity of reuse candidates is also constrained in case-based planning. If there is a library case that matches a large part of the current problem, then it will be reused. Smaller cases that fit a smaller part of the problem (and thus could save a smaller amount of search) are ignored, even when no larger case can be found. Thus, case-based planners occasionally ignore a beneficial reuse opportunity.

The utility problem arises in CHEF in plan retrieval and during modification if the required plan modifications are ultimately impossible, but during both retrieval and modification, the effects can be minimized at the risk of precluding reuse. Learning a large number of plans impacts retrieval, but with proper indexing, retrieval performance degrades sub-linearly with the growth of the case library. Case-based planning is robust with respect to the reuse candidate. It is not necessary to get the solution to the *most* similar previously-encountered problem, although performance improvement varies with the modifications required; reusing a solution to *any* similar past planning problem will often result in a performance improvement. Therefore, retrieval in case-based planners can be less specific and faster than if the best candidate were required. As in STRIPS and MORRIS there is no guarantee that a reuse candidate is appropriate. The choice to reuse a case may eventually have to be retracted if the required plan modifications cannot be made; if so, then the time spent on case-based planning is wasted. There is no limit on the amount of work needed to modify a retrieved plan in case-based planning. However, by retrieving a plan that worked on a similar problem, case-based planners increase the likelihood that the plan is appropriate for reuse and requires a limited amount of modification. There are different strategies for handling failure in different case-based planners. Often, after failure in the case-based component, the problem is solved

[6] Kambhampati's Priar planner uses an analytic causal model consisting of the set of precondition establishments to retrieve reuse candidates and identify changes to be made by a generative planner. Thus Priar is a case-based planner that effectively eliminates dependence on heuristic procedures typically used in other case-based planners. (Kambhampati and Hendler, 1992)

by a generative component; this strategy contains the utility problem better than re-applying the case-based planner with an alternative case.

Typically, the case-based approach does not allow reuse of multiple cases. Thus, a typical case-based planner is not useful for problems that would benefit from reusing two or more solutions from past experience. For example, learning problems in CHEF's domain of Szechwan cooking cannot help CHEF solve a problem that involves goals of making dinner and keeping the kitchen clean, even when CHEF also learns plans for keeping kitchens clean. Because there are many more problem combinations than there are problems, it is important to be able to solve combination-type problems. Recently Veloso has developed a unique case-based planner that allows reuse of multiple cases (Veloso, 1992). (See Section 2.4.4).

Unlike typical case-based planners, HINGE and other macro planners can reuse multiple solutions from previously encountered problems. HINGE can insert multiple macros because it decomposes any set of goals which cannot be achieved directly. Case-based planners have not used decomposition, mainly because the largest benefit from reuse corresponds to reusing the largest candidate. Case-based planners focus on the effort needed to modify a known solution. In contrast, HINGE and other macro planners avoid modification entirely. Instead they look for other opportunities for reuse on the remaining sub-problems. Even though these sub-problems are smaller and the potential benefit from reuse is diminished, reuse is still likely to be cheaper than generation.

2.4.3 SNLP+EBL: A Plan-space Macro Planner. At about the same time that HINGE was developed, Kambhampati and Chen assembled a plan-space macro planner to study the usefulness of such a planner for EBL-based plan reuse (Kambhampati and Chen, 1993). Kambhampati and Chen's macro planner was constructed by adding an EBL system to the SNLP generative planner which implements McAllester and Rosenblitt's planning model (Barrett *et al.*, 1991). For the purposes of this discussion, this planner will be called "SNLP+EBL."

Because they are both plan-space macro planners, SNLP+EBL and HINGE allow reuse precluded by state-space planners and typical case-based planners. SNLP+EBL and HINGE are unique among macro planners in their special ability, during planning, to expand macros in a plan. To expand a macro, the macro is replaced by the plan it represents (and is annotated with). Expanding two macros allows interleaving their primitive operators, allowing ordering constraints and reuse that would not be possible without expanding macros. Because they are plan-space planners, neither SNLP+EBL nor HINGE require any special mechanism to do the interleaving (just as they do not require a special mechanism to solve nonlinear problems or to be complete planners). Furthermore, HINGE and SNLP+EBL can make use of multiple reuse candidates of any size to solve any planning problem, unlike case-based planners such as CHEF.

Unlike HINGE, SNLP+EBL was not designed to contain the utility problem. HINGE uses an efficient retrieval method based on hashing and specific indices. In contrast, SNLP+EBL searches the library linearly to find reuse candidates that match the most goals.[7] Retrieval based on goal matching is not as selective as retrieval based on appropriateness for reasons described in Section 4.2. Therefore, SNLP+EBL does not strongly constrain the amount of future work entailed by insertion of a reuse candidate, as selective reuse in of HINGE does. Finally, there is no restriction on the number of reuse candidates considered. For these reasons, SNLP+EBL does not contain the utility problem nearly as well as HINGE does.

2.4.4 PRODIGY/ANALOGY*: A State-space Case-based Planner.* Recently, Veloso has extended PRODIGY (a state-space planner) to reuse plans and control information found when deriving the plans, thus combining elements of macro planning and case-based planning. This planner, PRODIGY/ANALOGY, extends case-based planning by allowing multiple-case reuse. PRODIGY/ANALOGY also uses an efficient retrieval method to find similar past plans. Like other case-based planners, PRODIGY/ANALOGY addresses the utility problem by depend-

[7] Kambhampati has extensively researched retrieval based on a strong causal model which could be used to improve SNLP+EBL (Kambhampati, 1990). However, Kambhampati and Chen developed SNLP+EBL to compare a plan-space planner with other planners in terms of their ability to reuse plans. This purpose was well served by the simple storage and retrieval strategy described in (Kambhampati and Chen, 1993:517).

ing on its efficient retrieval method and its similarity-based estimate of "goodness" for the retrieval candidate (Veloso, 1992).

The hash-based retrieval method of PRODIGY/ANALOGY is similar to the one used in HINGE. Unlike HINGE, PRODIGY/ANALOGY uses a discrimination net to avoid testing preconditions common to two reuse candidate twice. HINGE's retrieval method is a bit less efficient, but it was simpler to implement and is sufficient for the purposes of this research. Both methods require polynomial time ($\mathcal{O}(n^3)$) and vary with the size of the initial state, the average number of macro preconditions, and the number of macros hashed to the same set of goals achieved (Veloso, 1992).

PRODIGY/ANALOGY uses its retrieval method to find macros that represent *similar* past plans. Therefore, the reuse candidates PRODIGY/ANALOGY considers are not guaranteed to work and do not imply any limitation on the work entailed by inserting them into a plan. In contrast, using its policy of selective reuse, HINGE retrieves candidates guaranteed to be appropriate for a problem or sub-problem. For reasons discussed in Section 4.2, appropriateness also implies that no future work will be required by an inserted reuse candidate. Because of selective reuse, HINGE contains the utility problem better than PRODIGY/ANALOGY.

To reuse plans, PRODIGY/ANALOGY searches in the library for a reuse candidate that achieves as many goals as possible. If no reuse candidate can achieve all outstanding goals, then the problem is effectively decomposed by finding a candidate which achieves some goals and searching again for a candidate which achieves the remaining goals.[8] This strategy works well when the library contains reuse candidates which achieve all or nearly all goals. Otherwise, the strategy must search through a much larger space to retrieve a reuse candidate because there are relatively many unique subsets when a set of goals is divided approximately equally. Thus, PRODIGY/ANALOGY has high retrieval costs for problems that must be decomposed into sub-problems of roughly equal size. Because of this strategy, PRODIGY/ANALOGY'S can efficiently solving only problems that are similar to previously solved problems, a limitation that

[8] An analogous decomposition method was developed for HINGE (see Section 3.7.3), but HINGE also uses other methods.

is characteristic of case-based planners. In contrast, with simple domain knowledge, HINGE can efficiently decompose problems into same-size chunks, resulting in a much more directed search for reuse candidates. Decomposition methods developed for HINGE are described in Section 3.7.

Finally, PRODIGY/ANALOGY is a state-space planner. As previously argued, state-space planners preclude reuse allowed by plan-space planners such as HINGE.

2.5 Summary

This chapter introduced plan-space planning and compared it with state-space planning in terms of the reuse allowed. A modified version of McAllester and Rosenblitt's plan-space planning algorithm was presented. Their algorithm results in a complete planner when exhaustive search is used. To speed planning, more powerful operators must be learned so that even complex problems can be solved using short plans. This chapter introduced explanation-based learning (EBL) and discussed the utility problem that arises when reusing EBL concepts. Importantly, the utility problem has been previously characterized in terms of costs, and primarily in terms of the cost of retrieving a good reuse candidate. Many researchers believe that selective learning is the only way to contain the utility problem, although the utility problem can also be addressed by limiting reuse to one candidate, as typical case-based planners do, or with efficient retrieval, as Veloso has done. Finally, four related planning systems were described in terms of their ability to reuse plans and their management of the utility problem.

The utility problem represents a fundamental limitation on planning systems that try to improve efficiency by reusing previous solutions. However, selective learning is not the only way to contain it. Combating the utility problem requires an architecture designed to increase the probability of reuse, while limiting the costs of retrieval and testing reuse candidates. Chapter III describes a planning system (HINGE) that promotes reuse with a flexible planning model which represents desired features of reuse candidates and offers multiple opportunities for reuse through search and decomposition. Chapter IV analyzes the costs of the utility

problem and derives a retrieval mechanism and a reuse policy that combine to reduce these costs.

III. HINGE

3.1 Introduction

Current methods for addressing the utility problem include selective learning (in macro planning) and limiting the reuse to a single candidate and/or efficient retrieval (in case-based planning). These methods try to solve the utility problem by limiting the cost of retrieval or by limiting expansion of the search space. In contrast, I treat the utility problem both in terms of costs and benefits from reuse. This chapter describes HINGE, a plan-space macro planner implemented to explore this eclectic approach. The chapter describes HINGE in general, but it focuses on HINGE's ability to promote search-saving reuse. Chapter IV describes two methods designed to strictly limit costs that contribute to the utility problem.

This chapter describes HINGE in terms of its planning model and search control mechanism. As an overview, the chapter begins by defining HINGE's planning model and abstract operators used in the model. An overview of HINGE is presented in Section 3.3, followed by a description of HINGE's basic search control mechanism and a simple example of planning (Sections 3.4 and 3.5). HINGE's hierarchical search control is discussed and a set of decomposition methods developed to support it are described in Sections 3.6 and 3.7. Section 3.8 presents a brief description of HINGE's EBL component. Finally, Section 3.9 reiterates the flexibility of reuse that HINGE supports by virtue of its planning model.

3.2 HINGE's planning model

To promote search-saving reuse, HINGE is based on a planning model that searches in a space of plans and explicitly represents goals that should be achieved by a reuse candidate. HINGE extends McAllester and Rosenblitt's plan-space model by including abstract operators, data structures for representing subsets of the outstanding goals. A subset of goals, in turn, represents a particular decomposition of a problem and also describes goals a reuse candidate must achieve. Thus, an abstract operator represents a sub-problem and can be used to

index a plan that solves the sub-problem. Abstract operators may also be used to establish preconditions and insert partial domain knowledge, when it is available.

In addition, HINGE's planning model was designed to concisely represent plans and the process of planning so that the planner's decisions can be explained. For representation, HINGE uses a single data structure that holds the context of the planning process. This data structure includes the set of establishments and operator ordering constraints from which a strong causal model may be derived. The planning process is captured by a plan tree that records the history of decision making and represents untried choices that are useful if backtracking is required.

3.2.1 A Data Structure for HINGE's planning model. For simplicity, HINGE's planning model uses only one data structure to represent outstanding goals, precondition establishments, and the current plan. This data structure, called a validated, hierarchical plan (VHPLAN, pronounced 'v-h-plan') also represents a plan tree that is used to make decisions about search control and captures planning decisions which led to the current plan.[1] This VHPLAN uses typed operators to differentiate actions and outstanding goals. Abstract operators, defined in the next section, identify goals to achieve, while concrete operators denote instantiated operator schemas inserted into the plan to achieve goals. Figure 13 shows a graphical representation of a VHPLAN. In the notation, precondition establishments and ordering constraints on operators are represented as directed links, parent/child relationships between operators are shown using undirected links, and operators are represented as boxes containing the operator name, preconditions, and effects. A distinguished operator labeled *root* is the root of the plan tree. The initial plan operators, *init* and *final*, are concrete operators which are not part of the plan tree. The leaves of the plan tree, along with *init* and *final*, represent the current plan. The parts of the VHPLAN (including establishments) that refer to leaves of the plan tree, *init*, or *final* define nodes in the space that HINGE searches. The

[1] The term "validation" is another term for "establishment" (Kambhampati and Hendler, 1992:206).

VHPLAN is useful because it compactly represents a great deal of information pertinent to planning.

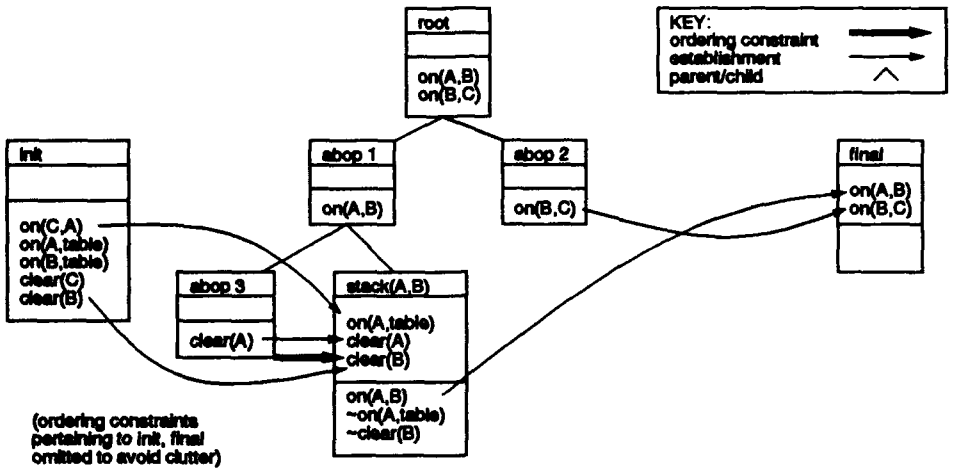


Figure 13. An example of the graphical notation used to denote a validated, hierarchical plan (VHPLAN) of HINGE's planning model.

3.2.2 Defining Abstract Operators to Record Goals and Insert Domain Knowledge.

To record planning goals and provide an insertion point for domain knowledge, I developed abstract operators and define them here. In order to systematically search its space, a planner must somehow keep track of the set of outstanding goals to be achieved. HINGE's model uses abstract operators for this purpose. Other operators are called concrete operators and denote actions that achieve goals:

> **Definition: Abstract and Concrete Operators** An abstract operator is a operator whose preconditions and effects are not completely specified in the sense that propositions are missing. If both preconditions and effects are completely specified for a particular operator, the operator is called a concrete operator.

Abstract operators support the insertion of useful domain knowledge when it exists. Without domain information, HINGE abstract operators never have preconditions and only have those effects that represent planning goals. Operator effects that achieve desired goals are called "primary effects" while those that do not are called "side effects." If partial (or total) domain knowledge exists, then preconditions and side effects in abstract operators support timely

insertion of the knowledge. For example, suppose a goal G exists, but available domain information specifies that G cannot be achieved without first achieving at least $P1$ and that at least the side effect $S1$ always accompanies G. The planner can use this information to immediately try to achieve $P1$ and ensure that $S1$ does not result in clobbering. If this domain information were not inserted until after the goal G were identified, then these planning actions would not be possible, and there would be no possibility to improve performance by re-directing search.

3.2.3 Using Abstract Operators in Establishments. Representing goals with an abstract operator leads to the possibility that an abstract operator may be used to establish new operator preconditions that it was not specifically inserted to establish. On the surface, this appears to be overcommitment on the part of the planner because there is no guarantee that the outstanding goals represented by the abstract operator will be achieved. However, the alternative choices may result in worse overcommitment and fail to take advantage of available information. To see this, consider the case in which an abstract operator $A1$ is the only existing establisher for a new operator's precondition p. Assume that to avoid overcommitment, $A1$ should not be used as an establisher. Two alternatives exist. The first alternative is to insert a new abstract operator $A2$ which has p as an effect, thus committing to doing additional work. Now the planner must find reductions for both $A1$ and $A2$. The planner has also committed to a course of action that assumes the failure of $A1$ and the accompanying failure of the concrete operator which requires $A1$. This planning choice is based on no new information and ignores existing information about goals to which the planner is already committed. The second alternative is to delay the choice of establisher for p until more information becomes available. For example, rather than establishing p with $A1$, the planner could choose to find a reduction for $A1$ that includes a concrete establisher for p. If a such a reduction is found, then this alternative was a poor choice—it would have been better to establish p with $A1$. If no such reduction is found, then the planner knows that $A1$ would have been a poor choice for establishing p, but it has no new information helpful to finding a better choice. The second alternative also ignores the planner's previous commitment to goals and

overcommits the planner to searching for reductions of $A1$. When knowledge of goals exists, the planner should make decisions based on the assumption that the goals will be achieved. The two alternatives to this approach disregard information and result in overcommitment that may be worse than committing to using an existing abstract operator as an establisher. Sections 3.4.1 and 3.4.3 discuss search control in HINGE designed to limit the impact of possible overcommitment associated with abstract operator establishment.

