# Automatically Generating Abstractions for Problem Solving

**Craig Alan Knoblock**

May 1991

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at Carnegie Mellon University.



The author was supported by an Air Force Laboratory Graduate Fellowship through the Human Resources Laboratory at Brooks AFB. This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U. S. Government.

# Abstract

A major source of inefficiency in automated problem solvers is their inability to decompose problems and work on the more difficult parts first. This issue can be addressed by employing a hierarchy of abstract problem spaces to focus the search. Instead of solving a problem in the original problem space, a problem is first solved in an abstract space, and the abstract solution is then refined at successive levels in the hierarchy. While this use of abstraction can significantly reduce search, it is often difficult to find good abstractions, and the abstractions must be manually engineered by the designer of a problem domain.

This thesis presents a completely automated approach to generating abstractions for problem solving. The abstractions are generated using a tractable, domain-independent algorithm whose only inputs are the definition of a problem space and the problem to be solved and whose output is an abstraction hierarchy that is tailored to the particular problem. The algorithm generates abstraction hierarchies that satisfy the "ordered monotonicity" property, which guarantees that the structure of an abstract solution is not changed in the process of refining it. An abstraction hierarchy with this property allows a problem to be decomposed such that the solution in an abstract space can be held invariant while the remaining parts of a problem are solved.

The algorithm for generating abstractions is implemented in a system called ALPINE, which generates abstractions for a hierarchical version of the PRODIGY problem solver. The thesis formally defines this hierarchical problem solving method, shows that under certain assumptions this method can reduce the size of a search space from exponential to linear in the solution size, and describes the implementation of this method in PRODIGY. The abstractions generated by ALPINE are tested in multiple domains on large problem sets and are shown to produce shorter solutions with significantly less search than problem solving without using abstraction. ALPINE's automatically generated abstractions produce an 85% reduction in search time on the hardest problem sets in two different test domains, including the time to generate the abstractions.

# Acknowledgements

There are a many people that contributed in one way or another to this thesis. I owe a large thanks to my advisor, Jaime Carbonell, for his years of guidance and encouragement. He provided me with a wealth of advice on this research and he painstakingly read and commented on the entire document. He was always a good source of ideas and insights, and he gave me the freedom and encouragement to develop and pursue my own.

The other members of my thesis committee, Tom Mitchell, Paul Rosenbloom, and Herb Simon, each left their mark on this research. Both Tom Mitchell and Herb Simon met with me on a regular basis, listened to my ideas, and provided many useful ones of their own. Paul Rosenbloom improved this thesis considerably through his detailed and insightful comments and suggestions. I am thankful that I had the opportunity to get to know each of them and they took the time to get to know me.

I am also grateful to all the members of the PRODIGY project, who helped provide a stimulating environment in which to do research. Steve Minton was a tremendous source of advice and encouragement. In my early days as a graduate student, Steve and I had endless discussions about the design and implementation of the PRODIGY system, and in the process I learned a lot from him. I consider myself very fortunate that I had his thesis work on which to build. Oren Etzioni was a great influence on my life in graduate school. We spent literally thousands of hours in the same tiny little office, during which time we had many exciting discussions. Oren was always probing the boundaries of my research, and forcing me to focus on the interesting issues. I also enjoyed working with Manuela Veloso, Yolanda Gil, Dan Kuokka, Robert Joseph, and Alicia Perez, all of whom gave me useful feedback on my research over the years. Dan Kahn, Michael Miller, and Ellen Riloff all did an outstanding job supporting and improving the PRODIGY system. A special thanks also goes to Claire Bono, Oren Etzioni, Yolanda Gil, Steve Minton, and Manuela Veloso, who each took the time to read the thesis and give me an abundance of helpful comments.

Outside of CMU, there are a number of people that contributed to the work in this thesis. I had a number of great discussions with Josh Tenenberg on topics related to abstractions. Through these discussions, Josh saved me a great deal of

time and effort by pointing out some of the problems that were likely to arise. He also introduced me to Qiang Yang, and the three of us endlessly debated the properties of abstraction hierarchies, which ultimately lead to a paper that formally defined the various properties. In the process of writing the paper, I learned a great deal from both of them. There are also a number of other people in the research community with whom I spent many enjoyable hours discussing research: James Altucher, John Anderson, Steve Chien, Jens Christensen, Nick Flann, Fausto Giunchiglia, Armand Prieditis, David Ruby, Devika Subramanian, Amy Unruh, Toby Walsh, Dan Weld, to name a few.

And then there are the friends that I haven't already mentioned that made my time at CMU a truly enjoyable and memorable one: Robert, Edie, Ruth, Jay, Chris, Richard, Robert, Geli, Bernd, Amy, Wenling, Guy, Puneet, Dave, Raul, Carmen, Jeff, Mark, Brad, and many others.

My family, Peter, Joy, Jan, Karen, Ellen, Marion, Todd, Kathy, Gary, and Zelda all provided much love and support throughout my life in graduate school. Finally, there is Claire, who lived through this thesis as much, if not more, than I did. She was always there to listen to my ramblings and to help me forget them.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Given a complex problem, automated problem solvers usually forge ahead, blindly addressing central and peripheral issues alike, without any understanding of which parts of a problem are more difficult and should therefore be solved first. This can result in a significant amount of wasted effort since a problem solver will spend time solving the details only to have to discard the solutions in the process of solving the more difficult aspects. Even for simple tasks, like building a stack of blocks or finding a path for a robot through a configuration of rooms, brute-force search can be ineffective since the search spaces can be quite large. As problem solvers are applied to increasingly complex problems, the ability to decompose a problem and solve the more difficult parts first becomes even more critical.

An effective approach to building more intelligent problem solvers is to use abstraction in order to help focus the search. Abstraction has been used successfully to reduce search in a number of problem solvers including GPS [Newell and Simon, 1972], ABSTRIPS [Sacerdoti, 1974], NOAH [Sacerdoti, 1977], NONLIN [Tate, 1976] MOLGEN [Stefik, 1981] and SIPE [Wilkins, 1984]. These systems use abstraction to focus attention on the difficult parts of the problem, leaving the details or less critical parts of a problem to be filled in later. This is usually done by first solving a problem in an abstract space and then using the abstract solution to guide the problem solving of the original more complex problem.

While abstraction has been widely used in problem solving, the problem of finding good abstractions has not been carefully studied, and has not been automated. In most problem solvers that use abstraction, the designer of a problem space must manually engineer the appropriate abstractions. This process is largely a black art since it is not even well-understood what makes a good abstraction. In addition, most existing hierarchical problem solvers employ a single, fixed abstraction hierarchy for all problems in a given domain, but in many cases the best abstraction for a problem is specific to the particular problem at hand. Automatically constructing

abstractions for problem solving frees the designer of a problem space from concerns about efficiency and it makes it practical to construct abstractions that are tailored to individual problems or classes of problems.

This dissertation develops a theory of what makes a good abstraction for problem solving, presents an approach to generating abstractions automatically using the theory, and investigates the use of these abstractions for problem solving. To demonstrate these ideas, the thesis describes the design, implementation, and evaluation of an abstraction learner and hierarchical problem solver. The implemented system produces abstractions in a variety of problem spaces and the experiments show that these abstractions provide significant reductions in search over problem solving without the use of abstraction.

## 1.1 Problem Solving

Problem solving involves finding a sequence of actions (operators) that solve some problem. A problem is defined in terms of an initial state and a set of goal conditions. The legal operators are defined in terms of preconditions and effects, where the preconditions must be satisfied before the action can be applied, and the effects describe the changes to the state in which the action is applied. A solution to a problem consists of a sequence of operators that transform the given initial state into some final state that satisfies the goals. The terms "problem solving" and "planning" have both been used to describe this process, although problem solving is sometimes considered to subsume planning. This thesis uses the terms interchangeably.

The problem-solving framework is sufficiently general to represent tasks in a large variety of domains, ranging from simple block stacking to more complex process planning and scheduling tasks. While problem solvers differ in the languages they use to express actions, and thus in the types of problems they can represent, they all share the problem of how to control search. Even the simplest problems can be hard to solve due to the large search spaces.

This thesis builds on the PRODIGY problem solver [Minton *et al.*, 1989a, Minton *et al.*, 1989b, Carbonell *et al.*, 1991]. PRODIGY is a means-ends analysis problem solver that was designed as a testbed for learning in the context of problem solving. The idea is to start with a simple and elegant problem solver that has a sufficiently expressive language to represent interesting problems. The problem solver is given a specification of the problem space and is expected to become proficient in the given space by forming its own control knowledge through both analysis of the problem space and problem-solving experience.

In addition to the basic problem solver, PRODIGY consists of a number of learning modules, including modules for explanation-based learning [Minton, 1988a], static

learning [Etzioni, 1990], learning by analogy [Veloso and Carbonell, 1990], learning by experimentation [Carbonell and Gil, 1990], graphical knowledge acquisition [Joseph, 1989], and abstraction generation. While this thesis describes only the problem solving and abstraction generation components of PRODIGY, it does provide comparisons with the explanation-based and static learning modules. Since the same basic problem solver serves as the underlying performance engine for the various learning modules, it facilitates both the comparison and integration of various approaches to reducing search in problem solving and enables the evaluation of the abstraction learning component on problem spaces that were previously developed in PRODIGY. As such, PRODIGY provides an ideal testbed for the work in this thesis.

## 1.2 Hierarchical Problem Solving

Hierarchical problem solvers employ one or more abstractions of a problem space to reduce the search in problem solving. Instead of attempting to solve problems in the original problem space by plowing through the morass of details associated with a problem, a hierarchical problem solver first solves a problem in a simpler abstract space where the problem solver can focus on the "real" problem and ignore the details.

There are several ways to form an abstraction of a problem space. The approach used in this thesis is to remove properties (literals) from the problem space. This has the effect of forming a *reduced model* of the original space in which a single abstract state corresponds to one or more states in the original problem space. In this thesis, the language of a reduced model is a subset of the language of the original problem space. An alternative approach to constructing abstraction spaces is to form a *relaxed model* by weakening the applicability conditions of the operators in a problem space. This was the approach taken in ABSTRIPS [Sacerdoti, 1974], where the preconditions of the operators were assigned criticality values and all preconditions with criticality values below a certain threshold were ignored.

A hierarchical problem solver is given a problem to be solved, a ground-level problem space, and one or more abstractions of that problem space. As illustrated in Figure 1.1, the abstractions are arranged in a hierarchy, where a problem is first mapped into the most abstract space in the hierarchy, solved in that space, and then the abstract solution is refined through successively more detailed spaces until the original problem is solved. An abstract solution is refined at each level by inserting operators to achieve the conditions that were ignored at the more abstract spaces.

An example problem domain in which hierarchical problem solving can significantly reduce search is a process planning and scheduling domain. The problem is to make a set of parts, which requires determining the particular machining operations needed to construct the parts and scheduling the operations on the available ma-

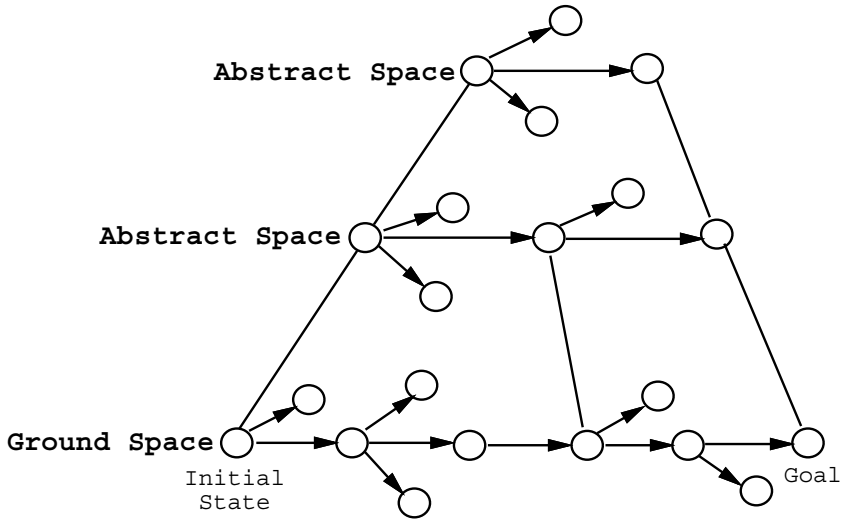


Figure 1.1: Hierarchical Problem Solving

chines. A natural abstraction of this problem, and one that is used in practice, is to separate the selection and ordering of the operations from the actual scheduling of the operations on the machines. This abstraction eliminates wasted effort by detecting interactions caused by the ordering of the operators before the scheduling has even been considered.

Hierarchical problem solving can reduce the size of the search space from exponential to linear in the size of the solution under certain assumptions. For single-level problem solving the size of the search space is exponential in the solution length. Hierarchical problem solving reduces this complexity by taking a large complex problem and decomposing it into a number of smaller subproblems. This thesis formally defines hierarchical problem solving, provides a theoretical analysis of the search reduction and identifies the conditions under which the technique can produce an exponential reduction in the size of the search space.

The hierarchical problem-solving method described in the thesis is implemented as an extension to the PRODIGY problem solver. The system was extended by adding a module to perform the hierarchical control, while employing the basic PRODIGY system to solve the subproblems that arise at each abstraction level. This approach maintains both the problem space and control languages of PRODIGY, with the added functionality of hierarchical problem solving.

## 1.3 Generating Abstraction Hierarchies

While it has been shown that using abstractions for hierarchical problem solving can reduce search, a question that previously has not been addressed is how to find an effective set of abstractions for use in problem solving. In most of the existing hierarchical problem solvers, the abstractions are constructed by the designer of the problem space. While this is possible in some cases, it is often difficult to find good abstractions and impractical to tailor them to individual problems.

A good abstraction is one that separates out those parts of the problem that can be solved first and then held invariant while other parts of a problem are solved. To capture this idea, the thesis identifies two properties of an abstraction hierarchy – the *monotonicity* and *ordered monotonicity* properties.

**Monotonicity Property:** the existence of a ground-level solution implies the existence of an abstract-level solution that can be refined into a ground solution while leaving the literals *established* in the abstract plan unchanged.

The monotonicity property holds for all abstraction hierarchies and guarantees that if a solution exists it can be found without modifying the abstract plans. This property is useful because it provides a criterion for backtracking that preserves completeness. Whenever a problem solver would undo a literal established in an abstract plan while refining the plan, the system can backtrack instead. Yang and Tenenberg [1990] developed a complete nonlinear, least-commitment problem solver using the monotonicity property to constrain the search for a refinement of an abstract plan. While the property is useful for constraining the refinement process, it is not restrictive enough to provide a criterion for generating abstractions.

A restriction of the monotonicity property, called the ordered monotonicity property, does provide a useful criterion for generating abstraction spaces. This property is defined as follows:

**Ordered Monotonicity Property:** every refinement of an abstract plan leaves *all* the literals that comprise the abstract space unchanged.

The ordered monotonicity property is more restrictive than the monotonicity because it not only requires that there exists a refinement of an abstract plan that leaves the literals in the abstract plan unchanged, but that every refinement of an abstract plan leaves all the literals in the abstract space unchanged. This property can be used as the basis of an algorithm for constructing hierarchies of abstraction spaces.

This thesis presents a tractable algorithm for automatically generating abstraction hierarchies from only the initial problem space definition and a problem to be solved. Using the definition of a problem space, the algorithm determines the possible

interactions between literals, which define a set of constraints on the final abstraction hierarchy. The algorithm partitions the literals of a problem space into levels such that the literals in one level do not interact with literals in a more abstract level. The resulting abstraction hierarchies are guaranteed to satisfy the ordered monotonicity property.