3.2.4 Definitions Required by Abstract Operators. Abstract operators appear in plans, but they should not appear in a solution because they represent goals and do not specify action. Therefore, introducing abstract operators requires a change to the definition of "solution" to avoid solutions that contain abstract operators. More generally, abstract operators in HINGE's planning model leads to a definition of typed plans.

> **Definition: Abstract and Concrete Plans** An abstract plan is a plan that includes at least one abstract operator. In contrast, a concrete plan is a plan that includes only concrete operators.

> **Definition: Solution** A solution to a particular planning problem is a concrete plan which is applicable and includes the initial plan of the planning problem.

Finally, HINGE's planning model requires some method of eliminating abstract operators in a plan. An abstract operator represents work to be done, and it is important to recognize that it may not be desirable to do all the work at once. The principle of least commitment suggests that some of the work could be done more efficiently if it is put off until more information is available. Therefore, the method of abstract operator replacement requires a data structure which is essentially a sub-plan, possibly containing other, new abstract operators to represent the delayed work. Such a sub-plan is called a *reduction* in the general case, or a *decomposition* if it is a sub-plan that contains only abstract operators. In HINGE, planning is the process of replacing abstract operators with reductions, or "reducing" abstract operators.

> **Definition: Reduction; Decomposition** A reduction of an abstract operator in a plan is a set of operators called reduction operators and a set of ordering constraints such that if the reduction replaced the abstract operator in the plan

1. the reduction operators would establish preconditions currently established by the parent abstract operator,
2. all preconditions of all reduction operators would be established, and
3. no clobberers of any establishment would exist.

A decomposition is a reduction that includes only abstract operators.

From this definition, it should be clear that reducing an abstract operator maintains the applicability of the plan.

The HINGE planning model represents planning as a tree search; in the tree, a parent/child relationship exists between an abstract operator and its reduction. For example, in Figure 13, *abop1* and *abop2* with no ordering constraints are the reduction, or in this case, the decomposition of *root*.

3.2.5 The Non-deterministic Algorithm of HINGE's planning model. HINGE's planning algorithm is specified below as the non-deterministic algorithm FIND-PLAN whose argument is a VHPLAN.

1. If the current plan of the VHPLAN is a solution to the planning problem, return the current plan.
2. Otherwise, there must be at least one abstract operator in the current plan. Select an abstract operator A from the current plan and for A find all reductions. The algorithm for finding reductions is McAllester and Rosenblitt's planning algorithm described in Chapter II. If there are more reductions, choose one, substitute it for the parent abstract operator in the current plan, and recursively call FIND-PLAN with this new VHPLAN that results; otherwise, backtrack.

Some differences between HINGE's algorithm and the modified version of McAllester and Rosenblitt's algorithm (Section 2.2.4) seem apparent, but they are actually quite similar. The major difference between the two algorithms is that they use different data structures which support different types of search control. HINGE's algorithm uses the VHPLAN data structure which includes a hierarchical plan tree with abstract operators representing subsets

of the outstanding goals. The hierarchical plan tree allows representation of a particular set of the goals of equal depth, while abstract operators provide a representation for a set of goals that are to be solved simultaneously. Both of these sets of goals are distinguished subsets of the whole set of outstanding goals being considered by the planner. The ability to represent these two subsets of the outstanding goals is important for directing search. Most importantly, representing a subset of goals to be solved simultaneously is critical for representing the part of the current problem that should be solved with reuse. In contrast, McAllester and Rosenblitt's algorithm uses the whole set of outstanding goals. This set representation cannot represent a distinguished subset of goals and thus cannot indicate a subset of goals to be achieved simultaneously. With McAllester and Rosenblitt's model, it is still possibly to find a reuse candidate that solves part of the problem, but the data structures used cannot represent *what part* should be solved. The result is that, if such information exists, it cannot be used to direct search. Section 3.6 discusses this point further.

Like McAllester and Rosenblitt's algorithm, the algorithm above is non-deterministic; both algorithms use depth-first search, but neither algorithm specifies how to make choices at branch points. The implementation of HINGE as a planner requires that search control be specified, thus HINGE's planning model is more general than the HINGE planner itself. HINGE's search control methods are discussed in Sections 3.4 and 3.7.

3.3 HINGE Overview

HINGE is a plan-space macro planner that includes an explanation-based learner for learning new plans and sub-plans. Because HINGE is a plan-space planner, it can insert operators anywhere in the plan and includes only required ordering constraints on operators as the plan is developed. Moreover, HINGE can optionally expand a macro in a plan, substituting its corresponding primitive operators and necessary ordering constraints for the macro itself. Thus, HINGE is capable of interleaving the primitive operators of macros to solve planning problems. HINGE and SNLP+EBL are the first planning systems that learn and can reuse macros in this way.

HINGE's search method is hierarchically-structured on the number of goals. By trying to solve goals simultaneously, HINGE promotes reuse of more powerful macros before weaker ones. The hierarchical structure imposed on search also facilitates sub-plan learning. If goals cannot be solved simultaneously, HINGE incrementally decomposes them using various strategies which are described in Section 3.7. Eventually, decomposition results in single goals that HINGE solves using generative planning. Because of its planning model, HINGE is sound[2] and causal structure systematic.[3] Because HINGE considers all reductions using an exhaustive search strategy, HINGE is also complete.

HINGE uses efficient storage, indexing, and retrieval mechanisms that, when the the initial state and problem goals are known, support polynomial-time access of macros that are guaranteed to be appropriate. If the problem must be decomposed, then the initial state and problem goals are known only for a sub-problem; in this case, the retrieval mechanism allows polynomial-time access of macros likely to be appropriate. The retrieval mechanism does not require general search and thus does not vary exponentially with the size of the macros or the number of them in the library. Storage and retrieval in HINGE are described in Chapter IV. HINGE includes an EBL component for learning macro operator schemas from the solutions to planning problems. HINGE's overall architecture is represented as shown in Figure 2 on page 2, where the output of the learner is macro operator schema, rather than operators.

HINGE is a domain-independent planner designed to use domain knowledge when it exists to enhance performance. For example, HINGE allows modular insertion of domain-specific procedures for choosing an abstract operator for reduction (goal ordering) and for selecting a reduction. These procedures are not described in this document. HINGE also performs better when there is a domain-definition of object independence. For example, knowledge of object independence was used to build a more specific index for the block-stacking problems which were used to develop and test HINGE. Independence is also the basis for two decomposition methods described in Section 3.7. Although a domain-definition of

[2]Soundness means that any plan returned is a solution of the problem being considered.

[3]Each node in the space being searched is defined by a plan and the set of establishments associated with the plan.

object independence is helpful for improving planning performance, lack of domain knowledge does not affect HINGE's completeness. Alternative methods of indexing and decomposition are available. However, for learning, a definition of independence is required[4], as described in Section 2.3.1. For example, two stacks of blocks that are both on the table are not considered to be related because of the special nature of the table. This domain information is required to avoid learning incorrect concepts.

HINGE is implemented in Common Lisp and CLOS and includes a number of utilities for displaying and analyzing the planning process. As a planning system designed to support further research, HINGE can easily be configured to run in many different modes. More than twenty global variables are used to vary HINGE's operation; many of the more important of these variables can be set using a simple graphical user's interface that is part of HINGE. HINGE has been used by three researchers to date. HINGE prototype applications include a scheduling algorithm for setting up a meeting without violating constraints imposed by the attendees, a labor-crew scheduler for a production shift, and an air campaign planning system.

3.4 Search Control

HINGE spends much of its time doing bookkeeping to maintain the plan tree and causal model used to ensure plan applicability when reductions replace abstract operators. These processes are interesting to define and challenging to implement, but they are well understood (Chapman, 1987, Currie and Tate, 1991, McAllester and Rosenblitt, 1991, Tate, 1977, Yang and Tenenberg, 1990). However, the search control used in HINGE is unique and interesting. As stated in Section 3.2.5, the overall search strategy is depth-first, and chronological backtracking is used to retract poor choices.[5] Within this framework, there are three places which obviously impact search in the HINGE algorithm: selection of the next abstract operator for reduction, generation of reductions, and preference-ordering of reductions. By setting

[4]It is possible to derive independence by analyzing the plan before learning as done in PRODIGY/ANALOGY (Veloso, 1992), but HINGE lacks the capability to do this and depends on a domain-definition of independence.

[5]HINGE searches its *plan-space* depth-first, but as discussed in Section 3.4.1, HINGE selects abstract operators from the plan tree using a breadth-first criterion. There should be no confusion about these terms because HINGE's plan-space is not the plan tree, but rather the changes in the leaves of the plan tree.

global variables, HINGE may be re-configured to alter its method of doing each of these. However, HINGE's default behavior is described here. This default behavior is designed to:

- focus attention on high-level goals
- reuse the most powerful reuse candidates first
- provide multiple opportunities for reuse
- limit the impact of future work
- make the best local decisions possible.

3.4.1 Selection of an Abstract Operator to Reduce. HINGE selects abstract operators to focus attention on important goals and to promote reuse of the most powerful macros first. Selecting an abstract operator to reduce amounts to choosing the order of goal achievement. HINGE's default method is to select abstract operators based on their depth in the plan tree: those of least depth. This strategy focuses attention on problem goals before operator sub-goals and avoids poor planning decisions which lead to blind alleys or looping. For abstract operators that are equidistant from the root, HINGE prefers the one with the largest number of effects because it corresponds to a potential opportunity to reuse the most powerful reuse candidate.

Selecting abstract operators based on least depth in the plan tree also reduces the potential overcommitment of depending on an existing abstract operator for precondition establishment. Existing abstract operators are always higher in the plan tree than new abstract operators. Therefore, if an existing abstract operator is used to establish a precondition of an operator external to the abstract operator's reduction, testing this choice is not delayed as it might be if abstract operators were selected some other way.

3.4.2 Generation of Reductions. In HINGE, reductions take on one of three forms as shown in Figure 14. Each form is described by the number and types of operators in the reduction. If the reduction includes a concrete operator, then it will be composed of either

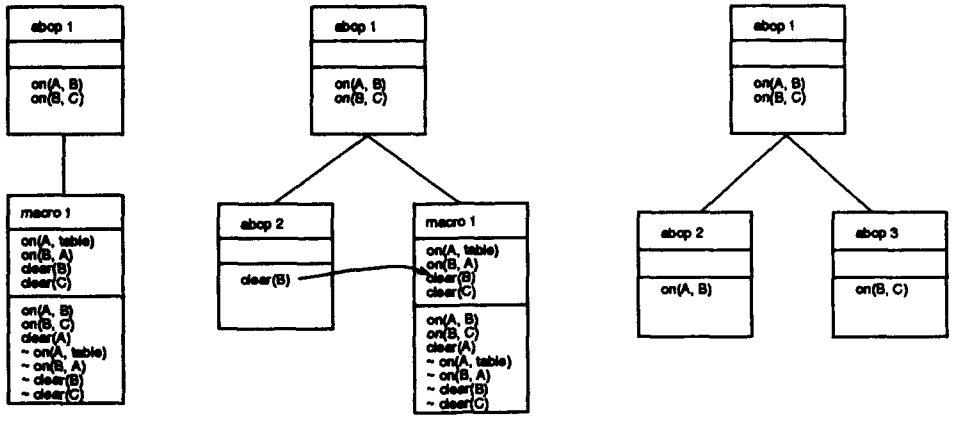


Figure 14. The three types of reductions allowed in HINGE.

a single concrete operator or a concrete operator and an abstract operator. Otherwise, the reduction will be a decomposition and will include multiple abstract operators (often two of them). These choices represent the simplest ways to construct a reduction and they correspond to three possibilities when trying to achieve goals represented by an abstract operator's effects. By definition, a reduction is required to achieve the effects of its parent abstract operator. If, by instantiating a library operator schema, a concrete operator can be found that achieves the effects of the parent abstract operator, then this concrete operator forms the basis of a non-decomposition-type reduction. If all preconditions of the concrete operator can be established by existing plan operators, then no other operator is required and the reduction is of the first form. Alternatively, if one or more preconditions of the concrete operator cannot be established by an existing plan operator, then HINGE adds a new abstract operator to establish these preconditions, and the reduction has the second form. Reductions of the second form always have at least one ordering constraint corresponding to the establishments made on behalf of the concrete operator. The third form of reduction arises in the event that the parent abstract operator has multiple effects and no operator schema in the library can be instantiated to achieve all of these effects. In this event, HINGE employs decomposition methods described in Section 3.7 to separate the goals represented by the parent's effects. This results in a reduction (a decomposition) that includes only abstract operators with no ordering constraints between any of them.

To limit the amount of future work, reductions are required to achieve all effects of the parent abstract operator. If a reduction fails to achieve all these effects, then some work is left undone and it is impossible to quantify the additional work required. In addition to limiting future work, insisting that the concrete operator of a reduction achieve all of the effects of the parent abstract operator is consistent with a strategy of selective reuse and greatly simplifies the macro retrieval method used in HINGE. This method results in far fewer reductions than would otherwise be generated.[6] If macros must achieve all effects of the parent, then the effects of operator schemas are a useful index for macro operator schema retrieval. Selective reuse and macro retrieval are described in Chapter IV.

In general, there are many ways to decompose an abstract operator, and the choice of decomposition clearly affects the search process. Decomposing an abstract operator is equivalent to partitioning a set of goals. One way to partition a set of goals is to make each element a singleton subset. This approach corresponds to completely decomposing an abstract operator with n effects into n abstract operators with 1 effect. To provide multiple opportunities for reuse, a less drastic approach is required because reuse is not possible unless multiple goals are to be achieved. Section 3.7 describes several decomposition methods developed for HINGE. These methods may be used to incrementally decompose problems and provide multiple opportunities for reuse, with the most powerful reuse attempted first.

By definition, no operator of a reduction may be a clobberer of any existing establishment in the plan. Therefore, the process of finding reductions includes a procedure that eliminates the possibility of clobberers either by adding ordering constraints or by discarding the reduction (when adding ordering constraints is not possible). As described in the definition of FIND-PLAN (page 42), reductions are found using McAllester and Rosenblitt's algorithm FIND-SOLUTION (page 19). For completeness, both alternatives in step 2 of FIND-SOLUTION must be considered.

[6]Requiring reductions to achieve all effects of the parent abstract operator has the disadvantage that some operator schemas that could be instantiated to achieve many of the parent's effects might be overlooked. Section 3.7.3 discusses a decomposition method that increases the visibility of such operator schemas and promotes their use.

3.4.2.1 Serendipity and Phantom Operators. Sometimes decomposition results in an abstract operator whose effects, by coincidence, can be satisfied by existing operators in the plan. Such effects correspond to goals that do not require achievement by new operators. It is important not to insert operators to achieve these goals, for doing so implies that action is required when it is not. In HINGE, there is a data structure called a "phantom operator" which exists only to associate a goal with an existing operator whose effects happen to establish it. A phantom operator is simply a concrete operator whose preconditions equal its effects. Phantom operators, being concrete, may form the basis of a reduction just as an instantiated library operator schema might. Interpreting a plan produced by HINGE must include removing any phantom operators and redirecting establishments as needed.

3.4.2.2 Phantom Goals. While phantom operators are an artifact of HINGE planning, phantom goals, on the other hand, are common in planning domains and are not unique to HINGE. Phantom goals are problem goals that appear in the initial state and will not be changed by the plan or require an operator to achieve. Before a plan is produced (and lacking special domain information), it is impossible to tell whether a problem goal that appears in the initial state is a phantom goal (will remain unchanged by the plan). Therefore, possibly-phantom goals present some difficulties that must be handled during search. Section 3.7.2 describes a decomposition method that is used to alter search based on the presence of problem goals that possibly are phantom goals.

3.4.2.3 Serendipity and Reduction Ordering. A concrete operator in a reduction under consideration can, by coincidence, establish one or more preconditions for operators in the plan other than those established by its parent abstract operator. This ability to establish extra preconditions is valuable if the preconditions are currently established by abstract operators. If so, then the reduction has additional merit that should be considered by the planner. HINGE takes advantage of this type of serendipity as described in Section 3.4.3.

Table 1. Cost weights for ordering reductions by means-everything analysis.

Reduction descriptor	*META Value*
Phantom Operator Reduction (Existing operators achieve goals)	-10,000
Serendipitous Establishment by Operator (per establishment)	-500
Abstract Operator (per abstract operator)	1,000
Abstract Operator Effect (per abstract operator effect)	500
Establishment by Abstract Operator (per establishment)	100
Required Ordering Constraint (per constraint)	50
Preconditions (per precondition)	20
Effects (per effect)	10

3.4.3 Ordering Reductions: Means-everything Analysis. To control search, it is helpful to assess, in as much detail as possible, the impact of inserting a particular reduction into the plan. Means-everything analysis is a domain-independent extension of means-ends analysis (Newell and Simon, 1963). For operators in the reduction, means-everything analysis considers not only the usefulness of the primary effects, but also the impact of side effects and preconditions. In particular, means-everything analysis considers

- the ability of existing operators to satisfy goals
- the usefulness of operator effects for serendipitously satisfying existing goals
- the potential work associated with unestablished preconditions
- the potential for backtracking because of dependence upon planning commitments (goals), rather than existing operators
- the potential for backtracking because of required ordering constraints
- the simplicity of the operator in terms of the number of its preconditions and effects

Means-everything analysis is similar in spirit to the domain-independent heuristic extension of means-ends analysis in SNLP, but means-everything analysis is more comprehensive in terms of the number of factors that it considers. In HINGE, means-everything analysis is implemented as a metric function on reductions. Table 1 shows the relative values assigned to each factor.

The costs shown in Table 1 were chosen based on extensive problem solving experience with HINGE. During problem solving, it is most important to avoid doing any work that has already been done. Therefore, a reduction containing a phantom operator is given a high negative cost because phantom operators represent outstanding goals which can be achieved by existing plan operators and require no additional work. Recognizing reductions that achieve goals serendipitously is also important, although not as important as avoiding work that is already done. Therefore reductions that support serendipitous establishments are awarded a moderate negative cost. To direct search, choosing reductions that require no extra work is quite important—more important than recognizing serendipitous goal achievement, but not as important as avoiding work that is already done. Furthermore, the amount of extra work is also important. Therefore, an abstract operator in a reduction adds a relatively high cost, and each effect in the abstract operator adds a moderately high cost. Reductions that depend on an external abstract operator for one or more establishments are penalized, but not as much as if they include an abstract operator. However, if a concrete operator can establish the same precondition, HINGE always prefers the concrete operator as an establisher. Each ordering constraint in the reduction costs something because ordering constraints impact the planner's ability to insert new operators. Ordering constraints are not considered as bad as depending on an external abstract operator to establish a reduction operator's precondition, however. Finally, the preconditions and effects of reduction operators cost a small amount to show preference for simple reductions over complicated ones.

Some of the weights are designed to to allow one factor to overwhelm all others, while other weights give factors more equal consideration. For example, a reduction with a phantom operator is considered to be very desirable because it represents knowledge of a part of a sub-problem that is already solved. Therefore, a reduction containing a phantom operator is currently preferred in any case.[7] In contrast, if two reductions both contain only one concrete

[7]HINGE's reduction metric makes no assumptions about the form of reductions allowed. Therefore, it is possible that a reduction which includes many abstract operators and a phantom operator would not be preferred.

operator, there is no preference between them when one of the concrete operators has two preconditions and one effect while the other operator has one precondition and four effects.

3.5 Planning in HINGE

HINGE plans by reducing abstract operators in an abstract plan until there are no more of them and the plan is concrete. At all times, HINGE maintains the applicability of the plan by finding reductions using the causal model implied by the set of recorded establishments which are part of the VHPLAN. It may be helpful to some readers to consider a simple example of generative planning at this point.

Figure 15 shows the definitions of the primitive operator schemas for the block-stacking domain. The *move-to* operator schema represents the action of moving some block from the top of one stack to the top of another stack. Similarly, the *stack* operator schema represents moving a block from the table to a stack and the *unstack* operator schema represents moving a block from a stack to the table. In all operator schemas, variable constraints force variables to unify only with block names and not to unify with the table. (Otherwise, *move-to* alone would suffice). Figures 16–20 show how the VHPLAN emerges as HINGE solves Sussman's Anomaly.

HINGE's input is an initial plan, consisting of operators *init* and *final* which are made from the initial state and problem goals according to the definition of initial plan given in Section 2.2.2, and a library of operator schemas that may be used to construct reductions of abstract operators. Because this example is for generative planning, assume that the library contains only those operator schemas shown in Figure 15. As planning begins, HINGE adds an abstract operator as the root of the plan tree and adds establishments to make the initial plan applicable, producing the VHPLAN shown in Figure 16. The root abstract operator has no preconditions and has effects equal to the problem goals. In Figure 16, the effects of the root abstract operator have been used to establish the preconditions of *final* to make the plan applicable. The ordering constraints shown in Figure 16 exist because, by definition, *init* precedes all other operators while all other operators precede *final*.

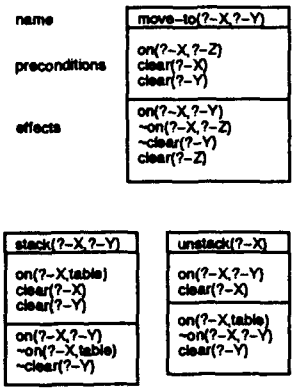


Figure 15. The primitive operator schemas for the block-stacking domain.

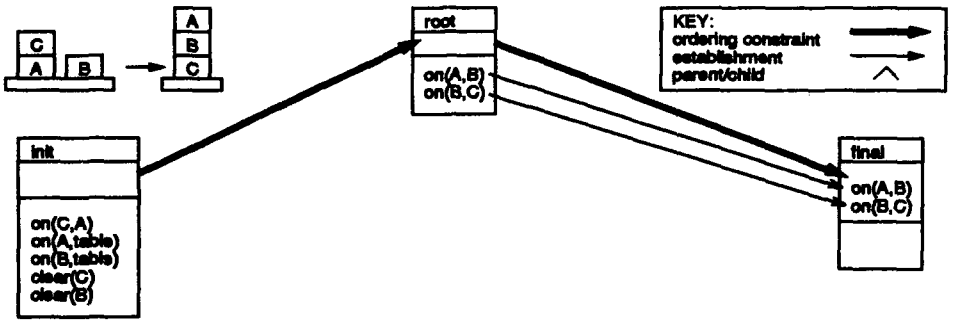


Figure 16. The initial VHPLAN for HINGE solving Sussman's Anomaly.

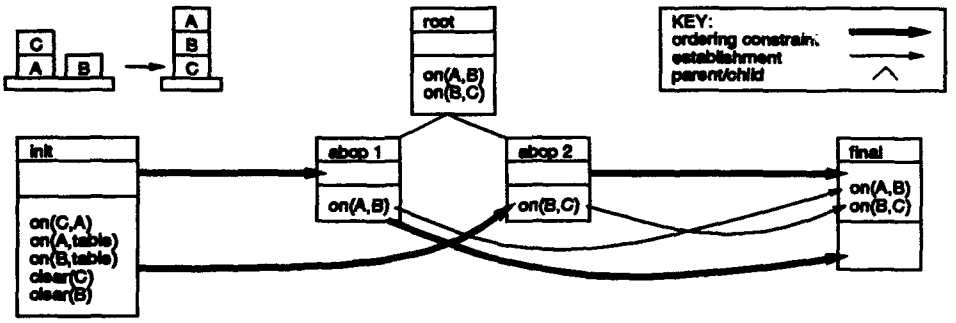


Figure 17. The VHPLAN after a complete decomposition of the root operator.

To plan, HINGE recursively selects an abstract operator to reduce, annotates the abstract operator with all possible reductions, and replaces the abstract operator with a selected reduction to produce the successive plan. Selecting abstract operators, finding reductions, and choosing a reduction to replace the parent abstract operator are planning decisions that are specified in Section 3.4, although this simple example will not illustrate all facets of search control. From the VHPLAN shown in Figure 16, HINGE can only select the root operator for reduction. Because no operator schema in the library can be instantiated to establish the effects of the root operator, the only reductions must be decompositions. Furthermore, because the only decomposition possible for an abstract operator of two effects is a complete decomposition, HINGE finds only one reduction, the complete decomposition. Reducing the root operator with its complete decomposition, adding new ordering constraints, and redirecting establishments results in the plan indicated by the VHPLAN of Figure 17.

HINGE must now select between reducing *abop1* and *abop2*. Because both abstract operators are equally deep in the plan tree and because they have an equal number of effects, the choice is arbitrary. Suppose HINGE selects *abop1*. Two operator schemas in the library may be instantiated to establish the effect *on(A,B)*. The schemas are named *move-to(?-X,?-Y)* and *stack(?-X,?-Y)*. Both of the concrete operators that result from the operator schemas require an abstract operator to help establish their preconditions and therefore both of the resulting reductions include at least one ordering constraint. No decompositions are possible because *abop1* has only one effect. Therefore, there are only two possible reductions of *abop1*. Neither reduction has clobberers. HINGE selects reductions based on means-everything analysis and the heuristic weighting function described in Section 3.4.3. HINGE selects the reduction of *stack(A,B)* because it has fewer preconditions that cannot be established by existing operators in the plan. Thus, the abstract operator required to partially establish *stack(A,B)* has fewer effects than the one required for *move-to(A,B)*. The relative weights make this factor overwhelm another factor that also favors the reduction of *stack(A,B)*: *move-to(A,B)* has one more effect than *stack(A,B)*, meaning that *move-to(A,B)* is more complex that *stack(A,B)*. Figure 18 shows the VHPLAN that results after HINGE reduces *abop1*.

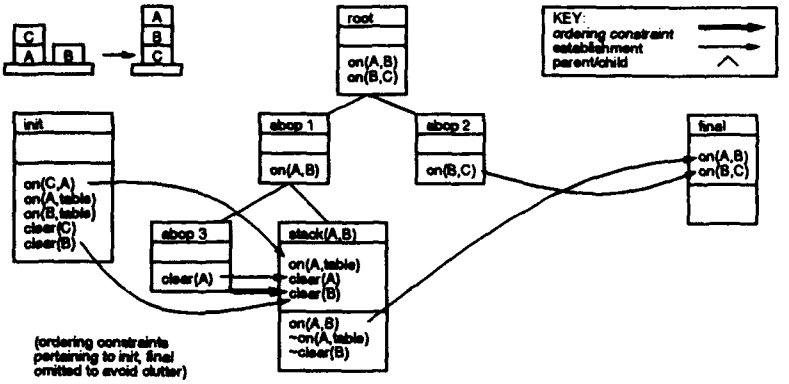


Figure 18. The VHPLAN after reducing abop 1.

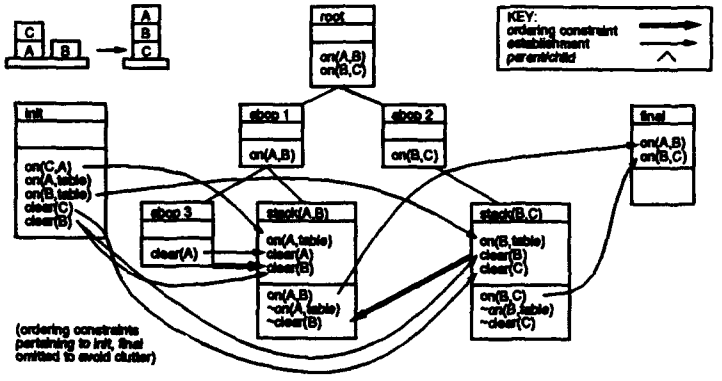


Figure 19. The VHPLAN after reducing abop 2.

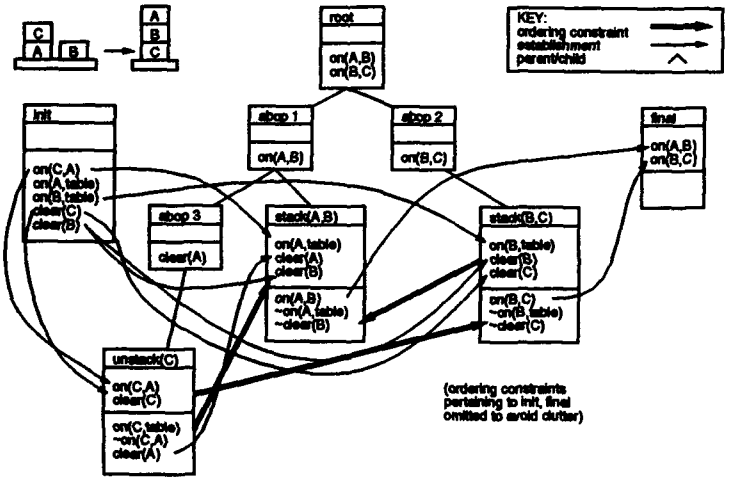


Figure 20. The VHPLAN representing a concrete plan resulting from reducing abop 3.

HINGE must now select between reducing *abop2* and *abop3*. Because *abop2* is less deep in the plan tree than *abop3*, HINGE selects *abop2*. Once again, two operator schemas in the library may be instantiated to establish the effect *on(B,C)*. As before, the schemas are *move-to(?-X,?-Y)* and *stack(?-X,?-Y)*. However, this time the concrete operator that results from instantiating *stack(?-X,?-Y)* does not require an abstract operator to establish any preconditions, while the concrete operator that results from instantiating *move-to(?-X,?-Y)* requires an abstract operator to establish *on(B, ?-Z)*. Once again, no decompositions are possible, and there is a choice between just two reductions. As before, neither reduction has clobberers. HINGE uses means-everything analysis again to select the reduction of *stack(B,C)*. This time, it is the existence of an abstract operator in the reduction of *move-to(B,C)* that overwhelms other considerations in selecting the reduction. HINGE installs the reduction of *stack(B,C)* in the VHPLAN by making it the child of abop 2 and redirecting ordering establishments as shown in Figure 19. Notice the ordering constraint that requires *stack(B,C)* to come before *stack(A,B)*. HINGE adds this ordering constraint to preclude *stack(A,B)* from clobbering the required establishment <*init, clear(B), stack(B,C)*>.

Finally, HINGE must reduce the remaining abstract operator, abop 3. Two operator schemas in the library may be instantiated to establish the effect *clear(A)*, and there are two reductions possible. The instantiated operator *unstack(C)* requires no abstract operator, so HINGE installs a reduction containing *unstack(C)*, resulting in the VHPLAN shown in Figure 20. Notice the ordering constraint that constrains *unstack(C)* to come before *stack(B,C)*. HINGE adds this ordering constraint to preclude *stack(B,C)* from clobbering the establishment <*init, clear(C), unstack(C)*>. There is also an ordering constraint between *unstack(C)* and *stack(A,B)*. This constraint was redirected from abop 3 when it was reduced by *unstack(C)*.

The current plan is represented by the leaf nodes of the plan tree, *init*, and *final*. To interpret this plan, the leaves of the plan tree are interpreted as actions and the ordering constraints are used to order actions. In this example, the plan is totally ordered, but that is not generally true.

In this example, no backtracking occurred to retract a poor planning decision. However, in the event that an abstract operator has no reductions, HINGE backtracks chronologically by removing the reduction that includes the abstract operator and substituting a different reduction associated with the abstract operator's parent. If the parent has no more reductions, HINGE backtracks until some ancestor has an untried reduction. If the root operator is reached and has no more reductions, then HINGE reports failure. Alternatively, if no more abstract operators remain in the current plan, HINGE returns the solution.

3.6 Hierarchical Reuse and Efficiency

HINGE is intended to promote reuse by structuring both the storage and reuse opportunities of learned macros hierarchically, where the hierarchy is based on the number of goals to be achieved. The number of goals is a heuristic indicator of the amount of work required and hence, the search that may be saved by reuse. In addition, the number of goals is a feature which may be used to index stored reuse candidates. HINGE actively promotes reuse of the most powerful candidate available, and in particular, HINGE seeks reuse candidates in decreasing order of the number of goals to be achieved. Thus, by default, HINGE initially acts like a case-based planner in that it attempts to solve the whole problem with one reuse candidate. Otherwise, the problem is decomposed using methods described in Section 3.7 and the subproblems are solved through recursive calls to the planner. Eventually, decomposition will result in abstract operators that each represent only one goal. In this event, HINGE acts like a generative planner, trying to solve the problem using only primitive operators. Using this strategy of hierarchical search, HINGE maximizes the likelihood of performance improvement through reuse and gracefully degrades to generative planning whenever reuse is not possible.