In the previous work on hierarchical problem solving, the problem solver was provided with a single, fixed abstraction hierarchy. However, what makes a good abstraction for one problem may make a bad abstraction for another. Thus, the algorithm presented in the thesis generates abstraction hierarchies that are tailored to the individual problems. For example, the STRIPS robot planning domain [Fikes and Nilsson, 1971] involves using a robot to move boxes among rooms and opening and closing doors as necessary. For problems that simply involve moving boxes between rooms, doors are a detail that can be ignored since the robot can simply open the doors as needed. However, for problems that require opening or closing a door as a top-level goal, whether a door is open or closed is no longer a detail since it may require planning a path to get to the door.

The algorithm for generating abstractions is implemented in the ALPINE system. Given a problem space and problem, ALPINE generates an abstraction hierarchy for the hierarchical version of PRODIGY. The system has been successfully tested on a number of problem-solving domains including the original STRIPS domain [Fikes and Nilsson, 1971], a more complex robot planning domain [Minton, 1988a], and a machine-shop planning and scheduling domain [Minton, 1988a]. In all these domains, the system efficiently generates problem-specific abstraction hierarchies that provide significant reductions in search.

## 1.4 Closely Related Work

This section briefly describes the most closely related work on both generating and using abstractions for problem solving. Chapter 6 provides a more comprehensive discussion of the work related to this thesis.

The first hierarchical problem solver was implemented as a planning method in GPS [Newell and Simon, 1972]. Given a problem space and an abstraction of that problem space, GPS maps the problem into the abstract space, solves the abstract problem and then uses the solution to guide the problem solving in the original space. The system was tested in the domain of propositional logic, where in the abstract space the differences between the connectives is ignored. GPS provided the first automated use of abstraction for problem solving, but did not automate the construction of the abstractions.

ABSTRIPS [Sacerdoti, 1974] employs a similar problem-solving technique to GPS

and was the first system to demonstrate empirically that abstraction could be used to reduce search in problem solving, In addition, the work on ABSTRIPS was the earliest attempt at automating abstraction formation. To form an abstraction hierarchy the system must be given an initial partial order of the problem space predicates. ABSTRIPS then forms the abstraction hierarchy by placing the static conditions (those conditions that cannot be changed by any operator) in the most abstract level and placing the preconditions that cannot be achieved by a "short plan" in the next level. The remaining levels come from the user-defined partial order. As described in Chapter 5, ALPINE completely automates the formation of the abstraction hierarchies in the ABSTRIPS's domain and produces abstractions that are considerably more effective at reducing search than the ones generated by ABSTRIPS.

Since these early efforts, there have been a number of systems that use abstractions for problem solving. These systems include NOAH [Sacerdoti, 1977], MOLGEN [Stefik, 1981], NONLIN [Tate, 1976], and SIPE [Wilkins, 1984]. However, all of these systems must be provided with abstractions that are hand-crafted for the individual domains.

More recently, Unruh and Rosenbloom [1989] developed a weak method in SOAR [Laird *et al.*, 1987] that dynamically forms abstractions for look-ahead search by ignoring unmatched preconditions. The choices made in the look-ahead search are stored by SOAR's chunking mechanism and the chunks are then used to guide the search in the original space. The system decides which conditions to ignore based on which conditions hold during problem solving. Since ALPINE constructs abstractions by analyzing the problem space and problem, while SOAR forms the abstractions based on which conditions did or did not hold in solving a particular problem, the abstractions produced by ALPINE are more likely to ignore the appropriate conditions for a given problems. On the other hand, the more stringent requirements on the abstractions formed by ALPINE prevent it from finding abstractions in problem spaces in which SOAR can produce abstractions (e.g., the eight puzzle).

Christensen [1990] also built a hierarchical planner, called PABLO, that also forms its own abstraction hierarchies. PABLO partially evaluates the operators before problem solving to determine the number of steps required to achieve a given goal. The system then uses this information to refine a plan by always focusing on the part of the problem that requires the greatest number of steps. This approach is a generalization of the ABSTRIPS approach, where ABSTRIPS forms the abstractions based on whether a short plan exists, and PABLO forms the abstractions based on the length of the plan. A difficulty with the approach implemented in PABLO is that it may be expensive to partially evaluate the operators in more complex problem spaces. In contrast, ALPINE constructs abstractions based on the interactions between literals, which can be determined without requiring any partial evaluation.

Korf [1987] developed an alternative method for using abstractions for problem solving. Instead of dropping conditions to form an abstract problem space, an abstract

space is constructed by replacing the original set of operators by a set of macro operators. This differs from hierarchical problem solving because once the problem is solved in the macro space the problem is completely solved. Korf shows that this approach can reduce the average search time from $O(n)$ to $O(\log n)$, but the disadvantage of this approach is that it may be difficult or impossible to define a set of macros that adequately cover the given problem space.

## 1.5 Contributions

The primary contributions of the thesis are the discovery and formalization of the properties that can be used to produce effective abstraction hierarchies, the approach to generating abstractions automatically, the definition and analysis of hierarchical problem solving, and the implementation and empirical demonstration of both the automatic abstraction generation and hierarchical problem solving. This section describes each of these contributions.

First, the thesis identifies the *monotonicity* property and a refinement of this property, called the *ordered monotonicity* property, which capture the critical features of an abstraction that determine its utility in problem solving. The monotonicity property is based on the idea that the basic structure of an abstract plan should not be changed in the process of refining the plan. This property provides a criterion for pruning the search for a refinement of an abstract plan since a problem solver only needs to consider refinements that do not violate the abstract plan structure. The ordered monotonicity property, in addition to pruning the search space, also provides an effective criterion for generating useful abstraction hierarchies. This property requires that the literals in an abstraction hierarchy are ordered such that achieving literals at one level will not interact with literals at a more abstract level.

Second, the thesis provides a completely automated approach to generating abstraction hierarchies based on the ordered monotonicity property. The algorithm described in the thesis is given a problem space and problem as input and, by analyzing the potential interactions, it finds a set of constraints on the possible abstractions hierarchies that are sufficient to guarantee the ordered monotonicity property. Because the best abstraction hierarchy varies from problem to problem, the algorithm generates abstraction hierarchies that are tailored to the individual problems. The resulting abstraction hierarchies define a set of abstract problem spaces that are each a reduced model of the original problem space. These abstract spaces can then be used for hierarchical problem solving, as well as learning control knowledge.

Third, the thesis presents a precise definition and analysis of hierarchical problem solving. Previous work on hierarchical problem solving provided only vague descriptions of the problem-solving method with little or no analysis of the potential search

reduction. The hierarchical problem-solving method described in this thesis uses the solutions at each abstract level to divide up a problem into a number of simpler subproblems. The thesis analyzes the potential search reduction of this method, shows that this problem-solving method can provide an exponential-to-linear reduction in the size of the search space, and identifies the conditions under which such reductions are possible.

Fourth, the thesis provides an implementation and empirical demonstration of both the abstraction learner and hierarchical problem solver. The abstractions are generated by a system called ALPINE and then used in a hierarchical version of the PRODIGY problem solver. The thesis presents results on both generating and using abstractions on large sets of problems in multiple problem spaces that had been previously defined in PRODIGY. The use of abstraction is compared in PRODIGY to single-level problem solving, as well as problem solving with hand-coded control knowledge and control knowledge learned by EBL [Minton, 1988a] and STATIC [Etzioni, 1990]. The results show that the abstractions provide significant reductions in search and improvements in solution quality.

## 1.6 A Reader's Guide to the Thesis

The thesis is divided into seven chapters, which describe problem solving, hierarchical problem solving, automatically generating abstractions, results, related work, and limitations and future work. The chapters of the thesis are organized as follows.

Chapter 2 presents the basic problem-solving model. This chapter defines a problem space and the corresponding problem solving terminology, and then describes the PRODIGY problem solver, which forms the underlying system for the implementation and evaluation of the ideas in the thesis. Chapters 2, 3, and 4 all have the same internal organization. They first present the basic ideas of the chapter, then illustrate these ideas using the Tower of Hanoi domain, and lastly, describe the implementation of the ideas.

Chapter 3 describes how the abstractions are used for hierarchical problem solving. This chapter defines an abstraction space, presents a method for hierarchical problem solving, shows that this method can produce an exponential-to-linear reduction in the size of the search space, and identifies the conditions under which such a reduction is possible. The last section of this chapter describes the implementation of hierarchical problem solving in PRODIGY.

Chapter 4 presents an approach to automatically generating abstraction hierarchies for problem solving. This chapter explores the relationships between a problem space and abstractions of a problem space, defines the monotonicity and ordered monotonicity properties, and then describes an algorithm for generating abstractions

based on the ordered monotonicity property. Lastly this chapter describes an implemented system called ALPINE that produces problem-specific abstraction hierarchies using this algorithm.

Chapter 5 presents the empirical results for both generating and using the abstractions for problem solving. This chapter is divided into four sections. The first section explores the affect of the problem-solving search strategy on the search reduction in hierarchical problem solving. The second section presents empirical results for both generating and using abstractions in ALPINE. The third section compares the use of the abstractions generated by ALPINE to the use of control knowledge generated using explanation-based learning. The last section compares the abstractions generated by ALPINE to those generated by ABSTRIPS.

Chapter 6 compares and contrasts the work in this thesis with other work related to generating and using abstractions for problem solving.

The final chapter, Chapter 7, describes the limitations of this work, presents a number of directions for future work, and attempts to characterize where this thesis leaves off and what remains to be done.

There are four appendices to the thesis, which contain the problem space definitions, example problems, and experimental results for the four different domains used in this thesis. Appendix A presents the Tower of Hanoi, Appendix B presents the extended robot-planning domain, Appendix C presents the machine-shop planning and scheduling domain, and Appendix D presents the original STRIPS domain as it is encoded in PRODIGY.

There are several ways one can approach this thesis besides reading the entire document from cover to cover. For a glimpse at the content of the thesis, read the example sections on the Tower of Hanoi – Sections 2.2, 3.4, and 4.3. These sections illustrate the basic ideas using the Tower of Hanoi puzzle. All the chapters of the thesis are fairly self-contained, and the reader should be able to skip around by reading the definitions at the beginning of the preceding chapters. On a first reading, the reader may want to skim the formal definitions and proofs.

# Chapter 2

# Problem Solving

Problem solving is a process that has been widely studied in AI from the early days of GPS [Newell *et al.*, 1962, Ernst and Newell, 1969] and STRIPS [Fikes and Nilsson, 1971] to more recent planners such as SIPE [Wilkins, 1984], SOAR [Laird *et al.*, 1987] and PRODIGY [Minton *et al.*, 1989b, Minton *et al.*, 1989a, Carbonell *et al.*, 1991]. A problem solver is a given a problem space definition and a problem and is asked to find a solution to the problem. A *problem space* is defined by the legal operators and states. *Operators* are composed of a set of conditions, called *preconditions*, that must be true in order to apply an operator and a set of *effects* that describe the changes to the state that result from applying an operator. *States* are composed of a set of conditions that describe the relevant features of a model of the world. A problem consists of an *initial state*, which describes the initial configuration of the world, and a *goal*, which describes the desired configuration. To solve a problem, a problem solver must find a sequence of operators that transform the initial state into a state that satisfies the goal.

This chapter contains three parts – a formal definition of problem solving, an example problem solving task, and a description of the PRODIGY problem solver. The formal definition of problem solving provides a precise definition of the task and the corresponding terminology, which is in turn used in the definitions in the following chapters. The second part provides an example from the Tower of Hanoi domain. This example is used throughout the thesis to illustrate the basic ideas. The last section describes the PRODIGY problem solver, which provides the foundation for the implementation of the ideas in the thesis.

## 2.1 Definition of Problem Solving

A problem space $\Sigma$ is a triple $(L, S, O)$, where $L$ is a first-order language, $S$ is a set of states, and $O$ is a set of operators.[1] Each state $S_i \in S$ is a finite and consistent set of atomic sentences in $L$. Each operator $\alpha \in O$ is defined by a triple $(P_\alpha, D_\alpha, A_\alpha)$, where $P_\alpha$, the preconditions, are a set of literals (positive or negative atomic sentences) in $L$, and both the deletes $D_\alpha$ and adds $A_\alpha$ are finite sets of atomic sentences in $L$. The combination of the adds and deletes comprise the effects of an operator $E_\alpha$, such that if $p \in A_\alpha$ then $p \in E_\alpha$ and if $p \in D_\alpha$ then $(\neg p) \in E_\alpha$.

A *problem* $\rho$ consists of two components:

- An initial state $S_0 \in S$, where $S_0$ is a description of an initial state of the world.
- A goal state $S_g \in S$, where $S_g$ is a partial description of a desired state.

The *solution* (or *plan*) $\Pi$ to a problem is a sequence of operators that transforms the initial state $S_0$ into some final state $S_n$ that satisfies the goal state $S_g$. A plan is composed of the concatenation of operators or subplans. (The '$\|$' symbol is used to represent the concatenation of operators or sequences of operators.)

Let $\mathcal{A} : O \times S \rightarrow S$ be an *application* procedure that applies an operator to a state to produce a new state by removing the deleted literals, and inserting the added literals. For any state $S_i$ (where '$\setminus$' represents set difference),

$$\mathcal{A}(\alpha, S_i) = (S_i \setminus D_\alpha) \cup A_\alpha.$$

The application procedure can be extended to apply to plans in the obvious way, where each operator applies to each of the resulting states in sequence. Thus, given the initial state $S_0$, a plan $\Pi \equiv \alpha_1 \| \ldots \| \alpha_n$ defines a sequence of states $S_1, \ldots, S_n$, where

$$S_i = \mathcal{A}(\alpha_1 \| \ldots \| \alpha_i, S_0) = \mathcal{A}(\alpha_i, S_{i-1}) \;\; 1 \leq i \leq n$$

A plan $\Pi$ is *correct* whenever the preconditions of each operator are satisfied in the state in which the operator is applied:

$$P_{\alpha_i} \subseteq S_{i-1} \;\; 1 \leq i \leq n$$

$\Pi$ *solves* a problem $\rho = (S_0, S_g)$ whenever $\Pi$ is correct and the goal $S_g$ is satisfied in the final state: $S_g \subseteq \mathcal{A}(\Pi, S_0)$.

[1] The formalization of problem solving presented in this section is loosely based on Lifschitz's formalization of STRIPS [Lifschitz, 1986].

Let $\mathcal{P} : \Sigma \times S \times S \rightarrow \Pi$ be a *problem solving* procedure that is given a problem space $\Sigma$, an initial state $S_0$, and a goal state $S_g$ and produces a plan $\Pi$ with the operators in $\Sigma$ that solves the goal.

$$\Pi = \mathcal{P}(\Sigma, S_0, S_g)$$

The procedure $\mathcal{P}$ is not guaranteed to produce a plan, since there may be goals that are not solvable from a given initial state and goals that are not solvable from any initial state.

There are a variety of approaches to problem solving, which range from simple forward-chaining or backward chaining to more sophisticated goal-directed approaches, such as means-ends analysis and least-commitment planning.