There is a subtle difference between the data structure used in the HINGE model and a goal set, but the difference has an important impact on search control. As stated in Section 3.2.5, the biggest improvement of HINGE's model over McAllester and Rosenblitt's model is that the HINGE algorithm manipulates a data structure (abstract operators in the

VHPLAN) that can represent a subset of the outstanding goals that should be achieved simultaneously. Because this set of goals is also used as an index for retrieving a reuse candidate, the HINGE model can specify a desired reuse candidate. Conversely, a set representation for goals cannot represent a distinguished subset; thus for models that depend on a goal set, finding a good reuse candidate is more haphazard. To illustrate this point, consider the block-stacking problem shown in Figure 21 and the two macros shown in Figure 22. A planner that depends on a goal set might choose to insert *Macro 2* because it achieves more of the problem goals than does *Macro 1*. For example, STRIPS would select Macro 2, because STRIPS uses means-ends analysis and a difference metric based on the number of goals solved.[8] Alternatively, if a planner can represent distinguished subsets of the outstanding goals, as HINGE can with abstract operators, then it can specify that the goals *on(A,B)*, *on(B,C)*, *on(C,D)* should be solved simultaneously, and therefore that *Macro 1* is preferable to *Macro 2*. For example, these three goals are represented by *abop 1* in the VHPLAN shown in Figure 23. *Macro 1* is a concrete operator that achieves all effects of *abop 1* and serves as the basis of a reduction for *abop 1*.

Knowledge of *which* subsets of the outstanding goals should be achieved simultaneously often comes from the domain, rather than from the HINGE planning model. For example, relationships between objects can sometimes be used to identify independent sub-problems or problems that are likely to be independent. Here, the term "independent" implies that no harmful goal interaction will occur in the plan. In Section 3.7, for example, two goal decomposition strategies which are based on a domain-definition of independence are described. If such domain knowledge exists, then the HINGE model can represent it in a data structure and use the data structure to direct search.

3.7 Decomposition Methods

In the HINGE model, if no appropriate reuse candidate exists for a particular problem, then the planner must have a method for decomposing abstract operators (partitioning a set

[8] Means-everything analysis would pick *Macro 2* also.

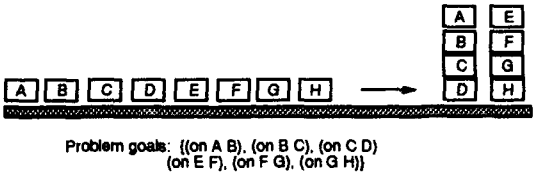


Figure 21. A block-stacking problem.

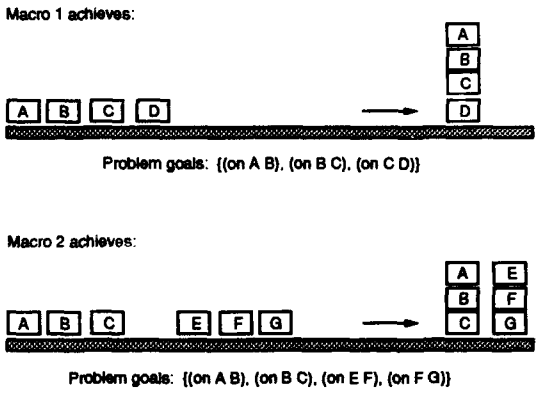


Figure 22. Macros that could be used to solve part of the block-stacking problem in Figure 21.

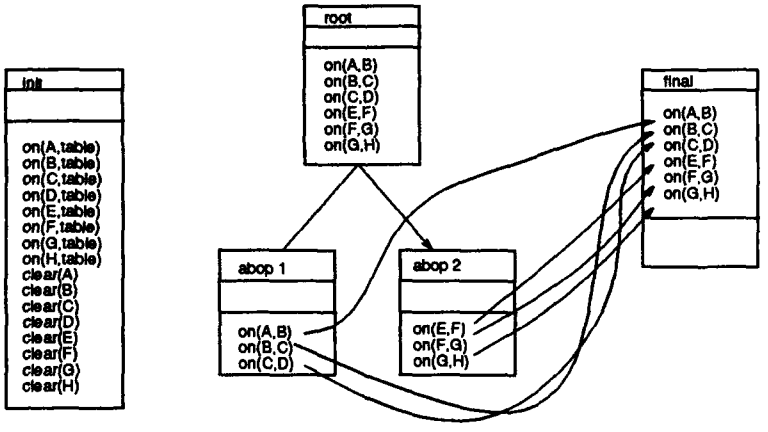


Figure 23. A HINGE VHPLAN representing sets of goals to be solved simultaneously for the block-stacking problem.

of goals). There are $\mathcal{O}(a^n)$ ways to decompose a set of n goals. Therefore, finding a good decomposition by exhaustive search may be expensive depending on the contents of the library of reuse candidates and on the domain (for the distribution of good decompositions in the space of all decompositions).

One apparent method of goal decomposition is complete-decomposition: trying to achieve each goal by itself. This method precludes reuse, but is good when reuse fails and generative planning is required. Other methods of decomposition are required for reuse. In this research, several methods were developed, and in HINGE, these methods are applied in a particular sequence which is designed to maximize reuse opportunities and separately solve independent or likely-independent sub-problems. The developed decomposition methods are based on a domain-dependent definition of independence, the existence of possibly-phantom goals, and contents of the operator schema library. The last decomposition method applied yields a complete decomposition. This sequence of decompositions gives HINGE its graceful-degradation behavior and guarantees completeness because the underlying generative planner in HINGE is complete. The following sections describe different methods of decomposition used in HINGE.

3.7.1 Independence-based Decomposition. When a domain-definition of independence exists, independence is a useful basis for decomposition. Operators are described by propositions which themselves describe the relationship between objects in a domain (or an attribute value for an object). Often a relationship between two objects can be used to describe a dependence between them that is important from a planning perspective. For example, in Sussman's Anomaly shown in Figure 7, it is clear that in the initial state Block C is related to Block A by virtue of the relation *on(C, A)*. By analyzing a set of propositions, sets of related objects can be identified. Because a state is a set of propositions, the set of objects in a state may be partitioned so that each partition-set contains objects related only to other objects in the partition-set. For example, the objects in the initial state shown in Figure 7 on page 20 are {Block A, Block B, Block C} and the set of related sets of objects (the partition) is {{Block A, Block C} {Block B}}. To find the independent objects in a plan, it is necessary

to analyze all the states visited when the plan executes. Alternatively, it is possible to predict the independent objects in a plan by analyzing the operators available to transition between states. For example, in most block-stacking examples, the primitive operator schemas are defined so that objects that are independent both in the initial state and in the problem goals are independent in some plans. Because of this, the independent objects can be used to define sub-problems which can be solved independently.

In HINGE, problems which cannot be solved with a whole-problem solution are decomposed using problem-independence when a domain-definition of independence exists. As an approximation to problem-independence, HINGE also uses *goal-independence* which is based on objects that are unrelated in the problem goals.

3.7.2 Phantom-goal Decomposition. Sections 3.4.2.1 and 3.4.2.2 described phantom operators and phantom goals. While HINGE's phantom operators are an artifact of HINGE planning, phantom goals or at least "possibly-phantom" goals are common in most planning problems. Any proposition that appears in the initial state and in the problem goals is possibly a phantom goal. Possibly-phantom goals arise because, before planning, it is hard to predict how the plan will affect objects. Therefore, the user of a planning system generally includes in the problem goals any initial-state proposition that should be maintained. For example, in formulating a plan for deploying troops, one problem goal might be to have transport aircraft engines in working order after deployment, even though the engines are working initially. The planner might be able to find a plan that does not affect the working state of the aircraft engines, in which case the goal is a phantom-goal, but if not, it will have to come up with a plan that includes repairing the engines to restore them to working order.

Possibly-phantom goals are a useful basis for decomposition. Any goal that turns out to be a phantom-goal requires no additional operators to be inserted by the planner. Therefore, it is often a good strategy to separate possibly-phantom goals from other goals to delay their consideration. In HINGE, the set of possibly-phantom goals is decomposed completely, and

the abstract operator that represents each goal is artificially deepened in the plan tree to delay its consideration.

3.7.3 Library-based Decomposition. Because the objective of decomposition methods in HINGE is to increase opportunities for reuse, it makes sense to examine the potential contribution of the library as an information source. After all, a macro operator schema can only be reused if it exists in the library. This section presents two methods by which knowledge of library contents can be used to find a decomposition. An advantage of using either of these methods is that retrieval of the reuse candidate can happen at the same time as the decomposition and without additional cost.

When no domain-dependent notion of independence exists, these methods can be useful. However, they are not useful for increasing the size of a library which contains only primitive operator schema. Clearly no reuse candidates can be retrieved if the library has none to begin with. By extension, these methods are less likely to work if the library is sparsely filled than if it contains many different reuse candidates.

3.7.3.1 Big-chunk-decomposition. In planning with reuse, the best performance occurs with reuse of the macro that saves the most search. Normally, this macro corresponds to the one that solves the most problem goals. When the library has no macro operator schema that achieves all of the n problem goals, the best alternative is to find a macro operator schema that solves $n - 1$ of the problem goals. HINGE's method of big-chunk-decomposition uses this approach. This decomposition method requires a procedure that decomposes n goals into a pair of sets, one with $n - s$ goals and the other with s goals, where $s = 1, 2, 3, ...SLimit$. In HINGE, all such pairs of sets are found for a particular value of s and reuse candidates are sought to achieve goals in the larger set of the pair. If no candidates are returned, the value of s is incremented unless its limit has been reached, and reuse candidates are sought again. The method fails if s reaches its limit before a reuse candidate is found. Alternatively, if a reuse candidate is found, HINGE forms a decomposition of the problem goals based on the reuse candidate. For a given s, HI big-chunk-decomposition yields

$(n!)/(s! * (n-s)!)$ pairs of sets (so large values of n and $SLimit$ are unlikely to give good performance).

3.7.3.2 Same-size-decomposition. While big-chunk-decomposition is unattractive as the size of the small subset increases, *same-size-decomposition* is a *polynomial* method which uses the library to decompose a planning problem. In same-size-decomposition,[9] reuse-candidates are selected arbitrarily from the library and tested to see if

1. they achieve some of the outstanding goals and
2. their preconditions can be established by existing plan operators.

If a reuse candidate's effects are a subset of the outstanding goals and if the candidate's preconditions are a subset of any of the possible states in which it is applied (according to operators and ordering constraints in the plan), then the reuse candidate solves part of the problem. The solved part of the problem and the remaining part indicate a decomposition.

As discussed in Section 4.4.3, the subset tests require $\mathcal{O}(n^2)$ time in the worst case for each reuse candidate tested. Therefore, in the worst case, n reuse candidates are tested, and this decomposition method requires $\mathcal{O}(n^3)$ time. However, even though same-size-decomposition is a polynomial method, the number of macros that can be instantiated from library operator schemas is quite large. For this reason, good performance is not likely with same-size-decomposition, and HINGE does not currently implement same-size-decomposition.

3.8 Learning

3.8.1 Learning Mechanism. HINGE's learner is typical of other explanation-based learning (EBL) procedures used to construct macros for planning systems. To make a macro

[9] The name for this decomposition method is indicative of its development, rather than its usefulness. Because big-chunk decomposition becomes expensive as the size of the smaller subset increases, another decomposition method was sought that could decompose problems more evenly at lower cost, and same-size-decomposition can sometimes do this. However, same-size-decomposition may be used to produce arbitrarily lop-sided decompositions as well.

from a plan, all that is needed is to find the macro's preconditions and effects and give the macro a name. The macro's preconditions are the set of all plan operator preconditions which require establishers that are external to the plan. The macro's effects are found by simulating the plan using the macro's preconditions as the initial case. Preconditions and effects of a macro are subsets of all preconditions and all effects, respectively, of the primitive operators in the plan represented by the macro; a macro is annotated with this plan, but the plan is not used during planning unless the macro is expanded. In many macro planning systems and in HINGE, a learned macro is generalized (made into an operator schema) for compact storage.

In HINGE, macros are learned in response to problem solving without regard to their usefulness in future planning. When a plan is returned, HINGE learns all sub-plans that save search when tested on the training problem. Each abstract operator in the plan tree is the "root" of a sub-plan; the sub-plan consists of the leaves of the corresponding sub-plan tree.[10] After eliminating phantom operators, HINGE learns from each sub-plan that has multiple concrete operators which are related to one another. As discussed in Section 3.8.2, plans with unrelated operators are not generally valuable for future problem solving, and learning such plans leads to potentially large storage requirements. Two operators are related to one another whenever they appear together in an establishment or, except for an ordering constraint, would clobber an establishment—in short, whenever there is an ordering constraint between them. For reasons discussed in Section 3.8.2, HINGE's learning element uses other filters as well, but none of the filtering criteria depend on a prediction of future utility. Therefore, HINGE does not learn selectively, as PRODIGY and MORRIS do.

Kambhampati and Kedar have characterized the EBL macro learning techniques as being unsuitable for learning plans with partially-ordered operators such as the ones learned by HINGE. The unsuitability arises because reuse of a generalized solution with partially-ordered operators may result in variables being instantiated in a way that was not intended. For example, consider Kambhampati and Kedar's example shown in Figures 24 through 27. Suppose a planner returns the plan with partially-ordered operators illustrated in Figure 25 for

[10]The sub-plan must be a concrete plan, and concrete operators are always at the leaves of a plan tree.

the problem illustrated in Figure 24. An EBL component like the one used in HINGE returns the generalized plan shown in Figure 26. It is desirable to ensure that *any* total order of this partial order of operators will solve a new problem. Now, consider using the generalized plan on the problem shown in Figure 27. It is possible to instantiate the generalized plan to solve the new problem by binding the variables *?-W* and *?-Z* both as *B*. However, this plan fails unless a particular total order is imposed on the operators. Kambhampati and Kedar think that the generalized plan should be able to solve the new problem, but doing so is not trivial. They state

> To avoid this problem, the EBG [EBL] algorithm needs to be more systematic in accounting for *all* possible interactions among operators corresponding to *all* possible total orders consistent with the partial ordering. There are two options for doing this. One is to *modify the algorithm:* For instance, repeatedly compute the weakest conditions of all total orders of the partial order and then conjoin them in some way. Another option is to *modify the input:* Provide a full explanation of correctness of the instantiated partially ordered plan, and use that explanation to produce the correct generalized initial conditions for the generalized partially ordered plan. (Kambhampati and Kedar, 1991)

Kambhampati and Kedar chose the second approach. They present an interesting and complex technique based on Chapman's modal truth criterion to guarantee that generalized plans will be applicable despite the total order chosen.

HINGE solves this problem much more simply by assuming that the generalized plan in Figure 26 should not be used in the new problem in Figure 27. HINGE's approach to guaranteeing that generalized plans work for any total ordering of operators is to use variable separation constraints. For example, in HINGE, the generalized plan in Figure 26 would also include constraints that the variable *?-W* cannot be instantiated to the same object as is the variable *?-Z* and that the variable *?-X* cannot be instantiated to be the same object as the variable *?-Y*. In HINGE, even for generative planning, the domain theory necessary to properly instantiate operator schemas is embodied in variable separation constraints attached

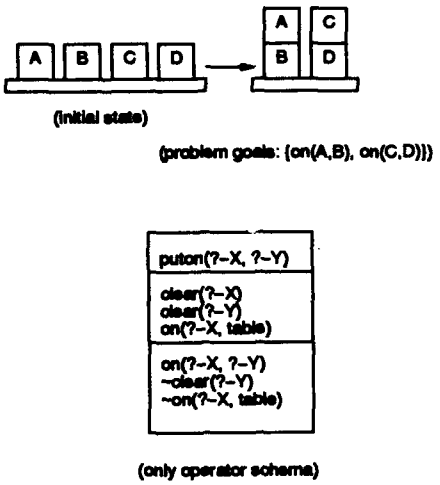


Figure 24. A problem for a partial-order planner.

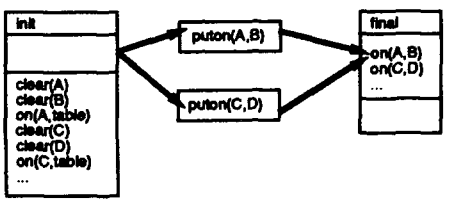


Figure 25. The resulting partially ordered plan.

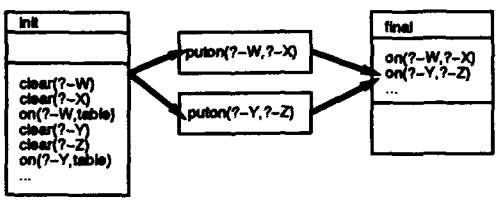


Figure 26. The resulting generalized plan.

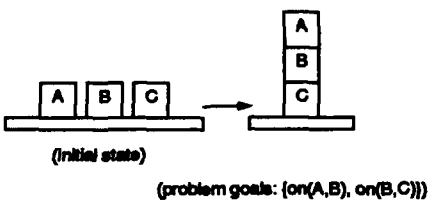


Figure 27. A problem for which the generalized plan seems appropriate but fails when A is put on B first.

to operator schemas.[11] For example, if an operator schema has the precondition *on(?-X,?-Y)*, then the domain theory requires that that *?-X* and *?-Y* must be instantiated to a different objects. Using variable separation constraints to restrict the possible misuse of generalized plans is a simple solution which requires no new algorithm. This simple solution avoids much work at the expense of some generality during reuse.

3.8.2 Learning Filters. Because they potentially require excessive space to store and because they are infrequently useful, HINGE does not learn macros whose primitive operators are unrelated. If the plan represented by a macro M has any two primitive operators which are unrelated, then the plan has independent sub-plans and in HINGE, M is not learned. Instead, HINGE learns any of the independent sub-plans that include multiple primitive operators and represents these sub-plans as macros. Suppose HINGE learns n macros corresponding to the independent sub-plans with multiple primitive operators. Now consider this question: why should HINGE learn only the n macros and not M? The answer is that there are potentially a large number of macros like M that represent plans with independent sub-plans and these macros are unlikely to be reused. There are potentially a large number of macros like M because the n macros of independent sub-plans (and the other macros like them already in the library) may be combined in an enormous number of ways. Each of these combinations solves a specific, large problem, and the probability of encountering this problem is small (assuming a somewhat uniform distribution of problems). Rather than learn and reuse macros like M, HINGE relies on decomposition methods described in Section 3.7 to break up large problems into their independent sub-problems which can be solved with known macros.

HINGE also filters any macro that does not appear to be useful for saving search. Including a macro in the set of candidates considered expands the search space. In some cases such as when the macro is small, the macro does not save enough search to pay for the expanded search space it creates. To identify these unproductive macros, HINGE tests the

[11] HINGE operators have more attributes than a name, preconditions, and effects. These additional attributes are useful in the planning process, but they are not fundamental to describe planning models or characteristics of plans. Therefore, these attributes have not been described to simplify the discussion.

training problem to see if having the macro operator schema in the library results in faster planning than not having it in the library. If performance does not improve, the macro is discarded. Importantly, this filter is *not* based on the relative amount of search saved; instead, all macros that save search on the training problem are learned.

The learner in HINGE also has a filter to keep duplicate macro operator schemas out of the library. Learning duplicate concepts does not affect the function of the library or macro operator schemas retrieved from it during the course of planning. However, search-space expansion and macro testing costs are minimized when duplicate operator schemas are avoided.

Unlike MORRIS, PRODIGY, and similar problem solvers that rely on selective learning, HINGE learns all search-saving macros that correspond to independent problems. HINGE uses filters to avoid learning useless macros, but it does not use filters based on utility in future planning. For example, PRODIGY uses a selective filter based on (Minton, 1990):

1. a prediction about the relative probability that macros will be used in future problem solving
2. the relative amount of search saved, and
3. the relative cost of retrieval.

HINGE uses none of these as filter criteria because HINGE does not learn selectively.

3.9 Reuse allowed in HINGE

HINGE's plan-space planning model is primarily responsible for HINGE's ability to reuse macros more flexibly than state-space macro planners like MORRIS. As discussed in Section 2.4.3, HINGE, like SNLP+EBL, can insert macro operators anywhere in the plan. Kambhampati and Chen have concluded that the ability to insert macros anywhere in the plan allows reuse precluded by state-space planners (Kambhampati and Chen, 1993). HINGE and SNLP+EBL, developed at about the same time, were the first macro planners with this ability.

Both SNLP+EBL and HINGE can expand two macros and, ordering constraints permitting, interleave the corresponding primitive operators in order to solve a problem. This ability allows HINGE and SNLP+EBL to reuse macros that would otherwise conflict with one another. (Kambhampati and Chen, 1993)

Like other macro planners, HINGE also has a more flexible reuse policy than case-based planners which allow only one reuse candidate and will not reuse candidates below a certain size. Unlike case-based planners, HINGE can decompose a set of goals which cannot be achieved directly and reuse multiple solutions from previously encountered problems.

Unlike other macro planners, HINGE explicitly represents certain subsets of the outstanding goals to direct the search for reuse candidates. This ability, supported by decomposition methods, allows HINGE to specify and find good reuse candidates much faster than when outstanding goals are stored in a set.

3.10 Summary

HINGE is a macro planner whose flexible macro insertion capabilities and unique search strategy promote reuse and facilitate learning for improved planning performance. HINGE uses abstract operators to represent a set of goals to be solved simultaneously in the plan. The overall search control in HINGE is depth-first with chronological backtracking. To focus attention on high-level goals, abstract operators are selected based on their depth in the plan tree. HINGE seeks reuse candidates that simultaneously achieve the goals of abstract operators. When no such candidates are possible, HINGE decomposes the goals represented by the abstract operator to maximize the opportunities for reuse, eventually solving the problem generatively when no reuse is possible. In this way, HINGE reuses the most powerful macros first and retains completeness. HINGE's conventional learning module was made suitable for partial-order planning by the relatively simple method of posting variable separation constraints, rather than the complex method proposed by Kambhampati and Kedar (Kambhampati and Kedar, 1991). By virtue of its plan-space planning model, HINGE and SNLP+EBL are the first macro planners that can solve any problem by reusing available

macros, even when those macros must be expanded and interleaved to integrate sub-problem solutions.

IV. Addressing the Utility Problem

4.1 Introduction

The utility problem is the possibility that a problem solver which learns and reuses solutions will perform worse with additional learning. Previous research has focused on some of the costs associated with reusing plans. In my research, I treat the utility problem using an eclectic approach, paying attention to both costs and benefits of reusing plans. Chapter III described HINGE and the features of HINGE that promote beneficial reuse such as its plan-space planning model and its hierarchical search method. This chapter focus only on utility problem costs. In particular, this chapter describes two methods that strictly contain the costs of reuse: an efficient retrieval method and a policy of selective reuse. The chapter begins by analyzing the utility problem and previous planning methods that motivate selective reuse. Section 4.3 defines selective reuse and an important implication of selective reuse. Because selective reuse depends on efficiently finding a specific reuse candidate in the library, Section 4.4 describes the development of a discriminating retrieval method that takes advantage of complete problem information. Section 4.5 shows that selective reuse limits both search and the expansion of the search space better than other reuse policies. Section 4.6 shows how selective reuse can be relaxed to solve problems by decomposition using reuse candidates which are likely to be appropriate and to require no future work.

4.2 Analyzing the Utility Problem

4.2.1 Separate Costs. The which may be separated into these components:

1. the cost of retrieving, from the library, reuse candidates that are likely (or guaranteed) to be appropriate (i.e., to be on a solution path).
2. the cost of selecting from among the retrieved reuse candidates
3. the cost of doing future work which might arise from trying to reuse the best candidate; this cost may be further broken down:

(a) if the selected reuse candidate is appropriate, there is only the cost of search required to establish any of the candidate's preconditions that are not established by existing plan operators; otherwise,

(b) if the selected reuse candidate is inappropriate, there is the cost of search required to discover that its preconditions cannot all be established plus the cost of finding and testing an appropriate reuse candidate.

4.2.2 Previous Methods for Containing Costs. As described in Chapter II, selective learning has been the method of choice for containing the utility problem. Selective learning limits retrieval cost directly by constraining the size of the library (and therefore, the search required to find a particular reuse candidate in the library). By limiting the number of reuse candidates available, selective learning also limits the expansion of the search space to some extent. However, the primary motivation for selective learning is limiting the retrieval cost.

In the literature, retrieval has been characterized as requiring general search and matching. Because matching and search are supposedly required, the cost of retrieval has been claimed to grow exponentially with both the size of the learned macros and the number of them in a library, despite any indexing method used (Minton, 1990). However, these characterizations are not necessarily true. For example, it is possible to find a polynomial algorithm for retrieving appropriate macros, as shown in Section 4.4.

Chapter II also described case-based methods which address the utility problem. Case-based planners usually have efficient retrieval methods that do not require a general search algorithm. Therefore, case-based planners do not depend on selective learning to retrieve a reuse candidate quickly. To contain the expansion of the search space caused by considering additional operators, case-based planners typically limit reuse to a single macro which is modified to solve the current problem. There are three drawbacks associated with this approach:

1. the retrieved reuse candidate is only *heuristically similar* to the desired plan; there is no guarantee of appropriateness,

2. allowing only one reuse candidate means that new problems solvable using case-based planning are those which are similar to previously-solved problems,[1] and

3. it is impossible to predict the amount of additional work required of the planner to modify the reuse candidate so that it is a solution to the current problem.

4.3 Selective Reuse

Considering the costs associated with plan reuse (Section 4.2.1), the best method of containing costs is to retrieve only appropriate reuse candidates so long as the cost of doing so is not too high. If the retrieved reuse candidates are all guaranteed to be appropriate, then selecting one of them costs virtually nothing because the choice is arbitrary; inserting any one of them into a plan will lead to a solution.[2] The future work entailed by inserting an *appropriate* operator is the work done by the planner to establish any of its preconditions that cannot be established by existing plan operators. In contrast, if an *inappropriate* operator is inserted into the plan, then its insertion entails both the work of establishing operator preconditions (until one is found that cannot be established) and the additional work done by the planner to backtrack and find an appropriate operator. In general, the future work entailed by inserting either type of reuse candidate cannot be predicted.

If retrieving only appropriate reuse candidates is the best method of containing the costs of plan reuse, then how can appropriate reuse candidates be identified? Assuming no special domain knowledge for identifying appropriate reuse candidates, there are only two ways to identify appropriate reuse candidates. The first way is to test them by inserting them into a plan and using the planner itself. If inserting a reuse candidate into a plan does not cause backtracking during the search for a solution, then the reuse candidate must have been on a solution path, i.e., appropriate. Clearly, this test is of no value because it requires finding a

[1] Veloso's PRODIGY/ANALOGY allows multiple reuse candidates. However, because it depends on a decomposition method similar to big-chunk-decomposition (Section 3.7.3), PRODIGY/ANALOGY cannot efficiently solve problems that must be decomposed into same-size chunks. Therefore, in some sense PRODIGY/ANALOGY is also limited to solving problems that are similar to problems solved previously.

[2] Of course, selecting one appropriate reuse candidate from a set of them could be based on another criterion such as execution cost or side-effect preference.

solution; the appropriateness of a reuse candidate is needed before it is inserted into a plan, not when the planner solves the problem.

The second way to identify appropriate reuse candidates is to use a description of the problem solved by the reuse candidate and compare it with the current planning problem. The reuse candidate solves the current planning problem if

1. the effects of the reuse candidate achieve all problem goals, and
2. the preconditions of the reuse candidate are a subset of the initial state

or, in plan-space terms, if

1. the effects of the reuse candidate can establish the preconditions of *final*, and
2. all of the preconditions of the reuse candidate can be established by *init*.

Any reuse candidate that meets these criteria solves the *whole* problem and is therefore called a "whole-problem solution." Inserting such a reuse candidate into a plan clearly leads to a solution, thus such a reuse candidate is appropriate by definition. Because there is no general method for predicting the appropriateness of reuse candidates that are not whole-problem solutions, the following result holds:

> **Synonymous Terms:** In the context of a particular planning problem, an *appropriate reuse candidate* means the same thing as a *whole-problem solution*.[3]

There are two important implications that arise because appropriate reuse candidates are also whole-problem solutions. First, it becomes quite clear how appropriate reuse candidates can be retrieved from a library and what information is required for retrieval. Reuse candidates are described by the problem they solve; the initial state is represented by the macro's preconditions and the problem goals achieved by the macro are represented by its effects. Retrieving a candidate requires an index that captures macro preconditions and effects and

[3]The HINGE planning model assumes that no domain knowledge exists to guarantee the appropriateness of a reuse candidate. Another planner may not make this assumption, so these two terms may not by synonymous in the context of this other planner.

this same index must be generated by a new problem which the macro can solve. To generate the right indexing function, complete information about the new problem is required. That is, its initial state and problem goals must be completely specified. With these conditions and the appropriate index, efficient retrieval is possible. Sections 4.3.1 and 4.4 address indexing and develop an efficient retrieval method for finding only appropriate reuse candidates.

The second implication from the equivalence of "appropriate reuse candidate" and "whole-problem solution" is that the following simple result holds:

> **Future Work Required by Appropriate Macros:** Inserting an *appropriate reuse candidate* (a whole-problem solution) into a plan entails no future planning work.

The costs associated with the utility problem are contained the most when reuse is confined to only appropriate reuse candidates. This is the basis of selective reuse.

> **Selective Reuse:** The policy of retrieving and inserting only one *appropriate reuse candidate* into a plan.

Clearly, if selective reuse only allows reuse of whole-problem solutions, then it is not interesting when a whole-problem solution does not exist in the library. However, a relaxed form of selective reuse can be applied to sub-problems which result from decomposing a planning problem. This relaxed form of selective reuse uses incomplete problem information, but assumes that the information is complete. The method is described in Section 4.6. Before analyzing selective reuse further, it is important to describe the retrieval method which makes selective reuse a viable method of controlling the utility problem.

4.3.1 "Impedance Mismatch" in an Index. Part of the difficulty in finding a discriminating index even with sufficient information is that the problem solver and learner often have different "views" of the information. These different views lead to a potential "impedance mismatch" when making the index for a particular concept. That is, the index that the learner uses to store the concept may be quite different from the index that the problem solver will use to try to retrieve the same concept. This effect is easy to see in macro planners. A macro often has fewer preconditions than the initial state of a problem that the macro solves

(if so, then part of the initial state is not required to establish preconditions in the macro). Therefore, an impedance mismatch results if the index tries to match the initial state with preconditions of reuse candidates. Impedance mismatch is a recognized problem in other EBL systems also (Keller, 1987, Minton, 1990). Minton notes:

> "...as traditionally viewed, the operationality criterion does not take into account how the learned description will be used later to improve the performance [of the] system, which determines its benefit (Minton, 1990)."

To avoid impedance mismatch in constructing an index, a careful analysis of the domain must be used to find ways to translate problem descriptions and learned-solution descriptions into identical indices. In Section 4.4, such an analysis is described for the block-stacking domain. Given an appropriate index, the polynomial retrieval method described in Section 4.4.3 may be used.

4.4 Retrieval: An Extended Example

This section presents an extended example that develops an index and polynomial retrieval method based on selective reuse. Section 4.4.1 describes an indexing scheme for linear-order retrieval of macro operators that completely solve problems in a constrained STRIPS block-stacking planning problem. Unlike general search and matching methods, the indexing and retrieval method does not require an exponential amount of time to find macros despite the size of the macros or the number learned. This example shows that retrieval does not require general (exponential) search when complete problem information exists, and only constrained search is required when incomplete problem information exists.

In Section 4.4.2, the effects of removing some of the assumptions in the example problem are discussed along with the space/time trade-offs that exist. As shown in Section 4.4.3, in the case that the impedance mismatch cannot be eliminated, a polynomial-time algorithm may exist for finding a reuse candidate that is guaranteed to be appropriate. In Section 5.3, empirical results are presented that suggest problem solving performance degrades linearly with the size of the library in the worst case.

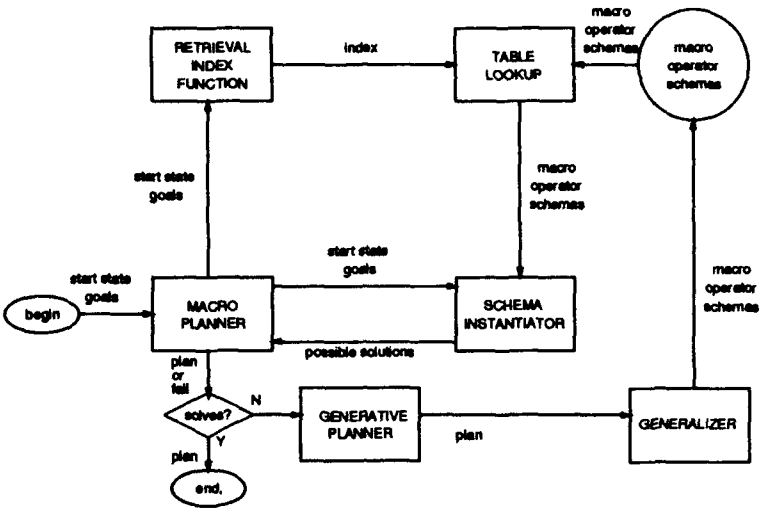


Figure 28. The retrieval process for a macro planner.

4.4.1 Solving the Utility Problem in a Blocks World. Figure 28 shows the retrieval process of a macro planner. At the beginning of the planning process, the planner has a set of goals to be achieved and a specification of the initial state. Using this information, a retrieval index function returns an index that is used to retrieve a set of macro operator schemas (plan schemas) that might achieve the goals, given the current state. The retrieved macro operator schemas are tested until an instantiation of one of them solves the problem. If none of the macro operator schemas can be used to solve the problem, then the macro-based planner fails and a solution must be generated from primitive operators. Whenever such a failure occurs, planning with reuse takes longer than generative planning. On the other hand, if it can be guaranteed that all retrieved macro operator schemas can be instantiated to solve the problem (i.e., the index is sufficiently discriminating to find appropriate reuse candidates), then no testing is required, and failure of the retrieval mechanism bears only a small cost. With the best hashing methods, the worst-case time order of retrieval is $\mathcal{O}(n)$ in the number of stored macro operator schemas (Horowitz and Sahni, 1990). Therefore, the time complexity of the best retrieval algorithm is no worse than linear in the number of macro operator schemas.