Means-ends analysis, which was developed in GPS [Newell *et al.*, 1962], integrates the forward- and backward-chaining approaches. A means-ends analysis problem solver identifies the differences between the goal and the current state and then selects an operator that reduces these differences. If the selected operator is directly applicable in the current state then it is applied to produce a new state. Otherwise the problem solver attempts to reduce the differences between the current state and the state in which the selected operator can be applied. This process is repeated until the initial state is transformed into a state that satisfies the goal. Other means-ends analysis problem solvers include STRIPS [Fikes and Nilsson, 1971] and PRODIGY [Carbonell *et al.*, 1991].

A least-commitment problem solver, which was first implemented in NOAH [Sacerdoti, 1977], searches through the space of plan refinements, instead of searching the state space, in order to build a partially ordered sequence of operators that solves a given problem. The plan refinement space consists of a set of plan modification operators that construct and refine a partially ordered plan. For example, an *establishment* operator inserts an operator into the partial plan to achieve a goal and a *promotion* operator orders one operator before another in the plan. Chapman [1987] identified a complete set of plan modification operators and implemented them in a planner called TWEAK. Other least-commitment problem solvers include NONLIN [Tate, 1976], MOLGEN [Stefik, 1981], and SIPE [Wilkins, 1984].

## 2.2 Tower of Hanoi Example

This section presents an example of problem solving in the Tower of Hanoi puzzle, which is then used in the following chapters to illustrate the techniques for both hierarchical problem solving and generating abstractions. The puzzle requires moving a pile of various-sized disks from one peg to another with the use of an intermediate peg. The constraints are that only one disk at a time can be moved, a disk can only

be moved if it is the top disk on a pile, and a larger disk can never be placed on a smaller one. Figure 2.1 shows the initial and goal states of a three-disk Tower of Hanoi problem.

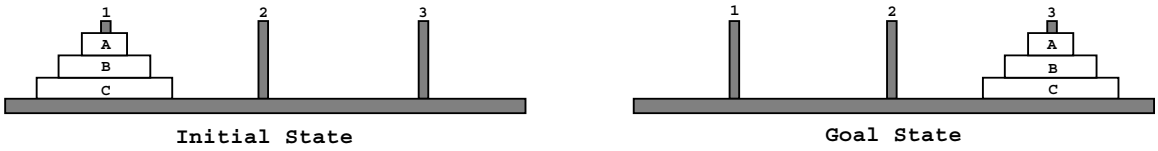


Figure 2.1: Initial and Goal States in the Tower of Hanoi

There are a variety of ways to express the legal operators of the Tower of Hanoi for problem solving. The usual approach is to express the operators as a set of operator schemata, where each schema is parameterized over one or more arguments of the operator. Thus, each operator schema corresponds to one or more fully-instantiated operators. Using operator schemata the three-disk Tower of Hanoi can be axiomatized in three operators, where there is one schema for moving each disk. Table 2.1 shows the schema for moving the largest disk, `diskC` from a source peg to a destination peg. The Tower of Hanoi can also be axiomatized as a single-operator that is parameterized over both the pegs and the disks. The single-operator representation is described in Section 7.2.1.

```
(Move_DiskC
  (preconds (and (on diskC source-peg)
                 (not (equal source-peg dest-peg))
                 (not (on diskB source-peg))
                 (not (on diskA source-peg))
                 (not (on diskB dest-peg))
                 (not (on diskA dest-peg))))
  (effects ((del (on diskC source-peg))
  (add (on diskC dest-peg)))))
```

Table 2.1: Operator Schema in the Tower of Hanoi

To simplify the exposition, a fully-instantiated representation of the Tower of Hanoi example will be used throughout the thesis, where there is an operator for moving each disk between each pair of pegs. For the three-disk problem, this axiomatization requires 18 operators (6 for each disk). Table 2.2 shows the operator for moving disk C, the largest disk, from peg 1 to 3. The preconditions require that disk C is initially on peg 1 and that neither disk A nor B are on peg 1 or 3.

```
(Move_DiskC_From_Peg1_to_Peg3
  (preconds (and (on diskC peg1)
                 (not (on diskB peg1))
                 (not (on diskA peg1))
                 (not (on diskB peg3))
                 (not (on diskA peg3))))
  (effects ((del (on diskC peg1))
            (add (on diskC peg3)))))
```

Table 2.2: Instantiated Operator in the Tower of Hanoi

The difficulty of the Tower of Hanoi puzzle increases with the number of disks in the problem. The number of possible states for a given puzzle with $n$ disks and $p$ pegs is $p^n$ since each disk can be on one of the $p$ pegs. The state space, which is the set of states reachable from the initial state using the given operators, for the three-disk puzzle is shown in Figure 2.2 [Nilsson, 1971]. Each node represents a state and is labeled with a picture of that state, and each arrow represents an operator that can be applied to reach the adjacent state.

A solution to the three-disk problem given above consists of any path through the state space that starts at the initial state and terminates at the goal state. The shortest solution follows the path along the straight line between the initial and goal states. Means-ends analysis can be used to solve the Tower of Hanoi problem as follows. First, the goal is compared to the initial state and one of the differences is selected. There are three possible differences to consider: `(on diskA peg3)`, `(on diskB peg3)`, and `(on diskC peg3)`. Assume the problem solver selects the last one. Next an operator is selected that reduces this difference. There are two possible operators, one for moving `diskC` from peg 1 to 3 and the other for moving the disk from peg 2 to 3. If it selects the former, then it would subgoal on moving the smaller disks so that this operator could be applied. This process continues until the initial state has been transformed into the goal state. If the problem solver reaches a dead end or encounters a cycle, it backtracks to one of the previous choice points in the search.

## 2.3 Problem Solving in PRODIGY

The PRODIGY problem solver [Minton *et al.*, 1989b, Minton *et al.*, 1989a, Carbonell *et al.*, 1991] serves as the foundation for the implementation of the work in this thesis. PRODIGY is a general-purpose, means-ends analysis problem solver coupled with a va-

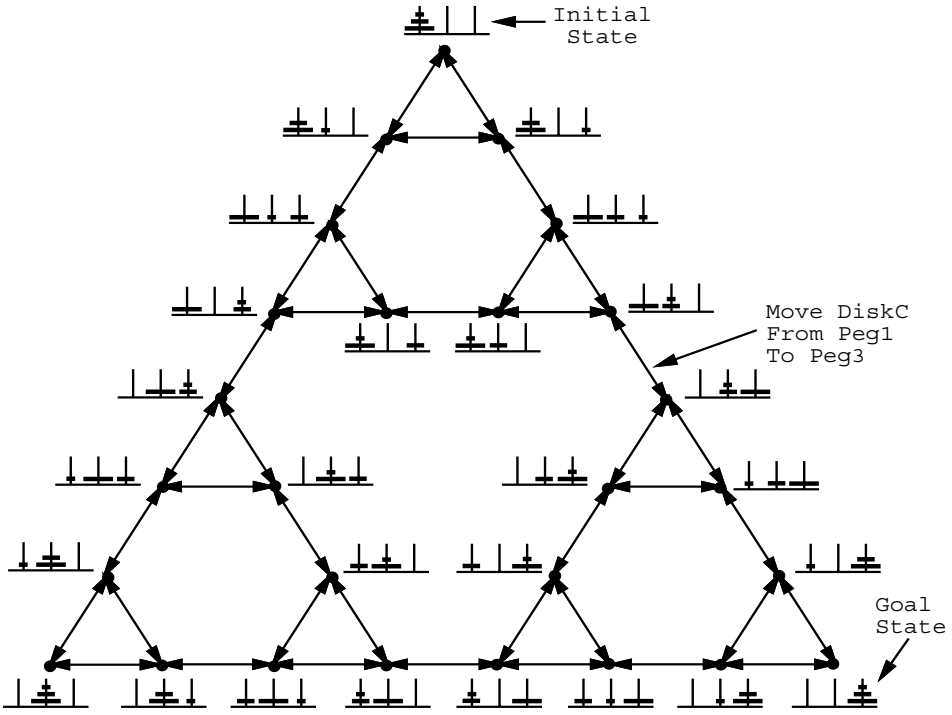


Figure 2.2: State Space of the Three-Disk Tower of Hanoi

riety of learning mechanisms. In addition to the automatic generation of abstractions described in this thesis, PRODIGY includes modules for explanation-based learning [Minton, 1988a], static learning [Etzioni, 1990], learning by analogy [Veloso and Carbonell, 1990], learning by experimentation [Carbonell and Gil, 1990], and graphical knowledge acquisition [Joseph, 1989]. PRODIGY has been applied to a variety of domains including the blocks world [Nilsson, 1980], the STRIPS domain [Fikes and Nilsson, 1971], an augmented version of the STRIPS domain [Minton, 1988a], discrete machine-shop planning and scheduling domain [Minton, 1988a], a brewery scheduling domain [Wilkins, 1989], and a computer configuration domain [McDermott, 1982, Rosenbloom *et al.*, 1985].

The section below presents an overview of the basic PRODIGY problem solver. It describes PRODIGY's problem space and problem definitions, describes how the problem solver searches this space, and explains PRODIGY's use of control rules to guide this search. A complete description of the PRODIGY problem solver is presented in [Minton *et al.*, 1989b], and the extensions to PRODIGY for hierarchical problem solving are described in the next chapter.

The description of PRODIGY that follows draws on an example from a machine-shop planning and scheduling domain. This example was previously described in

[Minton *et al.*, 1989a]. The machine-shop domain contains a variety of machines, such as a lathe, mill, drill, punch, spray painter, etc, which are used to perform various operations to produce the desired parts. Given a set of parts to be drilled, polished, reshaped, etc., and a fixed amount of time, the task is to find a plan to both create and schedule the parts that meets the given requirements. A complete definition of the problem space can be found in Appendix C.

### 2.3.1 Problem Space Definition

A problem space in PRODIGY is defined by a set of operators and inference rules. The operators describe the legal transformations between states and the inference rules describe the properties that can be derived from a state.

A state is represented by a database containing a set of ground atomic formulas. There are two types of relations used in the system – primitive relations and defined relations. Primitive relations are directly observable or "closed-world". This means that the truth value of these relations can be immediately determined in a given state. Primitive relations may be added to or deleted from a state by the operators. In contrast, defined relations are inferred on demand using the inference rules. The purpose of defined relations is to avoid explicitly maintaining information that can be derived from the primitive relations.

An operator is composed of a precondition expression and a list of effects. The precondition expression describes the conditions that must be satisfied before the operator can be applied. The expressions are well-formed formulas in the PRODIGY description language (PDL) [Minton *et al.*, 1989b], a language based on predicate logic that includes negation, conjunction, disjunction, existential quantification, and universal quantification over sets. The list of effects describe how the application of the operator changes the world. The effects are a list of atomic formulas that describe primitive relations to be added or deleted from the current state when the operator is applied.

An example operator schema from the machine-shop problem space is shown in Table 2.3. This operator is used to schedule a turn operation on a part, which makes a part cylindrical by turning it on a lathe. The precondition requires that the lathe is idle and the part has not been scheduled on any machines at the same or later time (parts are scheduled starting at the beginning of the schedule). The effects of this operator are that the part has the desired cylindrical shape, it is scheduled on the machine, and it is now last-scheduled at the given time. There are also a number of side-effects from the turning operation, which include removing any paint and making the surface condition rough.

Inference rules have the same syntax as operators, but are used to infer defined relations. As such, an inference rule can only add defined relations, which are not

```
(TURN (part time)
  (preconditions
    (and
      (is-part part)
      (last-scheduled part prev-time)
      (later time prev-time)
      (idle lathe time)))
  (effects (
      (delete (shape part old-shape))
      (delete (surface-condition part old-condition))
      (delete (painted part old-paint))
      (delete (last-scheduled part prev-time))
      (add (surface-condition part rough))
      (add (shape part cylindrical))
      (add (last-scheduled part time))
      (add (scheduled part lathe time)))))
```

Table 2.3: Operator in the Machine-Shop Domain

found in the effects of an operator. For example, Table 2.4 provides an inference rule from the machine-shop problem space. This rule defines the predicate `idle` by specifying that a machine is idle during a time period if no part is scheduled for that machine during that time period.

```
(IS-IDLE (machine time)
  (preconditions
    (not (exists part (scheduled part machine time))))
  (effects
    ((add (idle machine time)))))
```

Table 2.4: Inference Rule in the Machine-Shop Domain

Because inference rules are encoded and applied in a manner similar to operators, PRODIGY can employ a homogeneous control structure, enabling the search-control rules to guide the application of operators and inference rules alike.

### 2.3.2 Problem Definition

As described earlier, a problem consists of an initial state and goal. An initial state is specified as a conjunction of literals. The initial state for the example, illustrated in Figure 2.3, contains `part-b` and `part-c` already scheduled, and `part-a` which has yet to be scheduled. The schedule consists of 20 time slots, and `part-a` is initially unpolished, oblong-shaped, and cool.

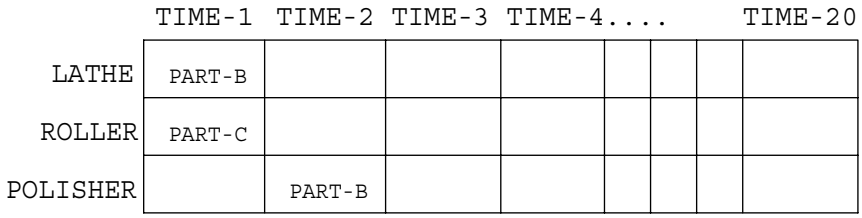


Figure 2.3: Initial State in the Machine-Shop Domain

A goal is any legal PDL expression. An example goal expression for the machine-shop problem space is shown below, where the goal is satisfied if the part named `part-a` is polished and has a cylindrical shape.

```
(and (shape part-a cylindrical)
     (surface-condition part-a polished))
```

### 2.3.3 Searching the Problem Space

PRODIGY begins with a search tree containing a single node representing the initial state and the desired goals. The tree is expanded as follows:

1. **Decision phase:** There are four types of decisions that PRODIGY makes during problem solving. First, it must decide what node in the search tree to expand next, where each node consists of a set of goals and a state describing the world. After selecting a node, PRODIGY chooses a goal, an operator relevant to achieving the goal, and an appropriate set of bindings for the operator. Each choice point in the decision phase can be mediated by a set of control rules, which are described in the next section.

2. **Expansion phase:** If the instantiated operator's preconditions are satisfied, the operator is applied. Otherwise, PRODIGY subgoals on the unmatched preconditions. In either case, a new node is created.

These steps are repeated until PRODIGY generates a node whose state satisfies the top-level goal expression or all possible search paths have been exhausted.

The search tree for the example problem described above is shown in Figure 2.4. The left side of each node shows the goal stack and the pending operators at that point. The right side shows the relevant subset of the state. For example, at Node 3, the current goal is to clamp the part. This is a precondition of the `polish` operator, which is being considered to achieve the goal of being polished. The predicates like `cylindrical` and `hot` in the figure are shorthand for the actual formulas, such as `(shape part-a cylindrical)` and `(temperature part-a hot)`.

In the search tree, the first top-level goal `(shape part-a cylindrical)` is not satisfied in the initial state. To achieve this goal, PRODIGY considers the operators `turn` and `mill`. Both operators have effects that unify with the goal, so either operators can be used to make a part cylindrical, but they have different side-effects. Since there are no control rules to guide this decision, PRODIGY arbitrarily decides to try `mill` first. In order to satisfy the preconditions of `mill`, PRODIGY must infer that `part-a` is available and the milling machine is idle at the desired time. Assume that previously acquired control knowledge indicates a preference for the earliest possible time slot, `time-2`. After milling the part at `time-2`, PRODIGY attempts to polish the part, but the preconditions of `polish` specify that the part must either be rectangular, or clamped to the polisher. Unfortunately, clamping fails, because milling the part has raised its temperature so that it is too hot to clamp without deforming the part or clamp. Since there is no operation to cool the part or make the part rectangular, the attempt to apply `polish` fails at that node.