If the macro planner fails, a complete, generative planner produces a plan that achieves the goals, given the initial state. Whenever a plan is generated, a generalization module creates

the macro operator schema by substituting variables for objects and object attribute values. A storage index function is used to store the new macro operator schema. In order to find an indexing scheme that overcomes the utility problem, it is necessary to specify the index functions used to store and retrieve macro operator schemas and show that the retrieval index allows retrieval of only those schemas that can be instantiated to solve a particular planning problem. This is demonstrated below with a tightly constrained version of the block-stacking domain.

In this constrained block-stacking domain, six blocks may be arranged only in the configurations shown in Figure 29: this will hold for initial, goal, and intermediate states. Also, the position of stacks on the table is part of the state description. Each of the 11 configurations may be arranged in 6! different ways, so the total number of states for this world is $7,920$. There are $62,726,400$ ($7,920^2$) state transitions ($7,920$ of them "null" transitions) and an infinite number of plans possible for each state transition. The primitive operators are defined to perform these transitions in a single step. One more simplifying assumption is required to find a completely discriminating indexing scheme: the set of goals provided to the planner specifies the goal state, rather than a set of propositions the goal state must include. In other words, a problem is described by an initial state and a goal state rather than an initial state and a set of problem goals. The initial and goal states describe the planning problem and also describe the "interface characteristics" (I/O) of plans derived to solve the planning problem; hence, there will be no impedance mismatch. Similarly, a generalization of the two states defines a generalized planning problem and also describes the interface characteristics of a generalized plan for solving it. Therefore, the retrieval index function maps a planning problem into a generalized planning problem index, and the storage index function returns an index for a generalized planning problem.

Figure 30 (a) shows an example of a transition between a generalized initial state and a generalized goal state. Figure 30 (b) shows the same transition with a different set of variable assignments. For space efficiency, the index function should map one transition pattern into a single index, regardless of the variable names. One way to do so is to label the initial state

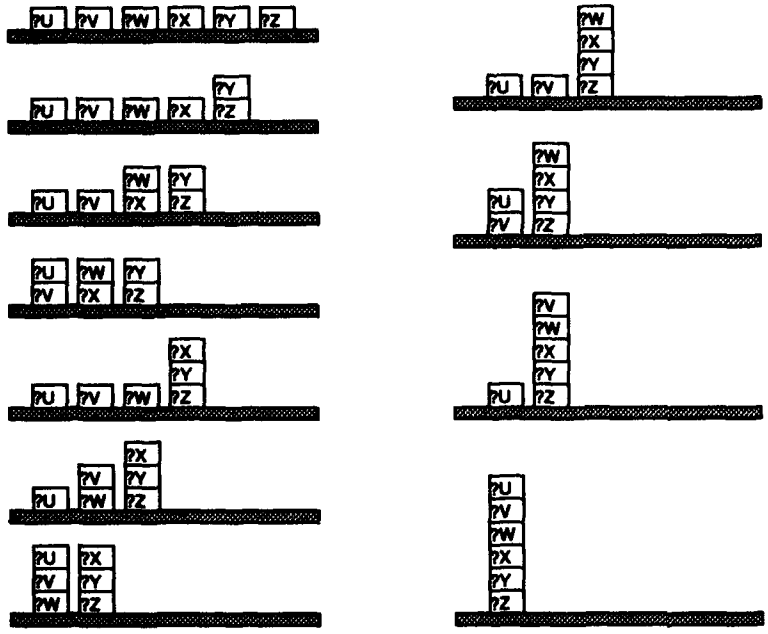


Figure 29. Allowed block-stacking configurations and initial state variable labeling convention.

with variables, using the convention shown in Figure 29. Figure 30 (c) shows the appropriate transition representation with this convention.

Now the index functions can be specified. For learning, the storage index function maps a transition between generalized states into two 6x6 arrays whose elements are either empty or contain the name of a variable representing a block. Figure 31 shows, in graphical form, the resulting index for the macro represented by the transition shown in Figure 30. For planning, the retrieval index function generalizes the initial and goal states and then applies the storage index function.

As in many problems, there can be an infinite number of plans possible for each of the state transitions (albeit, most of them are inefficient). However, for retrieval, only the generalized interface characteristics are of concern. In the example world, there are only 87,120 (62,726,400/6!) transitions between generalized states because each transition may be instantiated 6! ways. Thus, generalization saves considerable space (for a small cost in time) by mapping the given problem to a generalized problem and then instantiating

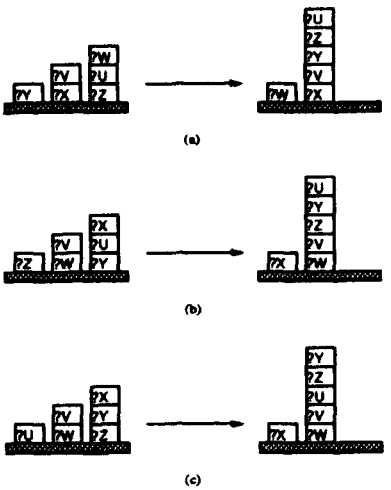


Figure 30. Alternative variable labeling for block-stacking transitions.

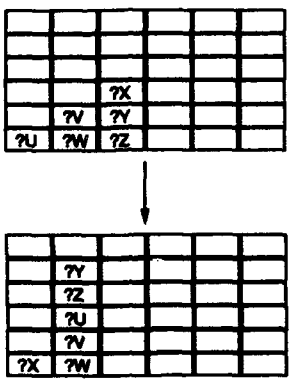


Figure 31. An index for the macro operator schemas.

the retrieved macro operator schemas to produce operators. Furthermore, if a sufficiently discriminating index is available, there is no cost of testing macro operator schemas for applicability. When a set of schemas is stored under an index corresponding to a single transition, the implication of using a sufficiently discriminating index is profound: all the schemas are retrieved simultaneously and are guaranteed to be appropriate. No testing is required, selection amongst them can be based upon other criteria (such as execution cost), and the retrieval of the multiple plans does not add to the cost of reuse.

4.4.2 Expanding the Blocks World. The previous example used two important assumptions: relative stack position is meaningful, and the goal state description is completely

specified. Relaxing these limitations will allow for more flexible problem solving while still retaining linear-order retrieval, provided a storage space penalty is accepted.

By making the relative stack position important, the first assumption restricts the allowable block configurations. This assumption provides for a very compact index and avoids storing a macro operator schema under multiple indices. However, if stack order is not important, a different approach to indexing is required. A single index cannot accommodate all permutations of a configuration when there are two or more stacks of the same height. Therefore, multiple indices are required, leading to a space/time trade-off: either a macro operator schema must be stored under multiple indices during learning, or multiple indices must be generated during planning, causing multiple retrievals. Assuming retrieval time is to be minimized, the best choice is to store macro operator schemas under multiple indices during learning; with this method, the time for retrieval is unaffected by removing the first assumption.

The second assumption requires that the planner be provided with a complete goal (final) state description rather than a set of goals that must be true in the final state. Under the indexing method described so far, if this assumption were to be relaxed, then retrieval would require that indices be generated for all possible final states that include the given set of problem goals. A better approach is to modify the indexing method to use the initial state and the problem goals when a schema is learned, rather than the initial state and the goal state. These problem goals become the *primary effects* of the learned macro operator. A space cost could be incurred if the problem goals are not minimally specified in the learning problems (i.e., side-effects are included in the problem descriptions), but this has no impact on retrieval time. The usefulness of these learning situations is questionable, however, since macro operator schemas are retrieved based upon exact match of the interface characteristics (generalized initial state and problem goals/primary effects). A discipline of only specifying the problem goals would prevent storage space growth and provide a better hit-rate on schema retrieval.

4.4.3 Polynomial-Time Retrieval. A particular macro operator often requires only a subset of the initial state propositions to satisfy its preconditions. Thus, from a space standpoint, it is attractive to reduce redundancy and index the macro operator schema on its preconditions, rather than on every superset of the preconditions that represents an initial state. This approach, however, produces an impedance mismatch that precludes linear-time retrieval: a problem description provides complete information, but the macro operator schemas are stored using only the necessary preconditions under which they were learned. Therefore, a two-step process is used for retrieval. Indexing is based only on goals that an operator schema achieves (its primary effects). To retrieve appropriate reuse candidates, possible candidates are retrieved based on the goals they achieve and then the candidates are tested to see if all their preconditions can be established by the problem initial state. Only those passing the test are forwarded to the planner.

When a macro operator schema is learned, the storage index is based only on its *primary* effects. For retrieval, the problem goals are used to form the index; therefore, there is no impedance mismatch and the retrieval time is, at worst, linear with the size of the library. All schemas that achieve those goals will be retrieved; let the number of retrieved schemas be M. Each schema will have a set of preconditions; let schema i have N_i preconditio[illegible]inally, the initial state of a problem has L propositions; the set of N_i preconditions may or may not be a subset of the initial state. When instantiated, a macro operator schema's N_i preconditions can be established if all preconditions are a subset of the L propositions of the initial state. Each precondition of each schema must be tested against the problem initial state, resulting in a maximum of $\sum_{i=1}^{M} L * N_i$ comparisons. Since N_i cannot be greater than L for an appropriate schema, the number of tests is upper-bounded by $M * L^2$. Therefore macro testing requires time of order $\mathcal{O}(n^3)$. Because the hash table lookup requires, at worst, $\mathcal{O}(M)$ time, the time complexity of finding *appropriate* macro operator schemas is a polynomial of order $\mathcal{O}(n^3)$.

Section 5.3 describes an experiment in which HINGE solves problems by selective reuse using the $\mathcal{O}(n^3)$ time retrieval method discussed here. In the experiment, the number of macro operator schema preconditions and the number of initial state propositions are fixed

while the number of macro operator schemas increases. The results are consistent with the analysis presented here: in the worst case, increasing the library size causes a performance degradation that is linear in the size of the library.

4.5 Analyzing Selective Reuse

There are alternative reuse policies that could be followed in a macro planner. However, to contain the utility problem, selective reuse (or the relaxed form of selective reuse) is the best policy because it most strictly limits search and contains search-space expansion. Furthermore, if reuse candidates are guaranteed to be appropriate (solve the whole problem), then the planner can consider just one of them, or, by extension, if reuse candidates are likely to be appropriate, then reuse will probably succeed if the planner limits consideration to a few of them. By considering only one or a few of the reuse candidates, the size of the search space is constrained further. These are the fundamental ideas of selective reuse, a reuse policy designed to contain the utility problem in macro planning.

Alternatives to reuse candidates that solve the whole problem are:

1. macros whose effects can achieve all problem goals but whose preconditions cannot all be established by existing plan operators,
2. macros whose preconditions can all be established by existing plan operators but whose effects cannot achieve all problem goals, and
3. macros whose preconditions cannot all be established by existing plan operators and whose effects cannot not achieve all problem goals.

Considering any or all of these reuse candidates in addition to those that solve the whole problem expands the search space being traversed by the planner because there are more choices available. Furthermore, all three alternatives require additional search, while reuse candidates that solve the whole problem do not. Inserting a macro of the first type adds goals for precondition establishment to the set of outstanding goals being considered by the planner. These new goals will also have to be achieved through search, and the amount of

search necessary to achieve them is unknown and unconstrained. Inserting a macro of the second type means that the choice may have to be retracted, resulting in backtracking, lost search effort, and additional search. For example, if *stack(A,B)* were a macro for Sussman's Anomaly (see Figure 15 on page 53 and Figure 7 on page 20), inserting it would not lead to a solution and backtracking would result. Inserting a macro of the third type would require additional search for both reasons. Because considering any of these alternatives in addition to whole-problem solutions expands the search space more and causes additional search, any policy that considers them is not as useful for limiting the utility problem as selective reuse.

4.6 Solving Problems Incrementally with Relaxed Selective Reuse

Section 4.4 described an example that depended on finding a whole-problem solution (a macro that is guaranteed to be appropriate and also solves the whole problem). Often, a whole-problem solution does not exist in the library of reuse candidates. If so, reuse may still be possible by decomposing the problem and finding a set of appropriate macros that solve pieces of it. To test for macro appropriateness, however, the planner itself must generally be used to solve the problem, making the test as expensive as solving the planning problem itself. A less-expensive strategy is to find macros that are very likely to be appropriate. For example, perhaps it is possible to identify parts of the problem which are likely to be independent. If so, then macros which are used to solve these subproblems will probably not be clobberers of each other's establishments, and using them is more likely to result in a solution. Under these assumptions, the macros found using the retrieval method presented in Section 4.4.3 are likely, rather than guaranteed, to be appropriate because sub-problem solutions may fail to integrate. This is the basis of the relaxed form of selective reuse.

> **Selective Reuse (relaxed form):** The policy of considering only a few reuse candidates which are appropriate for solving sub-problems of a problem for which no whole-problem solution can be found.

Solving a sub-problem increases the number of propositions that possibly match macro operator preconditions when testing for applicability, but retrieval still requires $\mathcal{O}(n^3)$ time. As described in Section 4.4.3, each learned macro operator schema is stored using an index

based only on its primary effects. There is no change in library retrieval when hashing is used: the problem goals are used to form the index; therefore, there is no impedance mismatch and the retrieval time is, at worst, linear with the size of the library. As before, all schemas that achieve those goals will be retrieved; let the number of retrieved schemas be M. Each schema will have a set of preconditions; let schema i have N_i preconditions. As before, the initial state has L propositions, the set of which may or may not be a superset of the N_i preconditions. However, because other plan operators may be able to establish the macro's preconditions, the effects of these operators must be added to the initial state propositions when testing to see if the macro preconditions can be established without additional work. Suppose there are K effects corresponding to other plan operators that may be able to establish the macro's preconditions. When instantiated, a macro operator schema's N_i preconditions can be established if they are a subset of the L propositions of the initial state unioned with the K effects of possibly-before operators. Suppose there are J of these propositions and effects, where $J \geq L$ always. Substituting J for L in the argument made in Section 4.4.3 shows that the time complexity of this approach to macro operator schema retrieval is, at worst, a procedure whose time complexity is $\mathcal{O}(n^3)$, as before.

4.7 The Impact of Efficient Retrieval and Selective Reuse on the Utility Problem

Efficient retrieval and selective reuse are effective strategies for strictly limiting performance degradation that possibly accompanies reuse, but these methods do not eliminate the utility problem.[4] Whenever an appropriate reuse candidate does not exist, the effort spent looking for an appropriate candidate will be wasted and macro planning performance will be worse than generative planning. Efficient retrieval and a policy of selective reuse are designed to minimize the effort spent looking for an appropriate reuse candidate. These methods and the methods that promote beneficial reuse discussed in Chapter III combine to contain the utility problem and lessen its importance.

[4]The same is true for selective learning and case-based planning—these methods do not eliminate the utility problem either.

4.8 *Summary*

After analyzing the utility problem, this chapter described two methods for containing the utility problem in macro planning. First, it introduced the policy of selective reuse to control search and contain the expansion of the search space caused by considering more operators. Selective reuse causes the planner to consider only whole-problem solutions (appropriate macros) or, in its relaxed form, whole-problem solutions to sub-problems (likely-appropriate macros). Second, it presented a generally-applicable retrieval method for macro planning that finds an appropriate or likely-appropriate reuse candidate in $\mathcal{O}(n^3)$ time when complete problem information exists. The performance of this retrieval method is independent of the size of the macros in the library and varies only linearly with the size of the library. Selective reuse was shown to better contain search and search-space expansion better than alternative reuse strategies. A relaxed form of selective reuse was defined to allow reuse when no whole-problem solutions exist in the library of reuse candidates. Using these methods, HINGE effectively contains the utility problem without selective learning. The next chapter presents experimental results that are consistent with analytical arguments made in this chapter.

V. Empirical Results

5.1 Introduction

In Chapters III and IV established the computational advantages of the planning, retrieval, and reuse approaches developed for containing the utility problem. This chapter describes the computational results observed when these approaches were implemented in HINGE. In particular, this chapter concentrates on HINGE's performance when retrieving reuse candidates from large libraries, when using an alternative reuse policy, and when learning from random problems.

5.2 The Domain and Method Used

Given the operator representation used in this research and defined in Section 2.2.2, any domain that can be represented by HINGE operators could have been chosen to gain computational experience with HINGE. I chose to use the block-stacking domain for this purpose because block-stacking problems are familiar to many researchers and are simple to understand. The block-stacking domain is described in (Norvig, 1992:136-142).

To avoid bias in choosing a test set, I used only randomly-generated problems. I developed a random-problem generator that works by producing two states, using LISP's pseudo-random number generator to decide the placement of each block in each state. The random-problem generator selects the simpler state (the one with the greater number of smaller block stacks) as the problem's initial state. The problem goals are produced by randomly selecting propositions from the remaining state, which represents one of the possibly many goal states.

I measured performance in terms of the time HINGE required to find a plan under different conditions for 22 randomly-generated 6-block block-stacking problems. Planning time is reported either for each individual problem or as a mean planning time for all problems in the test set. A Sun IPC was used for some measurements while others were made using a Sun SPARCstation 10. Therefore, absolute planning times vary because of different processing

power of the different computers used. However, direct comparisons were only made between problem runs on the same processor.

5.3 The Effect of Large Libraries

To examine the worst-case performance of HINGE's retrieval method as a function of the size of the library, HINGE was repeatedly trained on the test set of 22 problems and performance with reuse was measured as a function of increasing library size. The polynomial retrieval method described in Section 4.4 was used at all times. As described in Chapter IV, this retrieval method uses a hash table; in the worst case, all hash table entries are stored in a list indexed by a single hash key. To retrieve a hash entry in the worst case, each element of the list must be examined, an operation that requires time that is a linear function of the length of the list. By repeatedly learning solutions to the same set of problems, HINGE's worst-case library is built and HINGE's worst-case retrieval performance can be measured.

During this measurement, HINGE was set to retrieve all appropriate reuse candidates, but only one of them of them was used[1]; under these conditions, any increase in planning time is due to the retrieval method finding additional reuse candidates. Thus, the retrieval-method performance was measured indirectly by measuring planning time.

HINGE was initially set to learn macros, but not allowed to reuse them. Instead, HINGE solved each problem in the test set by generative planning using only the three primitive operator schemas shown in Figure 15 on page 53. After solving the 22 problems, the operator schema library contained 28 macro operator schemas[2] as well as the original 3 primitive operator schemas. Next, HINGE was set to reuse these macro operator schemas, but not to learn new ones. Planning times were found for the test set problems while HINGE used this small library.

---

[1] Only one reuse candidate is required to solve a problem if the candidate is appropriate because it must be a whole-problem solution.

[2] The learner generalizes and stores all significant subproblems, so each problem resulted in learning a variable number of macro operator schemas.

To see the effect of increasing the library size on retrieval performance, the learning filter that normally prevents redundant macro operator schemas from being stored in the library was disabled, and the procedure described in the paragraph above was repeated 29 times. After each iteration, the same 28 macros were learned and stored in the library using the same set of indices derived on previous iterations. Thus, after five iterations, for example, a particular macro index pointed to at least 5 macro operator schemas.[3] Normally, HINGE does not learn a particular plan more than once.

Figure 32 shows the planning time required for generative planning to solve the 22 problems as well as the corresponding solution times for planning with reuse using four different sizes of the macro operator schema library. Data points that are associated with a particular set of conditions are connected by a line segments on the graphs.

For most problems, the planning time increases as the library grows because the retrieval method must test additional reuse candidates. The planning times for problems 11 and 12 did not vary as the library grew because the solutions for these problems were not learned. These two problems had phantom goals which required planning effort to assure, but the problem solutions did not involve multiple, related, concrete operators. (See Section 3.8.2 for a description of the filters used in HINGE's learning element.) As shown in Figure 32, the increase in planning time caused by retrieval from larger libraries varies with the problem. However, for each problem, the time required to plan with reuse is proportional to the size of the macro operator schema library. Figure 33 shows the increase in the mean planning time as a function of library size for each of thirty iterations. Because the library size was increased in the worst way, these results illustrate that, in the worst case and for the test set used, retrieval required time that is linear in the size of the library.

[3] A particular index might point to more than 5 macro operator schemas because the index is based only on operator schema effects. Some of the 22 problems result in learning macro operator schemas that have the same effects and indices, but different preconditions.

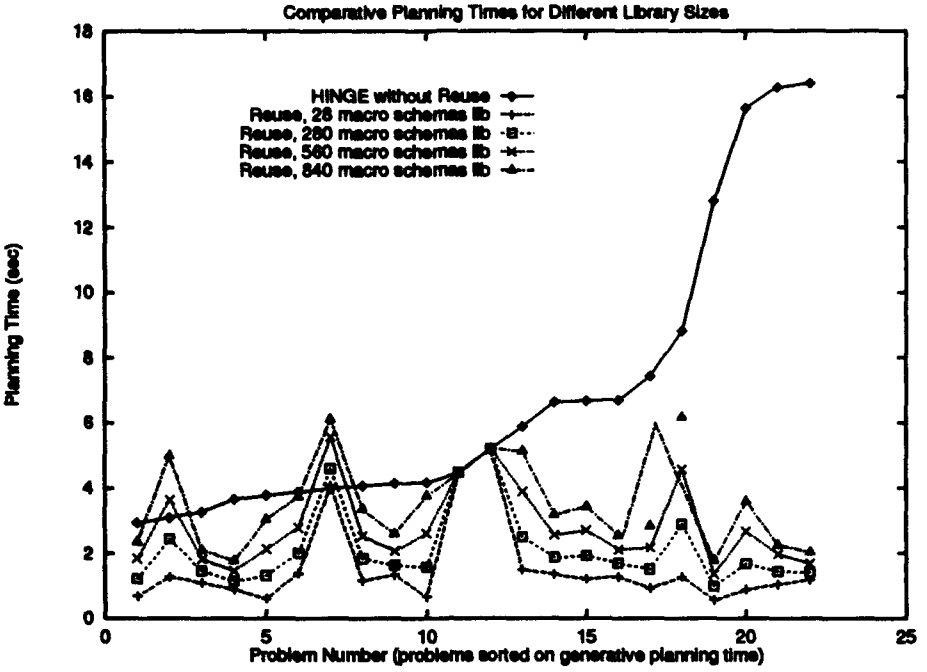


Figure 32. Comparative planning times with reuse at four library sizes.

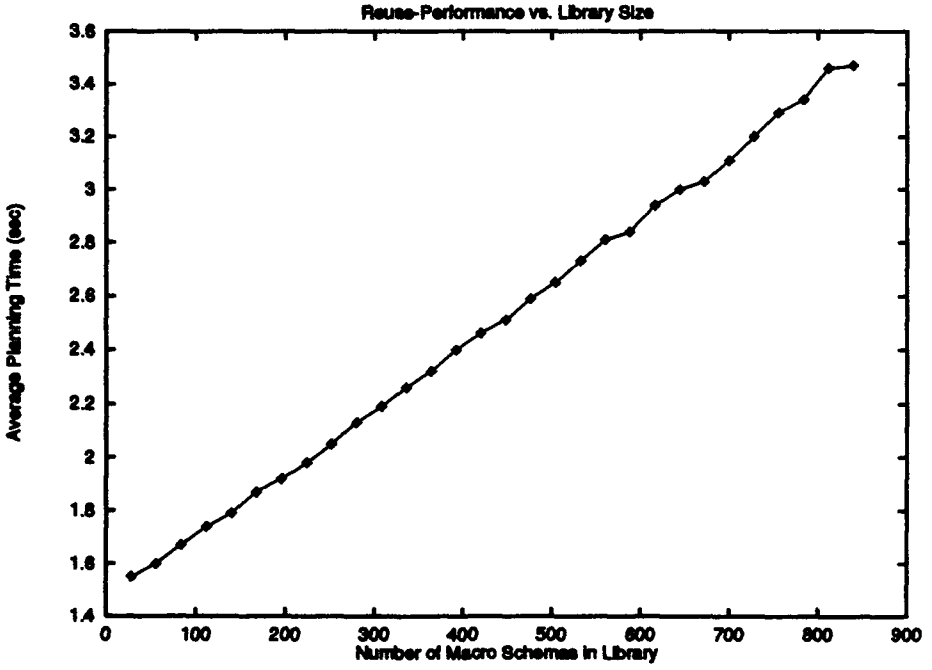


Figure 33. Average planning times with reuse at thirty library sizes.

According to the theoretical performance derived in Section 4.4, HINGE's retrieval method is, at worst, linear in the size of the library. The empirical results presented in this section is consistent with the derived theory.

5.4 Selective Reuse vs. Macros with Abstract Establishers

As discussed in Section 4.5, selective reuse reduces search and limits expansion of the search space expansion better than less selective reuse policies. One example of a less selective reuse policy is a policy that considers reductions whose operator preconditions cannot all be established by existing plan operators. Such preconditions must be established by an abstract operator in the reduction, and the abstract operator represents future work required of the planner. Performance data were collected when planning with reuse and allowing different numbers of abstract-operator-established preconditions. After training on the test set of 22 randomly-generated problems, HINGE was set to collect planning time data when 0, 1, 2, 3, and 4 abstract-operator-established preconditions were allowed in reduction operators. Selective reuse corresponds to allowing 0 abstract-operator-established preconditions.

Figure 34 shows that the average planning time for problems in the test set increased as the allowed number of abstract-operator-established preconditions was increased from 0 to 3. The planning time did not increase when the number of abstract-operator-established preconditions was increased from 3 to 4 because, for the test set used, no additional reuse candidates were considered as a result of this increase.

As described in Section 4.5, selective reuse is better than any of the three alternative reuse policies at limiting the expansion of the search space and search in the space. In particular, selective reuse reduces search more than a policy that considers reductions which include abstract operators because the latter policy requires additional planning work to establish preconditions. The results shown in Figure 34 are compatible with the analysis presented in Section 4.5.

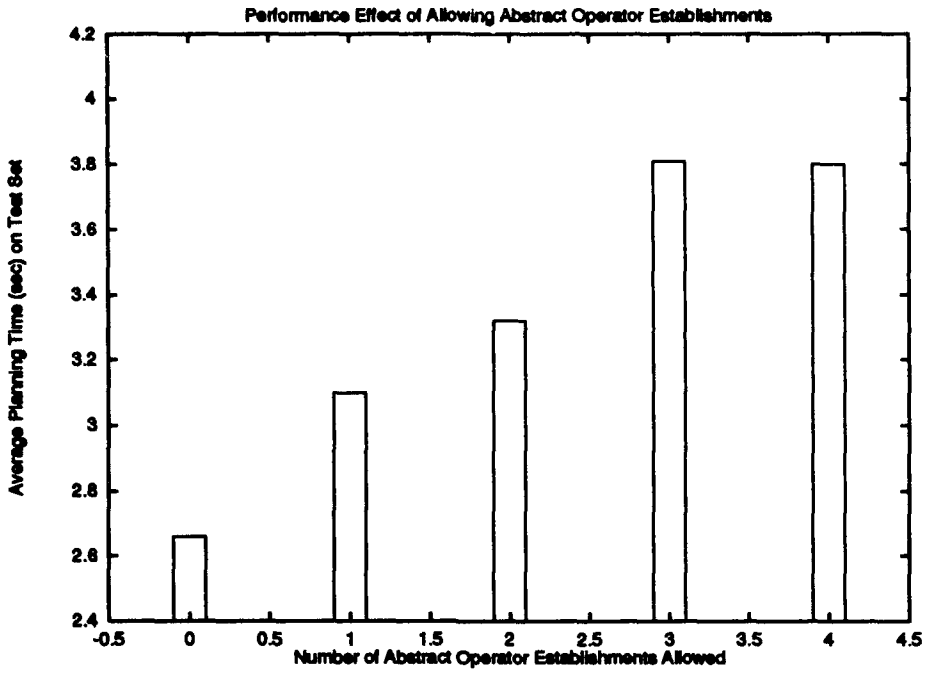


Figure 34. Performance as a function of the allowed number of new operator preconditions which cannot be established by existing plan operators.

5.5 Learning from Random Problems

A tenet of selective learning is that the utility problem makes non-selective learning on random problems a disastrous course of action. However, HINGE's efficient retrieval and selective reuse policy contain the costs associated with the utility problem and lessen the impact of having learned plans that are not useful for the current planning problem. Furthermore, HINGE's planning model and hierarchical search through successively finer decompositions allows flexible insertion of simple plans that may be learned from randomly generated problems.

To explore non-selective learning in HINGE 19 sets of 10 randomly-generated problems were solved and their solutions learned. After learning solutions for each set of randomly-generated problems, timing data were collected for planning with reuse on the 22 problems of the test set. Figure 35 shows that under these conditions, no problem in the test set required a great deal of extra time to solve.

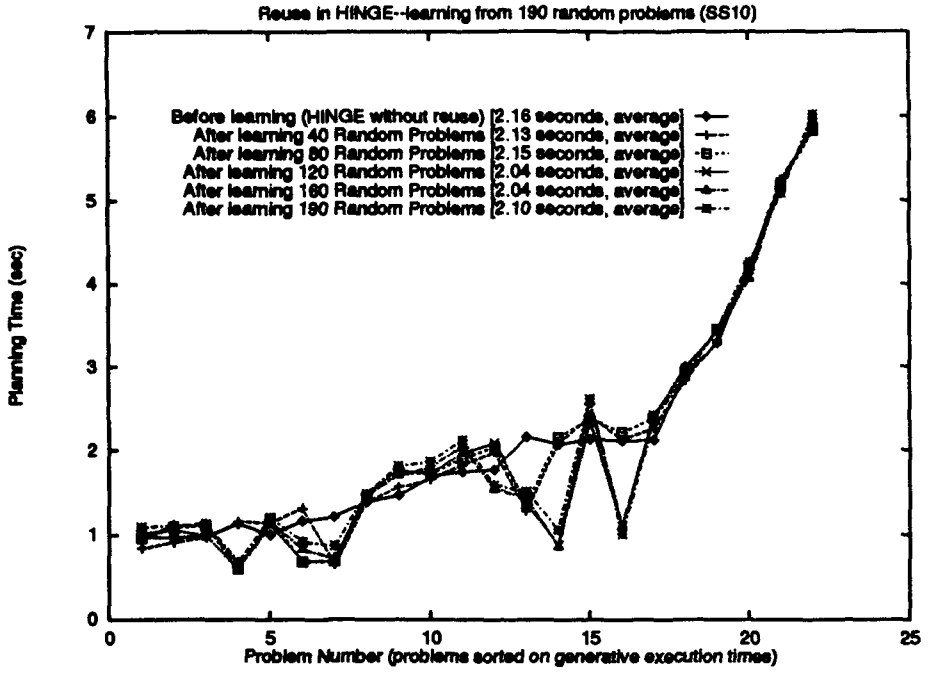


Figure 35. Performance improvement with reuse and different amounts of learning opportunities.

HINGE is designed to control the utility problem. HINGE's attributes have been shown analytically to contain the specific costs of the utility problem or improve the likelihood of reuse and its accompanying benefit. Allowing HINGE to learn 190 random problems did not drastically impact the performance of any one problem in the test set, a result that is in accord with the previous analysis of HINGE's features.

VI. Conclusions and Recommendations

6.1 Conclusions

This research describes a macro planning model and implementation designed to contain the utility problem. The utility problem is the possibility that the costs of reuse sometimes exceed its benefit. In this research, containing the utility problem depends on improving the probability of reuse and strictly limiting the costs that contribute to the utility problem. Because HINGE is based on a plan-space planning model, it can solve a wider variety of planning problems with reuse than state-space macro planners can. Furthermore, unlike typical case-based planners, HINGE and other macro planners are more flexible in terms of the number and granularity of plans that are reused. HINGE has a unique method of hierarchical search for reuse candidates which is supported by decomposition methods and increases the probability of reuse. To constrain costs which contribute to the utility problem, an efficient retrieval method was developed to reduce the retrieval cost associated with the utility problem. By design, this retrieval method requires time that varies linearly with three parameters: the number of reuse candidates in a library, the number of preconditions for macros resulting from retrieval of a reuse candidate, and the number of propositions in the initial state. A policy of selective reuse was developed to reduce search and limit the search-space expansion caused by considering reuse candidates in addition to primitive operators. Flexible macro insertion, hierarchical search providing multiple opportunities for reuse, efficient retrieval, and selective reuse combine to improve performance in HINGE. Together, these methods lessen the impact of the utility problem in macro planning without requiring selective learning and its characteristic potential for ignoring a plan with future value.

6.2 Specific Contributions

The novel planning ideas and contributions of this research are:

1. ***A model of planning and reuse that explicitly represents features of desirable macro operators and supports more flexible insertion of macro operators into the developing***

plan. In the planning model I developed, abstract operators represent goals which are used to index macro operators that achieve them. This ability to represent goals which should be achieved simultaneously by a macro operator is extremely useful for search control. When a good macro operator is retrieved, HINGE's plan-space planning model allows it to be inserted anywhere in the plan. This feature allows reuse that is precluded by state-space planning models and case-based planning models.

2. *A unique hierarchically-structured search control that encourages subproblem learning and efficient reuse while ensuring completeness.* HINGE searches for the most powerful reuse candidates first, using a hierarchically-structured search mechanism where abstraction levels are based on the number of goals that HINGE attempts to solve simultaneously. This hierarchical search is supported by several decomposition methods that allow abstract operator reduction when no appropriate or likely-appropriate reuse candidate can be found. HINGE does not differentiate between macros and primitive operators. The effect of these choices during planning is to facilitate opportunistic reuse of whole-problem solutions and flexible insertion of subproblem solutions when decomposition is necessary. If reuse fails, HINGE solves the problem using only primitive operators and a complete search strategy, meaning that HINGE always finds a solution to a planning problem if a solution exists. When learning, any sub-parts of the hierarchical search tree that correspond to useful knowledge are learned by HINGE's learning component.