Backtracking chronologically, PRODIGY then tries milling the part at `time-3`, and then `time-4`, and so on, until the end of the schedule is reached at `time-20`. Each of these attempts fails to produce a solution because the part remains `hot` and is therefore unclampable. (In practice, a part would cool down over time, but this process is not modeled in the axiomatization of the domain.) In any event, the problem solver finally succeeds when it eventually backs up and tries `turning` rather than `milling`.

### 2.3.4 Controlling the Search

As PRODIGY attempts to solve a problem, it must make decisions about the selection of which node to expand, of which goal to work on, of which operator to apply, and which bindings of the operator to use. To make these decisions, PRODIGY uses search control rules, which may be general or problem-space specific, hand-coded or automatically acquired, and may consist of heuristic preferences or definitive selections. In the absence of any search control, PRODIGY defaults to depth-first search with chronological backtracking.

Control rules can be employed to guide the search at the four decisions points described above (nodes, goals, operators, and bindings). Each control rule has an "if"

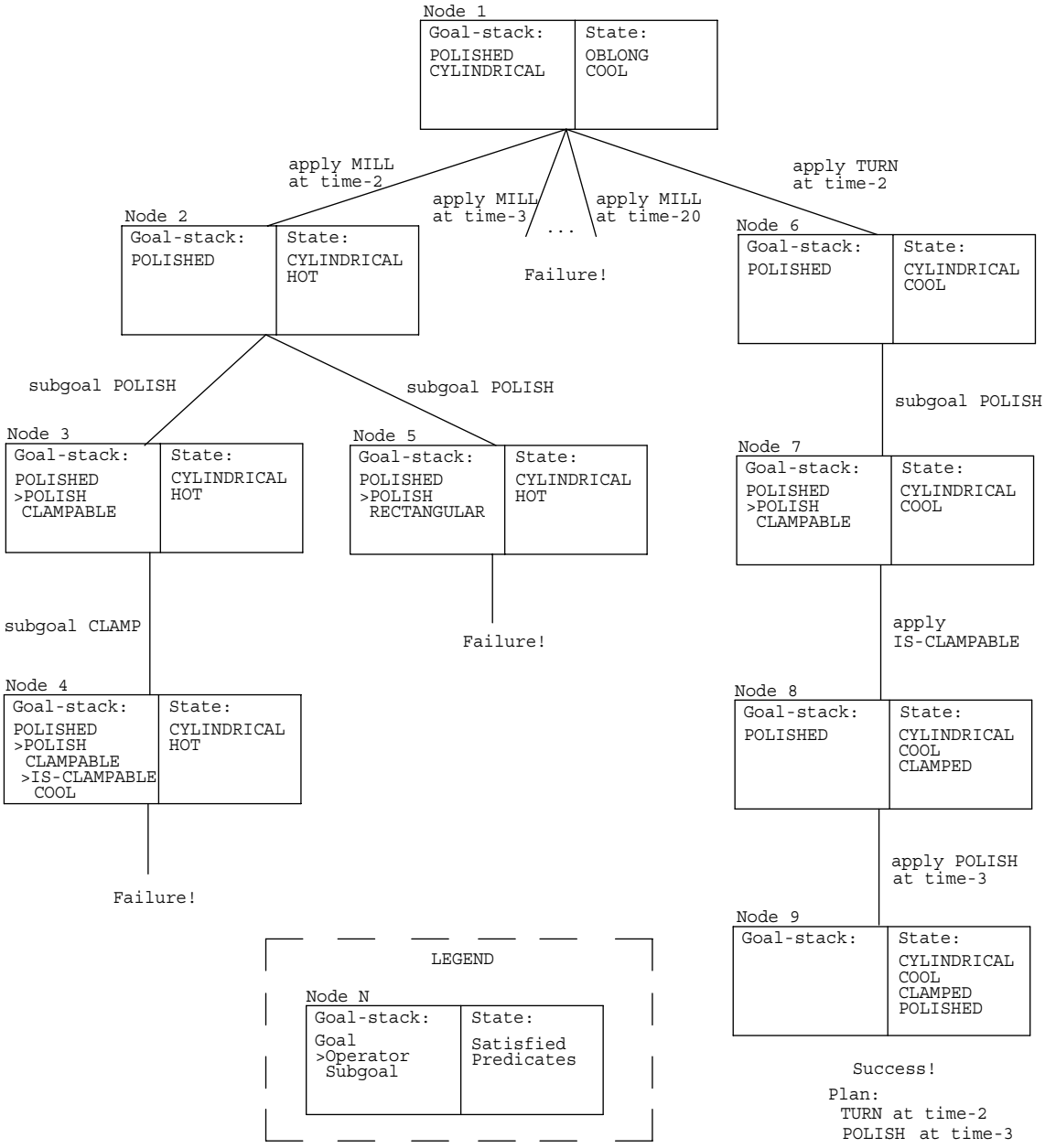


Figure 2.4: Search Tree in the Machine-Shop Domain

condition testing applicability and a "then" condition indicating whether to `select`, `reject`, or `prefer` a particular candidate. Given the alternatives at each decision point, PRODIGY first applies the applicable *selection rules* to select a subset of the alternatives. If no selection rules are applicable, all the alternatives are included. Next *rejection rules* further filter this set by explicitly eliminating some of the alternatives. Last, *preference rules* are used to order the remaining alternatives.

For example, the control rule depicted in Table 2.5 is an operator rejection rule that states that if the current goal at a node is to reshape a part and the part must subsequently be polished, then reject the `mill` operator. The example problem from

```
(DONT-MILL-BEFORE-POLISHING
    (if (and (current-node node)
             (current-goal node (shape part shape))
             (candidate-operator node mill)
             (is-top-level-goal node
                  (surface-condition part polished))))
    (then (reject operator mill)))
```

Table 2.5: Operator Rejection Rule in the Machine-Shop Domain

the previous section illustrates why this rule is appropriate: polishing `part-a` after milling it turned out to be impossible. Had the system previously learned this rule, the problem would have been solved directly, without the costly backtracking at Node 1.

Notice that the "if" condition of the control rule is written in PDL, the same language that is used for the preconditions of operators and inference rules, though different predicates are used. Meta-level predicates such as `current-node` and `candidate-operator` are used in control rules, whereas the predicates used in operators and inference rules are predicates of a problem-space definition, such as `shape` and `idle`. PRODIGY has a set of predefined meta-level predicates.

PRODIGY's reliance on explicit control knowledge distinguishes it from other domain-independent problem solvers. Instead of using a least-commitment search strategy, as in NOAH or SIPE, or a look-ahead search strategy, as in SOAR, PRODIGY expects that important decisions will be guided by the presence of appropriate control knowledge. This control knowledge can take the form of control rules, abstractions, or stored plans, all of which can be used to guide the search. If there is no control knowledge to guide a particular control decision, then PRODIGY makes the control choice arbitrarily. This is referred to as a *casual commitment* strategy. The rationale for this strategy is that for any decision with significant ramifications, control knowledge should be present; if it is not, the problem solver should not attempt to

be clever without knowledge, rather, the cleverness should come about as a result of learning. Thus, the emphasis is on an elegant and simple problem solving architecture that can produce sophisticated behavior by learning control knowledge specific to a problem space. Control knowledge is acquired through experience as in EBL [Minton, 1988a] and derivational analogy [Veloso and Carbonell, 1990] or through problem-space analysis as in STATIC [Etzioni, 1990] and ALPINE (described in the following chapters).

# Chapter 3

# Hierarchical Problem Solving

Abstraction has been used to reduce search in a variety of problem solvers. It reduces search by both focusing the problem solver on the more difficult aspects of a problem first. Most of these problem solvers employ one of three types of abstractions: abstract problem spaces, abstract operators, and macro problem spaces. These three approaches are briefly described below and then compared in more detail in Chapter 6.

The first approach, hierarchical problem solving using abstract problem spaces, employs a hierarchy of abstract problem spaces to first solve a problem in an abstract space and then refine the abstract solution into successively more detailed spaces until it reaches the ground space. This type of hierarchical problem solving is sometimes called length-first hierarchical problem solving since a problem is solved at one level of abstraction before moving to the next level. The technique was first used in GPS [Newell *et al.*, 1962] and ABSTRIPS [Sacerdoti, 1974].

The second approach, hierarchical problem solving using abstract operators, uses a predefined set of abstractions of the operators and expands each operator in the abstract plan to varying levels of detail. Instead of refining the entire plan at one level of detail, the problem solver refines the plan by selectively refining the individual operators in the plan. An operator is refined by replacing an abstract operator with a more detailed operator and achieving the unsatisfied preconditions of the new operator. This approach allows one part of the abstract plan to be expanded while another part is ignored, but eventually the entire plan will be expanded in the ground space. Unlike the length-first model, the abstractions need not be a set of well-defined abstract problem spaces. Instead the problem solver first selects abstract operators that directly achieve the goals and then refines the abstract operators by inserting preconditions of the operators that must hold before operators can be applied in the ground space. This approach, which requires a least-commitment problem solver, was developed in NOAH [Sacerdoti, 1977] and later used in NONLIN [Tate, 1976], MOLGEN [Stefik, 1981], and SIPE [Wilkins, 1984].

The third approach, abstract problem solving using macros, takes a problem and maps it into an abstract space defined by a set of macro operators and then solves the problem in the macro space. Unlike the first two approaches, once a problem is solved in the macro space, the problem is completely solved since the macros are defined by operators in the original problem space. Korf [1987] presented the idea of replacing the original problem space by a macro space. Other people have explored the use of macros in problem solving [Fikes *et al.*, 1972, Minton, 1985, Laird *et al.*, 1986, Shell and Carbonell, 1989], but in most cases the macros are simply added to the original problem space, which may or may not reduce search [Minton, 1985].

The work in this thesis builds on the first approach – problem solving using abstract problem spaces. Before describing this approach to hierarchical problem solving in detail, the first section compares two models of abstraction spaces and describes the one used in this thesis. The second section provides a precise definition of hierarchical problem solving. The third section shows that hierarchical problem solving can provide an exponential reduction in the size of the search space and identifies the assumptions under which this reduction is possible. The fourth section illustrates the use of hierarchical problem solving in the Tower of Hanoi puzzle. The last section describes the implementation of hierarchical problem solving in PRODIGY.

## 3.1 Abstraction Hierarchies

An abstraction hierarchy consists of a hierarchy of abstract problem spaces. This section first defines an abstract problem space and then defines a hierarchy of abstraction spaces.

### 3.1.1 Models of Abstraction Spaces

As described in the previous chapter, a problem space is composed of the legal states and operators. An abstract problem space is formed by simplifying a problem space. One approach is to drop the applicability conditions of the operators to form a *relaxed model*, and another approach is to completely remove certain conditions from the problem space to form a *reduced model*. This section defines relaxed and reduced models and describes the advantages of reduced models over relaxed models.

Given an initial problem-solving domain, a problem space or model of that domain is defined by the states and operators. Figure 3.1 shows a picture of a simple robot-planning domain and the corresponding models of this domain. The initial model is shown at the top of the figure and consists of four states and four different operators. The robot can move back and forth between the two rooms and the door between the rooms can be opened and closed. In the initial model, the robot can only change

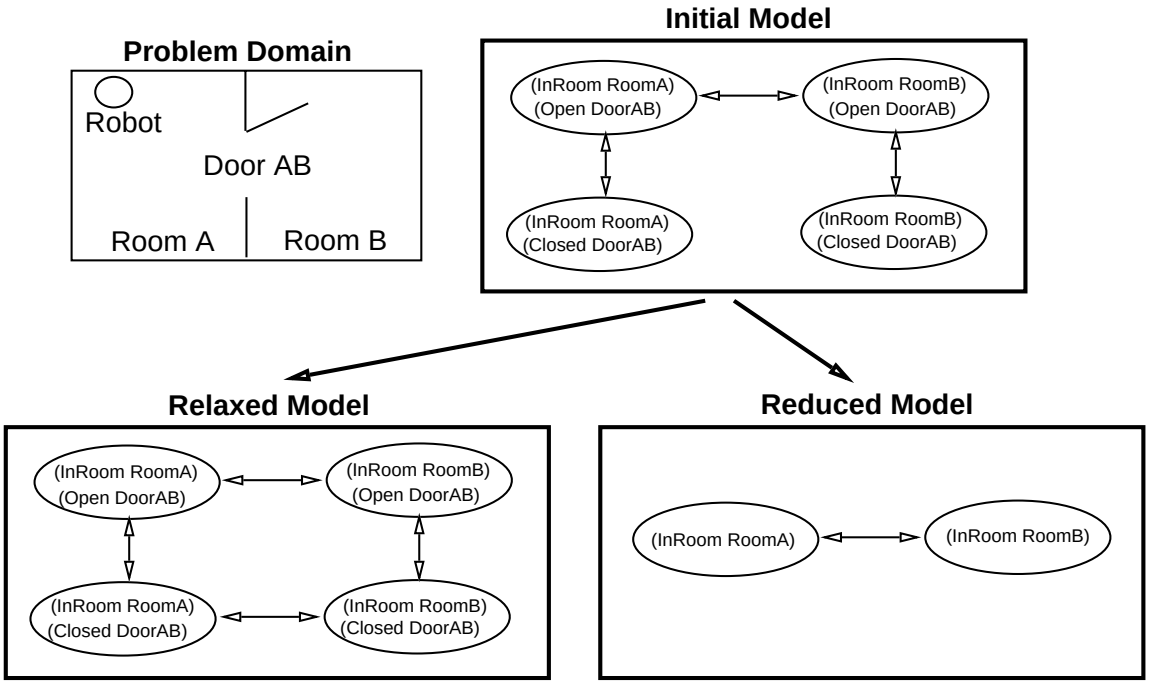


Figure 3.1: Comparison of Relaxed and Reduced Models

rooms if the door between the rooms is open, and the robot can open or close the door from either room.

Relaxed models [Pearl, 1984] are constructed by removing preconditions of operators. This is the approach taken in ABSTRIPS, where the preconditions of the operators are assigned criticality values and all preconditions with criticality values below a certain threshold are ignored. Viewed in terms of a state-space graph, the number of states in a relaxed model is the same as the initial model, but the possible transitions between the states is increased. In the example shown in Figure 3.1, a relaxed model can be constructed by dropping the precondition that the door must be open before the robot can move between rooms. In the resulting model, the operators for moving between rooms are applicable even when the door is closed.

Reduced models are constructed by removing properties (literals) from the original problem space. Thus, an abstract space is formed by dropping every instance of a particular set of literals from both the states and the operators. Also, operators that only achieve literals dropped from the abstract space are removed from the abstract space. In a reduced model of a problem space, a single abstract state corresponds to one or more states in the original problem space. For any pair of states in the original space, if there is an operator that transforms one state into another, there exists an abstract operator that provides a transformation between the corresponding abstract

states. In the case of the reduced model shown in Figure 3.1, ignoring the 'open' and 'closed' conditions involves dropping these conditions from both the operators and states. Thus the 'open' and 'closed' door properties are completely removed from the model, reducing the four original states to the two states shown in the figure and eliminating the operators for opening and closing doors.

Reduced models have a number of advantages over relaxed models. First, in a reduced model conditions are dropped from the states, which can decompose the goal of a problem since different goal conditions may occur at different levels in the hierarchy. Second, since operators are dropped in a reduced model, the branching factor of the abstract search is reduced. Third, a reduced model is a smaller problem space with fewer literals and operators, which can be more concisely represented and reasoned about. This makes it easier to combine the use of abstraction with other types of problem-space learning such as explanation-based learning [Minton, 1988b], macro-operator learning [Korf, 1985b], or learning by analogy [Carbonell, 1986]. Fourth, creating a reduced model allows operators and objects that are indistinguishable at an abstract level to be combined into abstract operators or objects. For example, if there are two operators for moving an object between rooms, where one operator involves carrying the object and the other involves pushing the object, and the distinctions between these operators are ignored, then they can be combined into a single abstract operator for moving an object between rooms. Fifth, as pointed out by Tenenberg [1988], performing any inferencing or theorem proving in a relaxed model may result in inconsistencies. The problem is that by ignoring only applicability conditions, operators can be applied in situations for which they were not intended and produce contradictory states. Reduced models avoid this problem by removing the operators and conditions that are not relevant to the current model.

While there are advantages of reduced models over relaxed models, they are similar in that the same basic techniques for generating and using abstractions apply to either model. Relaxed and reduced models are both *homomorphisms* [Korf, 1980] of a problem space, which means that information is discarded in the process of constructing these models. As such, after a problem is solved in either type of abstract space, the abstract solution must be refined in the original space in order to ensure that the solution applies to the original problem. The remainder of this thesis assumes that the abstraction spaces are reduced models where the language of the abstraction space is a subset of the language of the original problem space. However, the abstraction properties and the algorithms for both generating and using abstractions apply directly to relaxed models.

### 3.1.2 Hierarchies of Abstraction Spaces

An ordered sequence of abstraction spaces defines an *abstraction hierarchy*, where each successive abstraction space is an abstraction of the previous one. Since an abstraction space is formed by removing literals from the original problem space, an abstraction hierarchy can be represented by assigning each literal in the domain a number to indicate the *abstraction level* of the literal. The level $i$ abstraction space is identical to the original problem space, except operators and states will only refer to literals that have an abstraction level of $i$ and higher. Level 0 is the original problem space, also called the *ground space* or *base space*. The hierarchy is ordered such that the most abstract space (i.e., problem space with the fewest literals) is placed at the top of the hierarchy, and the ground space is placed at the bottom of the hierarchy. For any sufficiently rich problem space, there can be many different abstraction hierarchies, some more useful than others.

Formally, a $k$-level abstraction hierarchy is defined by the initial problem space $\Sigma = (L, S, O)$, where $L$, $S$, and $O$ are just as in the problem-space definition in Chapter 2, and a function *Level* which assigns one of the first $k$ non-negative integers to each literal in $L$.

$$\forall l \in L \;\; Level(l) = i, \text{ where } i \in \{0, 1, \ldots, k-1\}$$

The function *Level* defines an abstract problem space for each level $i$, where all conditions assigned to a level below $i$ are removed from the language, states, and operators:

$$\Sigma^i = (L^i, S^i, O^i).$$

Given the function *Level* an abstraction space $\Sigma^i$ is constructed from a problem space $\Sigma$ as follows. The language $L^i$ contains the literals in $L$ that are in level $i$ or greater.

$$L^i = \{l | (l \in L) \wedge (Level(l) \geq i)\}$$

Let $\mathcal{M}_s^i : S \rightarrow S^i$ be a *state mapping* function that maps a base-level state into an abstract state by removing literals that are not in the abstract language $L^i$. Thus, $s^i = \mathcal{M}_s^i(s)$ if and only if

$$s^i = \{x | (x \in s) \wedge (x \in L^i)\}.$$

Given the function $\mathcal{M}_s^i$, the states in $S^i$ are the abstract states that corresponds to the states in $S$.

$$S^i = \{s^i | (s \in S) \wedge (s^i = \mathcal{M}_s^i(s))\}$$

Let $\mathcal{M}_o^i : O \rightarrow O^i$ be a *operator mapping* function that maps a base-level operator into an abstract operator by removing the literals in the preconditions, deletes, and

adds that are not in the abstract language $L^i$. Thus, $\alpha^i = \mathcal{M}_o^i(\alpha)$ if and only if

$$\begin{aligned} \alpha^i &= (P_{\alpha^i}, D_{\alpha^i}, A_{\alpha^i}) \wedge \\ P_{\alpha^i} &= \{x | (x \in P_\alpha) \wedge (x \in L^i)\} \wedge \\ D_{\alpha^i} &= \{x | (x \in D_\alpha) \wedge (x \in L^i)\} \wedge \\ A_{\alpha^i} &= \{x | (x \in A_\alpha) \wedge (x \in L^i)\}. \end{aligned}$$

Given the function $\mathcal{M}_o^i$, the operators in $O^i$ are the abstract operators that correspond to the operators in $O$ that have nonempty effects in the abstract space.

$$O^i = \{\alpha^i | (\alpha \in O) \wedge (\alpha^i = \mathcal{M}_o(\alpha)) \wedge (A_{\alpha^i} \neq \{\} \vee D_{\alpha^i} \neq \{\})\}$$

Consider the example robot-planning problem space and an abstraction of that problem space, which were described in Section 3.1.1. The definition of the ground-level problem space $\Sigma^0$ is shown in Table 3.1. $L$ defines the language, $S$ defines the four possible states, and $O$ defines the three operator schemas for the problem space. An abstraction of this problem space is formed by dropping all of the conditions involving door status. This corresponds to the following definition of the *Level* function:

level(`(inroom roomA)`) = 1,
level(`(inroom roomB)`) = 1,
level(`(door roomA doorAB)`) = 1,
level(`(door roomB doorAB)`) = 1,
level(`(open doorAB)`) = 0,
level(`(closed doorAB)`) = 0.

The resulting abstraction space $\Sigma^1$ is shown in Table 3.2. In the abstract space, the abstract language $L^1$ consists of only `inroom` and `door` conditions, the abstract states $S^1$ consists of the two states that correspond to the possible rooms the robot could be in, and the abstract operators $O^1$ consists of the operators for moving between the two rooms. In practice, a problem space is usually defined by specifying only the operators, and an abstraction hierarchy is defined by assigning levels to each of the literals in the problem-space language. The language and states of a problem space are defined implicitly by the operators and problems to be solved.

## 3.2 Hierarchical Problem Solving

This section defines a hierarchical problem-solving method, building on the problem-solving definition in Section 2.2. First two-level hierarchical problem solving is defined, and then this definition is extended to multi-level hierarchical problem solving. This section also analyzes the completeness and correctness of this problem solving method.

$\Sigma^0 = (L, S, O)$

```
L = {(inroom roomA)(inroom roomB)
     (door roomA doorAB)(door roomB doorAB)
     (open doorAB)(closed doorAB)}

S = {((inroom roomA)(open doorAB)(door roomA doorAB)(door roomB doorAB))
     ((inroom roomA)(closed doorAB)(door roomA doorAB)(door roomB doorAB))
     ((inroom roomB)(open doorAB)(door roomA doorAB)(door roomB doorAB))
     ((inroom roomB)(closed doorAB)(door roomA doorAB)(door roomB doorAB))}

O = {(Move_Thru_Door (room-x room-y)
          (preconds (and (inroom room-x)
                         (door room-x door-xy)
                         (door room-y door-xy)
                         (open door-xy)))
          (effects ((delete (inroom room-x))
                    (add (inroom room-y)))))

     (Open_Door (door-xy)
          (preconds (and (door room-x door-xy)
                         (inroom room-x)
                         (closed door-xy)))
          (effects ((delete (closed door-xy))
                    (add (open door-xy)))))

     (Close_Door (door-xy)
          (preconds (and (door room-x door-xy)
                         (inroom room-x)
                         (open door-xy)))
          (effects ((delete (open door-xy))
                    (add (closed door-xy)))))}
```

Table 3.1: Definition of an Example Problem Space

### 3.2.1 Two-level Hierarchical Problem Solving

A hierarchical problem solver is given a problem space, a problem to be solved in that space, and an abstraction hierarchy. In two-level problem solving there are only two levels to the hierarchy: the *ground space* and an *abstraction space*. The problem solver maps the given problem into the abstraction space (by deleting literals that are not

$\Sigma^1 = (L^1, S^1, O^1)$

$L^1 =$ {(inroom roomA)(inroom roomB)
(door roomA doorAB)(door roomB doorAB)}

$S^1 =$ {((inroom roomA)(door roomA doorAB)(door roomB doorAB))
((inroom roomB)(door roomA doorAB)(door roomB doorAB))}

$O^1 =$ {(Move_Thru_Door (*room-x room-y*)
(preconds (and (inroom *room-x*)
(door *room-x door-xy*)
(door *room-y door-xy*)))
(effects ((delete (inroom *room-x*))
(add (inroom *room-y*)))))}

Table 3.2: Definition of an Example Abstraction Space

part of the abstraction space), solves the abstract problem, uses the abstract solution to form subproblems that are then solved in the ground space (by reintroducing the deleted literals).

Given an abstraction space $\Sigma^A$, the first step is to map the original problem into an abstract problem. (Note that since there are only two levels in the abstraction hierarchy the superscript $A$ is used to refer to terms in the abstract space.) The function $\mathcal{M}_s^A$, which was defined in the previous section, is used to map the initial and goal states $S_0$ and $S_g$ into abstract states $S_0^A$ and $S_g^A$.

$$S_0^A = \mathcal{M}_s^A(S_0)$$

$$S_g^A = \mathcal{M}_s^A(S_g)$$

The relationship between the states or operators in the ground space and states or operators in the abstract space is a many-to-one mapping since states or operators that differ in the base space may be indistinguishable in the abstract space. Figure 3.2 shows the mapping of the initial and goal states into the corresponding abstract states, and the mapping of the base-level operators into the corresponding abstract operators.

The next step is to use the problem-solving procedure $\mathcal{P}$, as defined in Chapter 2, to find a plan $\Pi^A$ in the abstract space that transforms the initial state $S_0^A$ into final state $S_n^A$ that satisfies the goal $S_g^A$.

$$\Pi^A = \mathcal{P}(\Sigma^A, S_0^A, S_g^A)$$

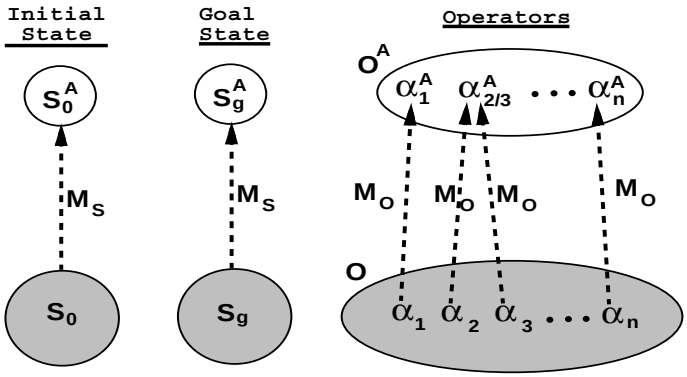


Figure 3.2: Mapping a Problem into an Abstract Problem

The solution to the abstract problem defines a set of intermediate abstract states. The intermediate states can be found by decomposing the abstract plan $\Pi^A$ into its component operators and using the application function $\mathcal{A}$ to apply each of these operators to successive states starting with the initial abstract state $S_0^A$.

$$\Pi^A \equiv \alpha_1^A \| \cdots \| \alpha_n^A$$

$$S_i^A = \mathcal{A}(\alpha_i^A, S_{i-1}^A) \; ; \; 1 \leq i \leq n$$

Since the language of the abstract space is a subset of the language of the base space, the intermediate abstract states can be used directly as intermediate goals in the base space (recall that a goal is a partial specification of a state). These goals define a set of subproblems that can be solved sequentially. Figure 3.3 shows the abstract solution and the intermediate goal states formed from the solution.

A problem solver would then solve each of the intermediate subproblems in the following order. First, a problem solver searches for a plan $\Pi_1$ that transforms the

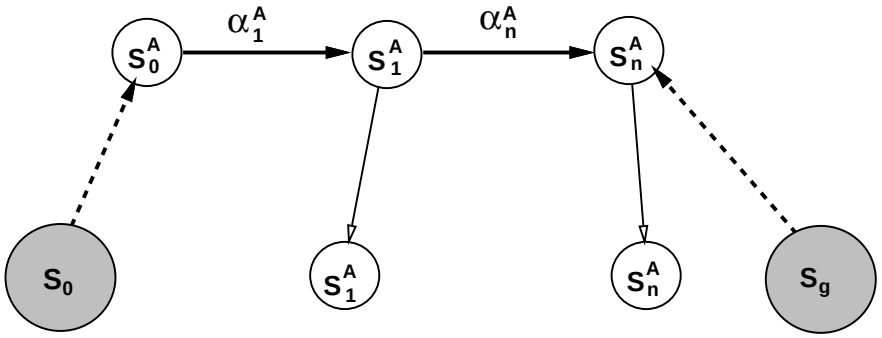


Figure 3.3: Using an Abstract Solution to Form Subproblems

initial state into the first intermediate goal state $S_1^A$. This is shown in Figure 3.4, where the specialization of the abstract state $S_1^A$ is $S_1$. State $S_1$ then serves as the initial state for the next subproblem. Next, a problem solver searches for a plan that transforms the resulting state $S_1$ into the next intermediate goal state $S_2^A$. This process is repeated for each of the intermediate states in the abstract space up to $S_n^A$.

$$\Pi_i = \mathcal{P}(\Sigma, S_{i-1}, S_i^A) \; ; \; 1 \leq i \leq n$$

$$S_i = \mathcal{A}(\Pi_i, S_{i-1}) \; ; \; 1 \leq i \leq n$$

The final step to produce a solution in the base space requires mapping the state $S_n$, which only satisfies the abstract goal $S_g^A$, into a state that satisfies the original goal $S_g$.

$$\Pi_g = \mathcal{P}(\Sigma, S_n, S_g)$$

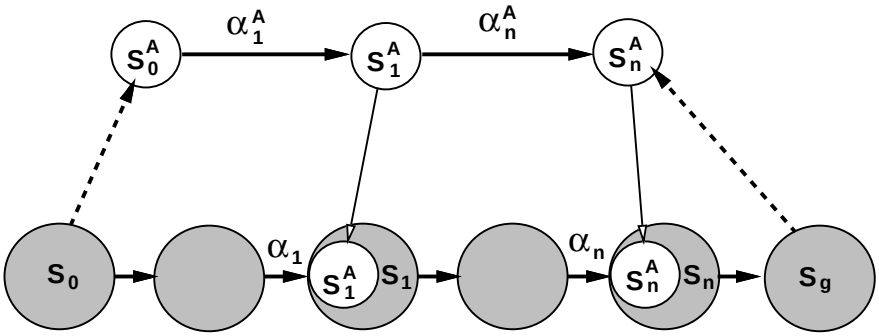


Figure 3.4: Solving the Subproblems in the Ground Space

In solving each of the subproblems, the abstract solution constrains the possible operators for the final operator in each of the subproblem solutions. In particular, since each operator in the abstract space is an abstraction of one or more base-space operators, the final operator in the solution sequence of each subproblem will be a specialization of the corresponding abstract operator. For example, in the diagram in Figure 3.4, $\alpha_1$ must be a specialization of $\alpha_1^A$ and $\alpha_n$ must be a specialization of $\alpha_n^A$. In general, given an abstract plan $\Pi^A \equiv \alpha_1^A \| \cdots \| \alpha_n^A$, then the following condition must hold for each ground-level subplan $\Pi_i \equiv \alpha_1 \| \cdots \| \alpha_n$ that achieves the abstract intermediate state $S_i^A$:

$$\alpha_i^A = \mathcal{M}_o^A(\alpha_n).$$

Since the mapping of the operators in the abstract space to operators in the base space is a one-to-many mapping, there may be several ways to specialize the abstract operators.

The final solution to the original problem is simply the concatenation of the solutions to all of the subproblems.

$$\Pi = \Pi_1 \| \Pi_2 \| \cdots \| \Pi_n \| \Pi_g$$

Figure 3.4 shows the final solution where the intermediate states in the plan must satisfy the intermediate abstract states, and the operators applied to reach these states are specializations of the intermediate abstract operators.

Hierarchical problem solving divides a problem into subproblems to reduce the size of the search spaces, but it does not completely eliminate the search. To solve a problem in the most abstract space will require searching for a sequence of operators in that space. The resulting abstract solution then defines a set of subproblems. Each of these subproblems will require an additional search. As shown in Figure 3.5, solving each of these subproblems involves both selecting a specialization of the abstract operator and then searching for a state in which this operator can be applied. The general idea is to replace the initial, potentially enormous search space with many smaller, more constrained search spaces, as described in Section 3.3.

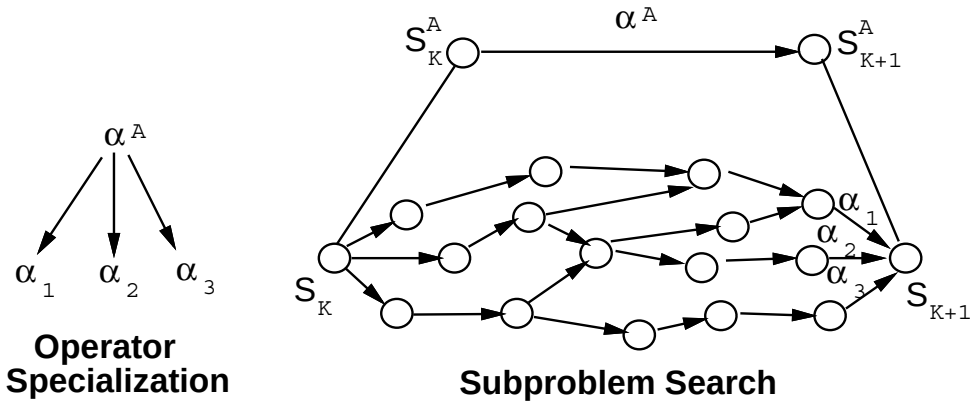


Figure 3.5: Search Space of a Subproblem

If the problem solver is unable to solve one of the subproblems, then it must backtrack to consider other ways of solving previous subproblems as well as other ways of solving the subproblems in more abstract spaces. During problem solving, there will be choices of which goal to work on next, which operator to use to achieve the goal, and which bindings of the operator to use. Each of these choices must be recorded during problem solving and upon failure the problem solver will need to return to these choices to try the alternatives. Thus, it may be necessary to backtrack within a particular subproblem, across subproblems, and across abstraction levels.

### 3.2.2 Multi-Level Hierarchical Problem Solving

Two-level hierarchical problem solving is easily extended to multiple levels. As shown in Figure 3.6, instead of a single abstraction space, there is a hierarchy of abstraction spaces. Problem solving using a hierarchy of abstraction spaces proceeds as follows. First, given a problem to solve in the ground space, the problem is mapped into the most abstract space in the hierarchy and solved in that space. Next, as in the two-level problem solving, the intermediate states are used to form the subproblems at the next level in the hierarchy. Each of these subproblems are solved and the solution to all of the subproblems are concatenated together to form the abstract plan at that level. Each of the intermediate states of the resulting abstract plan are then used to form subproblems at the next level. This process continues until the plan is refined all the way back to the ground space.

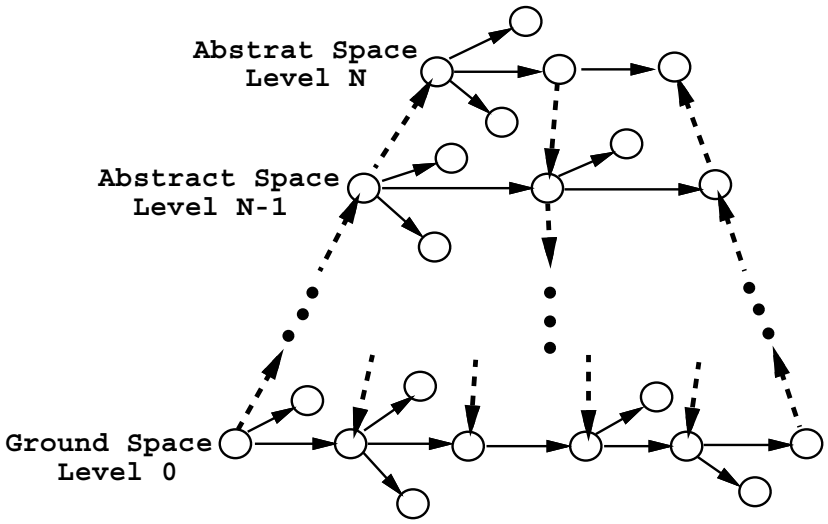


Figure 3.6: Multi-Level Hierarchical Problem Solving

### 3.2.3 Correctness and Completeness

A problem solver $\mathcal{P}$ is *correct* if every plan produced by $\mathcal{P}$ is correct, and $\mathcal{P}$ is *complete* if, given a problem that has a solution, $\mathcal{P}$ is guaranteed to terminate with the solution.

Given a correct and complete nonhierarchical problem solver $\mathcal{P}$, the hierarchical problem solver $\mathcal{H}$ described in the previous section is correct, but it is only complete for a certain class of problems and abstraction hierarchies. This section proves the correctness of $\mathcal{H}$, describes the class of problems for which $\mathcal{H}$ is complete, proves this

restricted completeness, and describes the tradeoffs in building a complete hierarchical problem solver.

**Theorem 3.1 (Correctness)** *Given a correct problem solver $\mathcal{P}$, any plan $\mathcal{H}$ produces will solve the problem.*

Proof: Every plan produced by $\mathcal{H}$ consists of the concatenation of solutions to all of the subproblems in the ground space. Since the solution to each of these subproblems is produced by $\mathcal{P}$, and $\mathcal{P}$ is assumed to be correct, the plans to solve the individual subproblems must be correct. The subproblems are concatenated together to produce the final plan. This resulting plan is a legal plan since the final state in the solution to each subproblem is used as the initial state for the following subproblem. The resulting plan correctly solves the given problem since the initial state of the first subproblem is the initial state of the given problem and the goal state of the last subproblem is the goal state of the given problem. □

Given a problem that is solvable, the hierarchical problem solver $\mathcal{H}$ is guaranteed to terminate with a solution if the problem is both *decomposable* and *linearizable* relative to the abstraction hierarchy.

A problem is *decomposable* relative to the abstraction hierarchy if it can be solved without interleaving the goals that arise in separate subproblems during hierarchical problem solving. This restriction stems from the fact that the hierarchical problem solver takes the abstract solution and partitions it into subproblems that are solved separately. There is no facility for interleaving the goals that arise in the various subproblems. If a problem is linear, which means it can be solved without interleaving any of the subgoals of a problem, it is sufficient to guarantee that the problem is decomposable relative any abstraction hierarchy. However, this is not a necessary condition since a nonlinear subproblem can be solved by the nonhierarchical problem solver $\mathcal{P}$, which is assumed to be complete. (See [Joslin and Roach, 1989] for a precise characterization of linear and nonlinear problems.)

A problem is *linearizable* relative to an abstraction hierarchy if every conjunctive set of goals that arises while solving the problem can be solved in the order that the goals appear in the levels of the hierarchy. (The hierarchy orders the goals from most abstract to least abstract.) This restriction arises because for any conjunction of goals, the hierarchical problem solver first solves those goals in the most abstract space, and then those in the next space, and so on without considering all possible orderings of those goals. Note that this restriction only applies to those goals that arise as a conjunctive set of goals, either as a conjunction of top-level goals or as a conjunctive set of preconditions to an operator.

Consider an example from the STRIPS robot planning domain [Fikes and Nilsson, 1971]. In this domain a robot can move between rooms, pushing boxes and opening

and closing doors. An abstraction hierarchy in this domain might consist of two levels, where conditions dealing with the locations of boxes are dealt with at one level, and conditions dealing with door status are dealt with at the next level. A problem is decomposable relative to this abstraction hierarchy if all the door-status goals can be solved independently. A problem is linearizable relative to this abstraction hierarchy if a set of goals can be solved by first solving the goals involving box location and then solving the goals involving door status. If the problem solver was given a problem consisting of four goals, two goals involving box location and two goals involving door status, it would solve the box location goals first, considering either ordering of these two goals, and then solve the door status goals next, considering any order of these two goals.

**Theorem 3.2 (Completeness)** *Given a complete problem solver $\mathcal{P}$, the hierarchical problem solver $\mathcal{H}$ is complete for any problem that is both decomposable and linearizable relative to the given abstraction hierarchy,*

Proof: Since a complete problem solver $\mathcal{P}$ is used to solve any subproblem within an abstraction level, any incompleteness in $\mathcal{H}$ could only be due to the partitioning and ordering of the subproblems that arise during problem solving. Given that a problem is decomposable, the partitioning of a problem into subproblems would not prevent the problem from being solved. Similarly, since the problem is linearizable, then any conjunction of goals can be solved in the order imposed by the hierarchy. Thus, the partitioning and ordering imposed by $\mathcal{H}$ could not prevent a problem from being solved. □

The decomposability restriction on a problem is no stronger than the usual assumptions that are made to show that the complexity of a problem can be reduced by identifying intermediate states [Minsky, 1963, Simon, 1977]. These analyses always assume that the problem can be divided into a number of smaller subproblems. As shown in the next section, dividing up the problem into independent subproblems is central to reducing the complexity of a problem. In addition, the decomposability restriction is not an issue if the single-level problem solver $\mathcal{P}$ is linear. A linear problem solver can only solve the class of linear problems. In this case, the only added restriction imposed by the hierarchical problem solver $\mathcal{H}$ is that the problems are linearizable relative to the abstraction hierarchy. This simply follows from the fact that if a problem is linear then it is decomposable.

The linearizability restriction on problems is also a fairly weak restriction. The next chapter describes an algorithm for generating abstractions, and the algorithm guarantees that the goals at one level in the hierarchy could not possibly violate the conditions at a more abstract level. The only cases where a problem might not be linearizable results from indirect goal interactions, where a goal for achieving a

condition at one level in the hierarchy deletes some condition needed to achieve a goal at a lower level in the hierarchy, and this needed condition cannot be reachieved. While this type of interaction can arise in practice, it did not prevent any problems from being solved in any of the domains explored in this thesis.

The hierarchical problem solver described in this chapter could be made complete. This would require a more flexible refinement mechanism that allowed the steps needed to refine a plan to be inserted anywhere in the abstract plan. Given this more general refinement mechanism, a problem could no longer be decomposed into subproblems, but the use of abstraction could still focus the problem solver on the more difficult aspects of a problem first. The disadvantage of this approach is that the worst-case complexity of hierarchical problem solver will be increased because the problems can no longer be decomposed into smaller independent subproblems.

## 3.3 Analysis of the Search Reduction

This section presents a complexity analysis of single-level problem solving, two-level hierarchical problem solving, and multi-level hierarchical problem solving [Knoblock, 1991]. The last part of this section identifies under precisely what assumptions hierarchical problem solving can reduce an exponential search to a linear one. Since the size of the search spaces are potentially infinite, the analysis assumes the use of a brute-force search procedure that is bounded by the length of the solution (e.g., depth-first iterative-deepening [Korf, 1985a]).

The analysis is similar to the analysis of abstraction planning with macros by Korf [1987]. Korf showed that the use of a hierarchy of macros can reduce an exponential search to a linear one. However, Korf's analysis applies to abstraction planning with macros and not to hierarchical problem solving because it makes a number of assumptions that do not hold for hierarchical problem solving. The most significant assumption that prevents Korf's analysis from applying to hierarchical problem solving is that it assumes that when the abstract problem is solved, the original problem is solved. The difficult part of solving a problem using macros is finding a path from a state in the base space to a state in the abstract space. Once a path to the abstract states has been found, the problem can be completely solved in the macro space. In contrast, using hierarchical problem solving it is straightforward to map from a state in the base space to a state in the abstract space. However, once the problem has been solved in the abstract space, the abstract solution must be refined into a solution in the base space.

### 3.3.1 Single-Level Problem Solving

For single-level problem solving, if a problem has a solution of length $l$ and the search space has a branching factor $b$, then in the worst-case the size of the search space is $\sum_{i=1}^{l} b^i$. Thus, the worst-case complexity of this problem is $O(b^l)$.

### 3.3.2 Two-Level Hierarchical Problem Solving

Let $k$ be the ratio of the solution length in the base space to the solution length in the abstract space. Thus, $\frac{l}{k}$ is the solution length in the abstract space. Since each operator in the abstract space corresponds to one or more operators in the ground space, the branching factor of the abstract space is bounded by the branching factor of the ground space, $b$. To simplify the analysis, $b$ is used as the branching factor throughout. The size of the search tree in the abstract space is $\sum_{i=1}^{l/k} b^i$, which is $O(b^{\frac{l}{k}})$. In addition, the analysis must include the use of this abstract solution to solve the original problem.

The abstract solution defines $\frac{l}{k}$ subproblems. The size of each problem is the number of steps (solution length) in the base space required to transform an initial state $S_i$ into a goal state $S_{i+1}$, which is represented as $d(S_i, S_{i+1})$. Thus, the search in the base space is:

$$\sum_{i=1}^{d(S_0,S_1)} b^i + \sum_{i=1}^{d(S_1,S_2)} b^i + \cdots + \sum_{i=1}^{d(S_{\frac{l}{k}-1},S_{\frac{l}{k}})} b^i, \tag{3.1}$$

which is $O(\frac{l}{k} b^{d_{\max}})$, where

$$d_{\max} \equiv \max_{0 \le i \le \frac{l}{k}-1} d(S_i, S_{i+1}). \tag{3.2}$$

In the ideal case, the abstract solution will divide the problem into subproblems of equal size, and the length of the final solution using abstraction will equal the length of the solution without abstraction. In this case, the abstract solution divides the problem into $\frac{l}{k}$ subproblems of length $k$.

$$b^{d_{\max}} = b^{\frac{l}{l/k}} = b^k \tag{3.3}$$

Assuming that the planner can first solve the abstract problem and then solve each of the problems in the base space without backtracking across problems, then the size of the space searched in the worst case is the sum of the search spaces for each of the problems.

$$\sum_{i=1}^{\frac{l}{k}} b^i + \frac{l}{k} \sum_{i=1}^{k} b^i \tag{3.4}$$

The complexity of this search is: $O(b^{\frac{l}{k}} + \frac{l}{k}b^k)$. The high-order term is minimized when $\frac{l}{k} = k$, which occurs when $k = \sqrt{l}$. Thus, when $k = \sqrt{l}$, the complexity is $O(\sqrt{l}\ b^{\sqrt{l}})$, compared to the original complexity of $O(b^l)$.

### 3.3.3 Multi-Level Hierarchical Problem Solving

Korf [1987] showed that a hierarchy of macro spaces can reduce the expected search time from $O(s)$ to $O(\log s)$, where $s$ is the size of the search space. This section proves an analogous result – that multi-level hierarchical problem solving can reduce the size of the search space for a problem of length $l$ from $O(b^l)$ to $O(l)$, where $b^l$ is the size of the search space.

In general, the size of the search space with $n$ levels (where the ratio between the levels is $k$) is:

$$\sum_{i=1}^{\frac{l}{k^{n-1}}} b^i + \frac{l}{k^{n-1}}\sum_{i=1}^{k} b^i + \frac{l}{k^{n-2}}\sum_{i=1}^{k} b^i + \frac{l}{k^{n-3}}\sum_{i=1}^{k} b^i + \cdots + \frac{l}{k}\sum_{i=1}^{k} b^i \tag{3.5}$$

The first term in the formula accounts for the search in the most abstract space. Each successive term accounts for the search in successive abstraction spaces. Thus, after solving the first problem, there are $\frac{l}{k^{n-1}}$ subproblems that will have to be solved at the next level. Each of these problems are of size $k$ since $k$ is the ratio of the solution lengths between adjacent abstraction levels. At the next level there are $\frac{l}{k^{n-2}}$ subproblems ($k\frac{l}{k^{n-1}}$) each of size k, and so on. In the final level there are $\frac{l}{k}$ subproblems each of size k. The final solution will therefore be of length $\frac{l}{k}k = l$.

The maximum reduction in search can be obtained by setting the number of levels $n$ to $\log_k(l)$, where the base of the logarithm is the ratio between levels $k$. Substituting $\log_k(l)$ for $n$ in Formula 3.5 produces the following formula:

$$\sum_{i=1}^{k} b^i + k\sum_{i=1}^{k} b^i + k^2\sum_{i=1}^{k} b^i + k^3\sum_{i=1}^{k} b^i + \cdots + k^{\log_k(l)-1}\sum_{i=1}^{k} b^i \tag{3.6}$$

From Formula 3.6, it follows that the complexity of the search is:

$$O((1 + k + k^2 + \cdots + k^{\log_k(l)-1})b^k). \tag{3.7}$$

The standard summation formula for a finite geometric series with $n$ terms, where each term increases by a factor of $k$, is:

$$1 + k + k^2 + \cdots + k^n = \frac{k^{n+1} - 1}{k - 1}. \tag{3.8}$$

Given Formula 3.8, it follows that the complexity of Formula 3.7 is $O(\frac{l-1}{k-1}b^k)$:

$$(1 + k + k^2 + \cdots + k^{\log_k(l)-1})b^k = \frac{k^{\log_k(l)} - 1}{k-1}b^k = \frac{l-1}{k-1}b^k. \tag{3.9}$$

Since $b$ and $k$ are assumed to be constant for a given problem space and abstraction hierarchy, the complexity of the entire search space is $O(l)$.

The analysis above shows that hierarchical problem solving can reduce the size of the search space from $O(b^l)$ to $O(l)$. This analysis assumes the best-case for the distribution and independence of problems in the hierarchy, but assumes the worst-case for search in each of the subproblems. The best-case assumptions are reviewed below. In practice, the size of the actual search space is between the two extremes.

### 3.3.4 Assumptions of the Analysis

1. *The number of abstraction levels is* $\log_k$ *of the solution length.* This assumption argues for problem-specific abstraction hierarchies over domain-specific hierarchies since the solution length of problems within a given domain can vary greatly from problem to problem.

2. *The ratio between the levels is the base of the logarithm,* $k$. A problem should be divided such that the length of the solution at each level increases linearly.

3. *Problems are decomposed into subproblems that are all of equal size.* The analysis assumes that the size of all the subproblems is the same in order to minimize $b^{d_{\max}}$. If all the other assumptions hold, the complexity of the search will be the complexity of the largest subproblem in the search. For example, if $b$ is constant and the largest subproblem is $b^{\frac{l}{2}}$, hierarchical problem solving would still reduce the search from $O(b^l)$ to $O(b^{\frac{l}{2}})$.

4. *The solutions produced by the hierarchical problem solver are the shortest ones possible.* If a problem has a solution of length $l$, then the length of the solution produced using hierarchical problem solving must also be $l$, or at least within a constant of $l$.

5. *There is only backtracking within subproblems.* This requires that an abstraction level can be refined into a solution at lower levels and that the solution to each of the subproblems within an abstraction level will not prevent any of the remaining subproblems at the same level from being solved.

The assumptions above are sufficient to produce an exponential-to-linear reduction in the size of the search space. The essence of the assumptions is that the abstraction

divides the problem into $O(l)$ constant size subproblems that can be solved serially. Of course, these assumptions are unlikely to hold in many domains and, if they do hold, it may not be possible to determine that fact *a priori*. For example, determining whether an abstract solution can be refined without backtracking requires determining whether plans exist to solve the subproblems, which in general will require solving the subproblems.

Consider the effect of weakening the various assumptions above. Assumptions 1, 2, and 3 divide up the problem into the optimal number of optimal size subproblems to reduce the complexity of the search. If any of these assumptions are weakened, the resulting complexity will depend on the complexity of the largest subproblem and the total number of subproblems. Thus, if the number of subproblems and the complexity of the largest subproblem are not much smaller than the complexity of the original problem, the abstractions will not provide a significant benefit.

Assumption 4 requires that the final solution is the shortest one. This assumption is needed to bound the potentially infinite search spaces. Producing optimal solutions usually requires an admissible search (e.g., breadth-first or depth-first iterative deepening) and with hierarchical problem solving there is no guarantee that the final solution will be optimal, only that the solution to the individual subproblems are optimal. The analysis holds as long as the final solution is within a constant of the optimal solution.

Assumption 5 requires that the problem can be broken up into subproblems that can be solved in order without backtracking. If this assumption does not hold, then some or all of the benefit of the abstraction could be lost since the worst-case complexity is no longer the sum of the subproblems, but the product. Note, however, that the structure of hierarchical problem solving can minimize the impact of backtracking in several ways. First, on backtracking across abstraction levels, it is not necessary to backtrack through every choice point in an abstract plan, but only those choice points in an abstract plan that preceded the failure point in the refinement of the abstract plan. Second, when refining the abstract plan after backtracking across levels, the detailed solution to the parts of the problem that precede the modification to the abstract plan remain valid and do not need to be replanned. Section 3.5.5 provides a detailed description of these two techniques.

## 3.4 Tower of Hanoi Example

This section describes an abstraction hierarchy for the Tower of Hanoi, explains how hierarchical problem solving can be used to solve this problem, and then shows that this approach reduces the size of the search space from exponential to linear in the solution length for this problem.

Consider the three-disk Tower of Hanoi puzzle, which was described in Section 2.2. A good abstraction of the problem, which was first identified by Korf [1980], is to separate the disks into different abstraction levels. The resulting abstraction hierarchy is shown in Figure 3.7, where the most abstract space contains only the largest disk, the next abstraction space contains the largest and medium size disk, and the ground space contains all three disks.

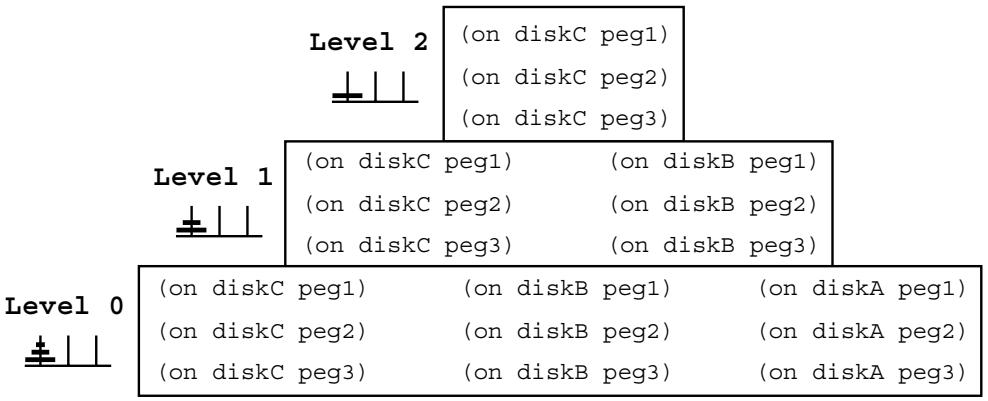


Figure 3.7: Abstraction Hierarchy for the Tower of Hanoi

Each abstraction space is formed by dropping all the literals that are not in the given level of the hierarchy from the initial state, goal, and operators. Table 3.3 shows, for an example initial state, goal, and operator, the conditions at each level of abstraction. Level 0 shows the initial specification, level 1 shows the conditions remaining after removing the smallest disk, and level 2 shows the conditions after removing both the smallest and medium-sized disks.

The abstraction hierarchy for the Tower of Hanoi can be used for hierarchical

| | | | Move DiskC From Peg1 to Peg3 | |
|---|---|---|---|---|
| | **Initial State** | **Goal State** | **Preconds** | **Effects** |
| **Level 2** | (on diskC peg1) | (on diskC peg3) | (on diskC peg1) | ¬(on diskC peg1)<br>(on diskC peg3) |
| **Level 1** | (on diskC peg1)<br>(on diskB peg1) | (on diskC peg3)<br>(on diskB peg3) | (on diskC peg1)<br>¬(on diskB peg1)<br>¬(on diskB peg3) | ¬(on diskC peg1)<br>(on diskC peg3) |
| **Level 0** | (on diskC peg1)<br>(on diskB peg1)<br>(on diskA peg1) | (on diskC peg3)<br>(on diskB peg3)<br>(on diskA peg3) | (on diskC peg1)<br>¬(on diskB peg1)<br>¬(on diskB peg3)<br>¬(on diskA peg1)<br>¬(on diskA peg3) | ¬(on diskC peg1)<br>(on diskC peg3) |

Table 3.3: Abstractions of an Initial State, Goal, and Operator

problem solving. The first step is to map the initial problem into the corresponding abstract problems. This is show in Figure 3.8, where the initial and goal states are mapped into initial and goal states at each level of abstraction. In addition, the operators of the initial problem space are mapped into abstract operators at each level. In the Tower of Hanoi the abstraction of the operators is a one-to-one mapping, where some of the operators are simply not relevant to a given abstraction level.

Figure 3.8: Mapping a Problem into an Abstract Problem

The next step is to solve the problem in the most abstract space. This is shown in Figure 3.9, where there is simply a one step plan that moves the largest disk (`diskC`) from `peg1` to `peg3`. As shown in the figure this creates two new subproblems at the next level, where the first subproblem is to reach the state where the abstract operator can be applied, and the second subproblem is to reach the goal state.

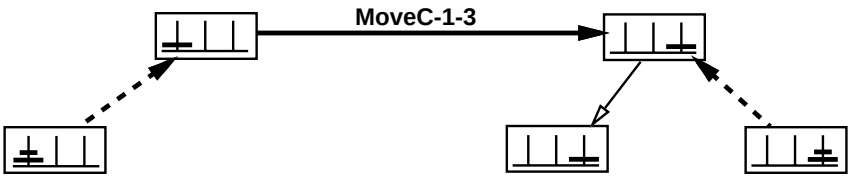


Figure 3.9: Solving an Abstract Problem

This process is continued by solving each of the subproblems at the second level and using the solutions to the subproblems to guide the search in the base level. Figure 3.10 shows the resulting three-step plan at the second level. For each operator in the abstract plan, a specialization of that operator must be used in any refinement of that abstract plan. In this case, `MoveC-1-3` is used in the second step since it is the only specialization of the corresponding abstract operator. Thus, only two additional steps were inserted at this abstraction level.

The final step in the hierarchical problem solving is to solve each of the subproblems in the base space. This produces the seven-step solution shown in Figure 3.11, where four additional steps were inserted at this level.

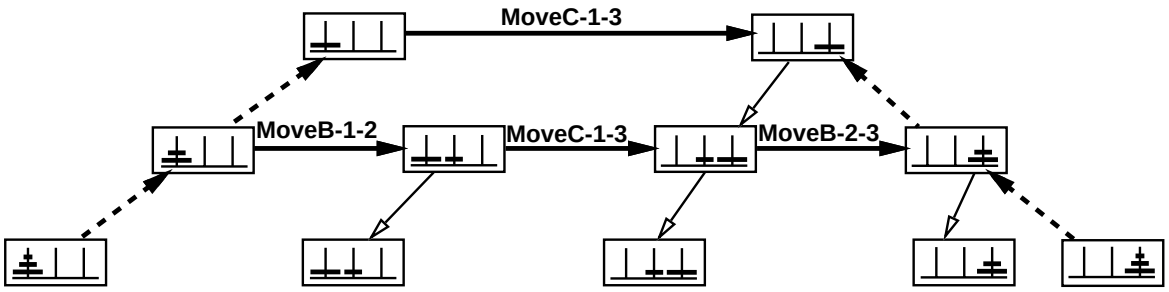


Figure 3.10: Solving the Subproblems at the Next Abstraction Level

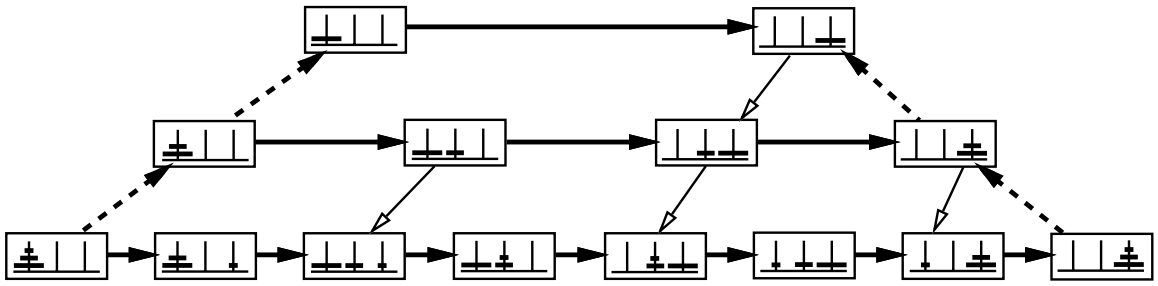

Figure 3.11: Solving the Subproblems in the Ground Space

This abstraction of the Tower of Hanoi is ideal in the sense that it produces a set of abstraction spaces that meet all of the assumptions listed in Section 3.3.4.

1. The number of abstraction levels is $\log_2(l)$, where $l$ is the length of the solution. For a n-disk problem the solution length $l$ is $2^n - 1$, and the number of abstraction levels is $n$, which is $O(\log_2(l))$.

2. The ratio between the levels is the base of the logarithm.. If the number of steps at a given level is $n$, then the number of steps at the next level is $2n + 1$. Thus, the base of the logarithm is 2, and the ratio between the levels is $O(2)$.

3. The problem is decomposed into subproblems that are all of equal size. These subproblems are effectively all of size one, since each subproblem requires inserting one additional step.

4. Using an admissible search strategy, the hierarchical problem solver produces the shortest solution.

5. There is only backtracking within a subproblem. The subproblems is this domain can always be solved in order without backtracking across subproblems or across abstraction levels.

Since these assumptions are sufficient to reduce the size of the search space from exponential to linear in the length of the solution, it follows that the hierarchical problem solving produces such a reduction for the Tower of Hanoi.

An more intuitive explanation for this search reduction is as follows. In the original problem the size of the search space is $O(b^l)$, where $b$ is the branching factor, and $l$ is the length of the solution. The abstraction hierarchy divides up the problem into $l$ subproblems of equal size that can be solved serially. Each step in the final solution will correspond to a subproblem in hierarchical problem solving, so the number of subproblems is $l$. Using an admissible search procedure, each subproblem will be solved in one or two steps. In those subproblems that require two steps, the second step is simply a specialization of the corresponding abstract step. Since each disk can be moved from one of two places, the branching factor is two. Thus, the size of each subproblem is $2^1 = 2$, so the entire search is bounded by $2l$, which is $O(l)$. Consequently, hierarchical problem solving reduces the search space in this domain from $O(b^l)$ to $O(l)$.

## 3.5 Hierarchical Problem Solving in PRODIGY

This section describes an extended version of PRODIGY that performs hierarchical problem solving. The extensions to PRODIGY are straightforward and are based on the formalization of hierarchical problem solving described earlier in this chapter.

### 3.5.1 Architecture

The hierarchical problem solver, which will be referred to as Hierarchical PRODIGY, is shown in Figure 3.12. The problem solver is given the operators that define a problem space, a problem in that problem space, and an abstraction hierarchy to be used to solve the problem. The nonhierarchical version of PRODIGY is employed as a subroutine, where the hierarchical problem solver selects problem spaces and problems for PRODIGY to solve.

Hierarchical PRODIGY uses the abstraction hierarchy to form the problem spaces for each level of problem solving. The system first maps the given problem into the most abstract space and then gives this problem to PRODIGY to solve. If the problem is solvable, PRODIGY returns a solution, which is then used to construct a number of subproblems to be solved at the next level in the abstraction hierarchy. PRODIGY is given each of these subproblems to solve and this process is repeated until the plan is refined into the ground space. If PRODIGY fails to find a solution to any subproblem, the hierarchical problem solver backtracks through the relevant unexplored search paths (e.g., alternative solutions to the abstract problem).

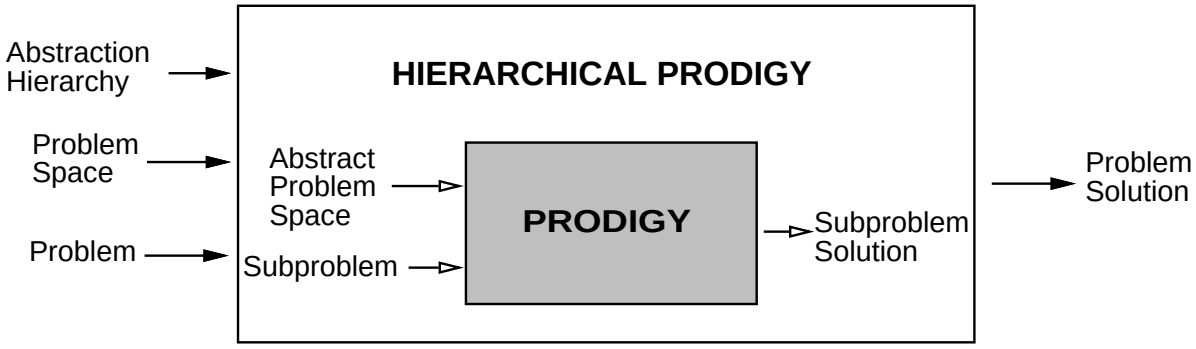


Figure 3.12: Hierarchical Problem-Solving Architecture

### 3.5.2 Representing Abstraction Spaces

To solve a problem in an abstract space requires mapping both the original problem and problem space into a corresponding abstract problem and abstract problem space. As described earlier, a problem is defined by the initial and goal states, and a problem space is defined by the operators for a domain. Given the abstraction hierarchy, which specifies the set of literals that comprise each abstract space, a given problem can be mapped into any level of the hierarchy simply by dropping all of the conditions from the initial state and goal that are not part of that level. Similarly, an abstract problem space can be constructed for each level of the hierarchy by dropping the literals not in the given abstract level from the preconditions and effects of all the operators.

Consider an example from the machine-shop process planning and scheduling domain described in Section 2.3. One possible abstraction of this domain is to separate the process planning from the scheduling by considering only the conditions relevant to planning the operations and ignore the conditions related to scheduling. In the case of the scheduling example, the abstract problem is shown in Table 3.4, where the boxed conditions are dropped in the abstract space. The abstract goal is the same as the original goal since the scheduling is an implicit part of performing an operation. However, the initial state would be simplified as shown, where conditions related to the scheduling are removed.

To form an abstract problem space, all the conditions related to scheduling would be removed from the set of operators. Table 3.5 shows the resulting `turn` operator, where the boxed conditions would be dropped from the operator in the abstract space.

Since the number of abstraction levels can be quite large and it may be necessary to backtrack across abstraction levels, it may be necessary to switch problem spaces frequently. To make this efficient, Hierarchical PRODIGY does not construct an explicit set of abstract operators. Instead, the system dynamically abstracts the

**Goal:** `(and (shape part-a cylindrical)`
`(surface-condition part-a polished))`

**Initial State:** `((shape part-a undetermined)`
`(temperature part-a cold)`
`(is-part part-a)`
`(is-part part-b)`
`(is-part part-c)`
`(last-scheduled part-a time0)`
`(scheduled part-b lathe time1)`
`(scheduled part-b polisher time2)`
`(scheduled part-c roller time1))`

Table 3.4: Abstract Problem in the Machine-Shop Domain

```
(TURN (part time)
  (preconditions
    (and (is-part part)
         (last-scheduled part prev-time)
         (later time prev-time)
         (idle lathe time)))
  (effects
      (delete (shape part old-shape))
      (delete (surface-condition part old-condition))
      (delete (painted part old-paint))
      (delete (last-scheduled part prev-time))
      (add (surface-condition part rough))
      (add (shape part cylindrical))
      (add (last-scheduled part time))
      (add (scheduled part lathe time))))
```

Table 3.5: Abstract Operator in the Machine-Shop Domain

operators during problem solving. At any given time, the system has a list of the literals that are relevant to the given abstraction level. When the system attempts

to match the preconditions of an operator, it only considers those preconditions that are relevant to the given level. Similarly, when the system applies an operator, only those effects that are relevant are applied to the state. This eliminates the need to maintain an explicit set of abstract operators for each abstraction level.

As described in Section 2.3.4, PRODIGY employs a set of control rules to guide the search process. Mapping control rules into an abstract space cannot be done by simply dropping the conditions that are not relevant in that space. The problem is that the selection and rejection control rules are assumed to be correct and, unlike operators, the problem solver does not backtrack over the decisions made by these control rules.[1] Simply dropping conditions from the control rules could result in overly general rules that apply in situations for which they were not intended.

To maintain the correctness of the control rules, they are only applied at a level in the abstraction hierarchy in which all the conditions of the original control rule are contained within the given problem space. The control rules are partitioned into separate abstraction spaces such that each rule is only applied in the most abstract space in which all the conditions of the rule are relevant. Once a rule is applied at a given level, it need not be considered at any lower levels since the abstractions used by the problem solver have the ordered monotonicity property and this property guarantees that a goal that arises at one level cannot arise at a lower level. If a control rule is not specific to a particular goal condition, it is applied at every abstraction level. The partitioning of the control rules is done before problem solving begins, where each control rule is associated with a particular abstraction level (unless the given rule should be applied at every level).

By representing the abstract operators and control knowledge implicitly, Hierarchical PRODIGY makes the switching of abstraction levels a no-cost operation. The system can change to a new level or backtrack across levels frequently without the overhead of constructing a new set of operators and control rules.

### 3.5.3 Producing an Abstract Plan

The first step in hierarchical problem solving is to produce an abstract plan. To do so Hierarchical PRODIGY forms the abstract initial and goal states, sets the current abstraction level to the most abstract level in the hierarchy, and then hands the abstract problem off to PRODIGY to solve. PRODIGY finds a solution to this problem and returns both the solution and the complete problem-solving trace, which can later be used to backtrack efficiently.

[1] Since the problem solver does backtrack over preference rules, the selection and rejection control rules could be changed to preferences rules, allowing the control rules to serve as heuristics in the more abstract levels. This is not done in the current implementation.

Consider the effect of abstraction on the example problem shown in Figure 2.4. The problem is to produce a plan to make a part cylindrical and polished. In the original search space, the mill operation, which makes an part cylindrical, is considered first at time-2, but this prevents the part from being polished at a later time. The system then considered milling the part at later times, but of course they fail for the same reason. Using the abstraction of the problem space that ignores the scheduling of the parts, there are no times, so PRODIGY will first try mill, find that it fails because the part cannot be polished, and then try turning the part on the lathe. The system produces the abstract plan shown in Figure 3.13, which consists of a turn operation, a check to make sure the part can be clamped to the polisher, and then a polish operation. The use of the abstraction separates the process planning from the scheduling task, and thus gains efficiency by eliminating unnecessary backtracking in the ground space.

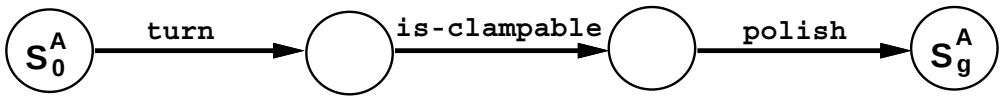


Figure 3.13: Abstract Solution in the Machine-Shop Domain

### 3.5.4 Refining an Abstract Plan

Once the system finds an abstract solution, that solution must then be refined through each of the successively more detailed abstraction levels. The abstract solution defines a sequence of subproblems, where each pair of adjacent intermediate states from the abstract solution forms the initial and goal states for a subproblem at the next level. The problems are solved in order since the final state that satisfies the goal of each subproblem becomes the initial state for the next problem. In addition, the choice of operators used to achieve each intermediate state in the abstract level constrains the choice of operators used to achieve the corresponding goal at the next level.

Returning to the example above, the problem is to take the abstract plan in Figure 3.13 and turn it into a plan that solves the problem in the ground space. Since there are three operations in that plan, Hierarchical PRODIGY generates three subproblems that are handed off to PRODIGY to solve. Each subproblem simply requires determining when to schedule the turn, is-clampable, and polish operations and checking that the required machines and parts are available at the selected times. The resulting refinement of the abstract plan is shown in Figure 3.14.

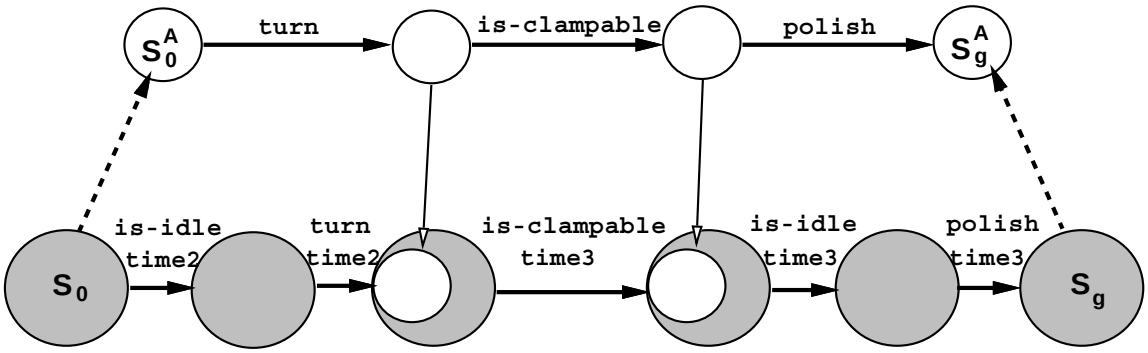


Figure 3.14: Refinement of an Abstract Solution

### 3.5.5 Hierarchical Backtracking

In addition to backtracking within a subproblem, which is taken care of by the backtracking mechanism within PRODIGY, it may also be necessary to backtrack across subproblems and across abstraction levels. The backtracking across subproblems and across abstraction levels, which I call *hierarchical backtracking*, exploits the structure of hierarchical problem solving to implement a simple form of dependency-directed backtracking [Stallman and Sussman, 1977] and to avoid replanning when possible.

Hierarchical backtracking avoids backtracking to choices in the search space that could not be relevant to a problem solving failure. Given an abstract plan, a hierarchical problem solver refines this plan by expanding each of the steps in the abstract plan. If a step in this abstract plan cannot be refined and the problem solver must backtrack, it can ignore any choice points in the abstract plan that occur later in the abstract plan than the step that could not be refined. Completeness can be maintained without backtracking to all of these choice points since they occur after the failure point and could not affect the ability of the problem solver to solve the problem.

In the example shown in Figure 3.14, consider what would happen to the abstract plan if the turn operator selected in the abstract space could not be refined in the ground space. This might occur if the machine was in use during the entire schedule. After trying any choice points earlier in the problem-solving trace at the level in which the failure occurred, the problem solver can backtrack directly to the turn operation in the abstract plan without considering other ways of polishing the part. This is illustrated in Figure 3.15, where the choice points beyond the selection of the turn operation need not be considered for backtracking.

Hierarchical backtracking also avoids unnecessary replanning of the parts of a problem that have not changed during backtracking. When it is necessary to backtrack across abstraction levels because a particular step in a plan cannot be achieved,

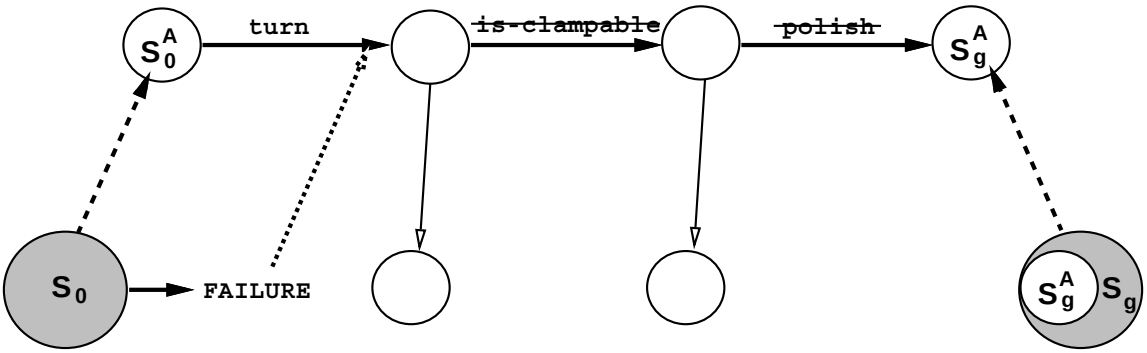


Figure 3.15: Hierarchical Backtracking

if steps earlier in the abstract plan have not changed, then any refinement of those steps can be retained.

Returning to the example, if the problem solver had found that the polish operation could not be refined, the problem solver would have to backtrack to the selection of polish in the abstract plan and solve the problem in a different way. For example, it might be possible to use the grinder instead of the polisher. After this step in the abstract plan is changed, then the plan must be refined. However, it may be possible to reuse much of the work already done to refine the abstract plan. In particular, the refinement of the steps before polish can be retained and only the refinement of the new grind operator needs to be considered. This is illustrated in Figure 3.16, where the part of the plan shown in gray can be reused and only the refinement of the plan after the change needs to be replanned.

Hierarchical PRODIGY can backtrack efficiently when necessary by maintaining the problem-solving traces from each of the solved subproblems, which contain the choice