3. *A polynomial-order procedure for retrieving only applicable macros, despite their size or number.* An important result of this research is the realization that, if complete problem information exists, it is possible to develop a discriminating index which can be used to efficiently retrieve only appropriate concepts learned by EBL (or any other learning technique), despite the size or number of the concepts learned. If a discriminating index is used and a whole-problem solution exists in the library of learned concepts, then retrieval costs that contribute to the utility problem can be less than the cost of general search and matching. For HINGE, this search requires $\mathcal{O}(n^3)$ time, while

the search corresponding to the planning itself requires exponential-order time. If no whole-problem solution exists in the library, these same results apply to sub-problem solutions except that the retrieved solutions are likely, rather than guaranteed, to be appropriate.

4. *A policy of selective reuse that limits the number of macros considered and restricts the future work they entail.* If macros that do not solve the whole problem are considered, the search space is expanded more than if only whole-problem solutions are considered. Whole problem solutions require no additional search, while the alternatives require an unpredictable amount of search. Furthermore, if multiple whole-problem solutions exist, there is no need to consider more than one of them, or, by extension, if multiple reuse candidates which are likely to be whole-problem solutions exist, then considering only one or a few of them is likely to lead to a solution. Limiting the number of reuse candidates considered is an effective method of limiting the search space. Therefore, in selective reuse, just one whole-problem solution is considered, or if appropriateness cannot be guaranteed, then a relaxed form of selective reuse only considers a few reuse candidates that are likely to be appropriate (for sub-problems). For domain-independent planning, any other policy re-introduces costs associated with the utility problem.

6.3 Recommended Future Work

There are three interesting extensions of this research, all involving search control. The first extension involves inserting partial domain knowledge using abstract operators. The second idea is to find new ways to use the library as a source of knowledge for finding decompositions. The third idea is to focus attention in planning by encapsulating parts of the plan when the details of those parts are not likely to be useful for establishing preconditions of new operators.

Abstract operators introduced in Chapter III provide a means for introducing partial domain knowledge into a plan when it is available. The method for doing so is to add known preconditions and effects to abstract operators. When preconditions are added to abstract

operators, these preconditions are established sooner during planning than they are when they only appear in the reduction of the abstract operator. Known effects may also be added to identify known side-effects that accompany the goals represented by a HINGE abstract operator. When known effects are added to an abstract operator, these effects may be used in establishments sooner or may give rise to additional ordering constraints sooner than if the effects do not appear until the abstract operator is reduced.

Using the library as a source of knowledge for finding decompositions appears to be fruitful, but more research is required. New indexing methods are potentially important for using the library in this way. More advanced techniques for learning appropriate decompositions from past problem solving are possible; these techniques probably require a redefinition of the library contents. Either of these extensions can co-exist with HINGE's current library and retrieval procedures.

Planning requires $\mathcal{O}(a^n)$ time for a plan of n operators (Bylander, 1991). If the number (and complexity) of operators in a plan can be reduced on-the-fly, planning can be more efficient. One way to reduce the number and complexity of operators in a plan is to encapsulate part of the plan in a macro operator and to replace that part with the macro. This encapsulation approach is effective when establishments for preconditions of new operators can be found in the macro effects or in the effects of other plan operators, but do not depend on the primitive operators encapsulated by the macro. The approach is likely to be effective when the part of the plan chosen for encapsulation is likely to be independent from the other parts of the plan. The complexity of macros that arise from encapsulation, in terms of the number of preconditions and effects, is sometimes much greater than for other operators, but in the case that many establishments are made internally, macro complexity differs little from that of other operators. Therefore, pieces of the developing plan that are likely to be independent and have many internal establishments are good candidates for replacement with a macro. It is worth mentioning that a particular operator, having failed to find establishments, might view such a macro in its expanded form to try to find an establishment, but this does not mean that other

operators need to do so. Encapsulation of parts of the plan with macros appears promising as a method of focusing attention during generative planning.

References

Alterman, 1986. Richard Alterman. An Adaptive Planner. In *Proceedings of the National Conference on Artificial Intelligence*, pages 65–69, 1986.

Barrett and Weld, 1991. Anthony Barrett and Daniel Weld. Partial order planning: Evaluating possible efficiency gains. Technical report, University of Washington, 1991.

Barrett *et al.*, 1991. Anthony Barrett, Steve Soderland, and Daniel Weld. The Effect of Step-Order Representations on Planning. Technical report, University of Washington, 1991.

Brown, 1990. Frank Markham Brown. *Boolean Reasoning: The Logic of Boolean Equations.* Kluwer Academic Publishers, Boston, 1990.

Bylander, 1991. Tom Bylander. Complexity Results for Planning. In *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence*, pages 274–279. Morgan Kaufmann, 1991.

Bylander, 1992. Tom Bylander. Complexity Results for Extended Planning. In James A. Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 20–27. Morgan Kaufmann, 1992.

Carbonell, 1983. Jaime G. Carbonell. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning*. Tioga Publishing Company, Palo Alto, CA, 1983.

Chapman, 1987. David Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32, pages 333–377, 1987.

Cheng and Irani, 1991. Jie Cheng and Keki B. Irani. Ordering Problem Subgoals. In *Proceedings of the National Conference on Artificial Intelligence*, pages 931–936. Morgan Kaufmann, 1991.

Cook, 1990. Diane J. Cook. Analogical Planning. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 22–27. Morgan Kaufmann, 1990.

Currie and Tate, 1991. Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52, pages 49–86, 1991.

Day, 1992. David S. Day. Acquiring Search Heuristics Automatically for Constraint-based Planning and Scheduling. In James A. Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 45–51. Morgan Kaufmann, 1992.

Dejong and Mooney, 1986. Gerald Dejong and Raymond Mooney. Explanation-Based Learning: An Alternate View. *Machine Learning*, 1, pages 145–176, 1986.

Drummond and Currie, 1991. Mark Drummond and Ken Currie. Goal Ordering in Partially Ordered Plans. In *Proceedings of the National Conference on Artificial Intelligence*, pages 960–965. Morgan Kaufmann, 1991.

Ellman, 1989. Thomas Ellman. Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Surveys*, 21, pages 163–221, 1989.

Fikes and Nilsson, 1971. Richard E. Fikes and Nils Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 5, pages 189–208, 1971.

Fikes *et al.*, 1972. Richard E. Fikes, P. E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3, pages 251–288, 1972.

Greiner, 1989. Russell Greiner. Towards A Formal Analysis of EBL. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 450–453, Cornell University, NY, June 1989.

Gries, 1985. David Gries. *The Science of Programming*. The MIT Press, Cambridge, MA, 1985.

Hammond and Converse, 1991. Kristian J. Hammond and Timothy M. Converse. Stabilizing environments to facilitate planning and activity: An engineering argument. In *Proceedings of the National Conference on Artificial Intelligence*, pages 787–793. Morgan Kaufmann, 1991.

Hammond, 1986. Kristian J. Hammond. CHEF: A Model of Case-Based Planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 261–271. Morgan Kaufmann, 1986.

Hammond, 1989. Kristian J. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, Inc., New York, 1989.

Hammond, 1990. Kristian J. Hammond. Explaining and Repairing Plans That Fail. *Artificial Intelligence*, 45, pages 173–228, 1990.

Hanks and Weld, 1992. Steve Hanks and Daniel S. Weld. Systematic Adaptation for Case-Based Planning. In James A. Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 96–105. Morgan Kaufmann, 1992.

Horowitz and Sahni, 1990. Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, New York, 1990.

Joslin and Roach, 1990. David Joslin and John Roach. A Theoretical Analysis of Conjunctive-Goal problems. *Artificial Intelligence*, 41, pages 97–106, 1990.

Kambhampati and Chen, 1993. Subbarao Kambhampati and Jengchin Chen. Relative Utility of EBG based Plan Reuse in Partial Ordering vs. Total Ordering Planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 514–519. Morgan Kaufmann, 1993.

Kambhampati and Hendler, 1992. Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55, pages 193–258, 1992.

Kambhampati and Kedar, 1991. Subbarao Kambhampati and Smadar Kedar. Explanation-Based Generalization of Partially Ordered Plans. In *Proceedings of the National Conference on Artificial Intelligence*, pages 679–685. Morgan Kaufmann, 1991.

Kambhampati, 1990. Subbarao Kambhampati. Mapping and Retrieval During Plan Reuse: A Validation Structure Based Approach. In *Proceedings of the National Conference on Artificial Intelligence*, pages 170–175. Morgan Kaufmann, 1990.

Kambhampati, 1992. Subbarao Kambhampati. Characterizing Multi-Contributor Causal Structures for Planning. In James Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 116–125. Morgan Kaufmann, 1992.

Kambhampati, 1993. Subbarao Kambhampati. On the Utility of Systematicity: Unde ing Tradeoffs between Redundancy and Commitment in Partial-order Planning. I ceedings of the Thirteenth International Joint Conference on Artificial Intelligence, pages 1380–1385. Morgan Kaufmann, 1993.

Keller, 1987. R. M. Keller. Defining operationality for explanation-based learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 482–487. Morgan Kaufmann, 1987.

Knoblock *et al.*, 1991a. Craig A. Knoblock, Steven Minton, and Oren Etzioni. Integrating Abstraction and Explanation-Based Learning in PRODIGY. In *Proceedings of the National Conference on Artificial Intelligence*, pages 541–546. Morgan Kaufmann, 1991.

Knoblock *et al.*, 1991b. Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing Abstraction Hierarchies for Planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 692–697. Morgan Kaufmann, 1991.

Knoblock, 1991a. Craig A. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 923–928. Morgan Kaufmann, 1991.

Knoblock, 1991b. Craig A. Knoblock. Search Reduction in Hierarchical Problem Solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 686–691. Morgan Kaufmann, 1991.

Kolodner *et al.*, 1985. Janet L. Kolodner, Robert L. Simpson, and Katia Sycara-Cyranski. A process model of cased-based reasoning in problem solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 284–290. Morgan Kaufmann, 1985.

Kolodner, 1983. Janet L. Kolodner. Recontructive Memory: A Computer Model. *Cognitive Science*, 7, pages 281–328, 1983.

Korf, 1983. Richard E. Korf. Operator Decomposability: A New Type of Problem Structure. In *Proceedings of the National Conference on Artificial Intelligence*, pages 206–209. Morgan Kaufmann, 1983.

Korf, 1987. Richard E. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33, pages 65–85, 1987.

Koton, 1988. Phyllis A. Koton. *Using Experience in Learning and Problem Solving*. PhD thesis, Massachusetts Institute of Technology, 1988.

Leake, 1991. David B. Leake. An Indexing Vocabulary for Case-Based Explanation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 10–16. Morgan Kaufmann, 1991.

Markovitch and Scott, 1989. Shaul Markovitch and Paul D. Scott. Information Filters and their Implementation in the SYLLOG System. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 404–407, Cornell University, NY, June 1989.

McAllester and Rosenblitt, 1991. David McAllester and David Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 634–639. Morgan Kaufmann, 1991.

McCartney, 1992. Robert McCartney. Case-based planning meets the frame problem (case-based planning from the classical perspective). In James A. Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 172–178. Morgan Kaufmann, 1992.

Minton, 1985. Steven Minton. Selectively Generalizing Plans for Problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 596–602. Morgan Kaufmann, 1985.

Minton, 1990. Steven Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. *Artificial Intelligence*, 42, pages 363–391, 1990.

Newell and Simon, 1963. Alan Newell and Herbert A. Simon. GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. McGraw-Hill, New York, 1963.

Norvig, 1992. Peter Norvig. *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, San Mateo CA, 1992.

Pollack, 1992. Martha E. Pollack. The uses of plans. *Artificial Intelligence*, 57, pages 43–68, 1992.

Redmond, 1990. Michael Redmond. Distributed Cases for Case-Based Reasoning; Facilitating Use of Multiple Cases. In *Proceedings of the National Conference on Artificial Intelligence*, pages 304–309. Morgan Kaufmann, 1990.

Rich and Knight, 1991. Elaine Rich and Kevin Knight. *Artificial Intelligence*. mh, New York, second edition, 1991.

Rissland *et al.*, 1993. Edwina L. Rissland, David B. Skalak, and M. Timur Friedman. Case Retrieval through Multiple Indexing and Heuristic Search. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 902–908. Morgan Kaufmann, 1993.

Rosenbloom *et al.*, 1991. Paul S. Rosenbloom, John E. Laird, Allen Newell, and Robert McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47, pages 289–325, 1991.

Sacerdoti, 1975. Earl D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5, pages 115–135, 1975.

Schank, 1977. R. C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, 1977.

Simmons, 1992. Reid G. Simmons. The roles of associational and causal reasoning in problem solving. *Artificial Intelligence*, 53, pages 159–207, 1992.

Simoudis and Miller, 1990. Evangelos Simoudis and James Miller. Validated Retrieval in Case-Based Reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 310–315. Morgan Kaufmann, 1990.

Tate, 1977. Austin Tate. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 888–893. Morgan Kaufmann, 1977.

Thagard *et al.*, 1990. Paul Thagard, Keith J. Holyoak, Greg Nelson, and David Gochfeld. Analog Retrieval by Constraint Satisfaction. *Artificial Intelligence*, 46, pages 259–310, 1990.

Veloso *et al.*, 1990. Manuala M. Veloso, M. Alicia Perez, and Jaime G. Carbonell. Nonlinear Planning with Parallel Resource Allocation. In *DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 207–212. Morgan Kaufmann, 1990.

Veloso, 1992. Manuala M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie-Mellon University, 1992.

Wilkins, 1988. David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

Woods, 1991. Steven G. Woods. An Implementation and Evaluation of a Hierarchical Nonlinear Planner. Master's thesis, University of Waterloo, 1991.

Yamamura and Kobayashi, 1991. Masayuki Yamamura and Sigenobu Kobayashi. An Augmented EBL and its Applications to the Utility Problem. In *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence*, pages 623–629. Morgan Kaufmann, 1991.

Yang and Tenenberg, 1990. Qiang Yang and Josh Tenenberg. ABTWEAK: Abstracting a Nonlinear, Least Commitment Planner. In *Proceedings of the National Conference on Artificial Intelligence*, pages 204–209. Morgan Kaufmann, 1990.

Vita

Captain Douglas E. Dyer was born on 24 September, 1959 in Ann Arbor, Michigan. He graduated from Blacksburg High School in Blacksburg, Virginia in 1977 and attended the Virginia Polytechnic Institute and State University where he received a Bachelor of Science Degree in Chemical Engineering and Biochemistry in 1984. Upon graduation, he was accepted into the Air Force Undergraduate Engineering Conversion Program, received his commission from Officer Training School, and went to Louisiana Tech University to receive his Bachelor of Science Degree in Electrical Engineering. His first assignment was to Rome Laboratory at Griffiss AFB, NY, where he served as Computer Engineer in artificial intelligence research. In 1990, he was awarded the Masters of Science Degree in Computer Science from the State University of New York Institute of Technology at Utica, and in 1991 he entered the in-residence Ph.D. program at the Air Force Institute of Technology (AFIT). His doctoral research focuses on improving efficiency of automatic planning systems through effective reuse of plans.

Permanent address: 4373 Ridgepath Dr.
Dayton, OH 45424

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	*Doctoral Dissertation*

4. TITLE AND SUBTITLE
Searching For Plans Using a Hierarchy of Learned Macros and Selective Reuse

5. FUNDING NUMBERS

6. AUTHOR(S)
Douglas E. Dyer, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/DS/ENG/94J-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
RL/C3C (Northrup Fowler III)
525 Brooks Road
Griffiss AFB, NY 13441-5700

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research presents a new approach to improving the performance of a macro planner: selective reuse. In macro planning, reuse can result in *poorer* performance than when planning with only primitive operators, a phenomenon that has been called the *utility* problem. The utility problem arises because the benefits of reuse are outweighed by the cost of retrieving a macro to reuse and the cost of searching through the larger search space caused by considering reuse candidates. Selective reuse contains the expansion of the search space by limiting the number of reuse candidates considered and limits the search required by considering only those reuse candidates that entail no additional work. Previously, performance improvement in a macro planner has been possible only by selective learning. Unlike selective learning, selective reuse never overlooks a learning opportunity that might have value in future problem solving. This research developed a polynomial-order retrieval method which reduces the cost of retrieving a reuse candidate likely to save search. A macro planner (HINGE) was implemented to explore selective reuse. To improve the probability of beneficial reuse, HINGE searches in a space of plans using a hierarchically-structured search method that provides multiple opportunities for reuse.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Planning, Learning, Problem-Solving, Indexed-Retrieval, Utility Problem	115
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL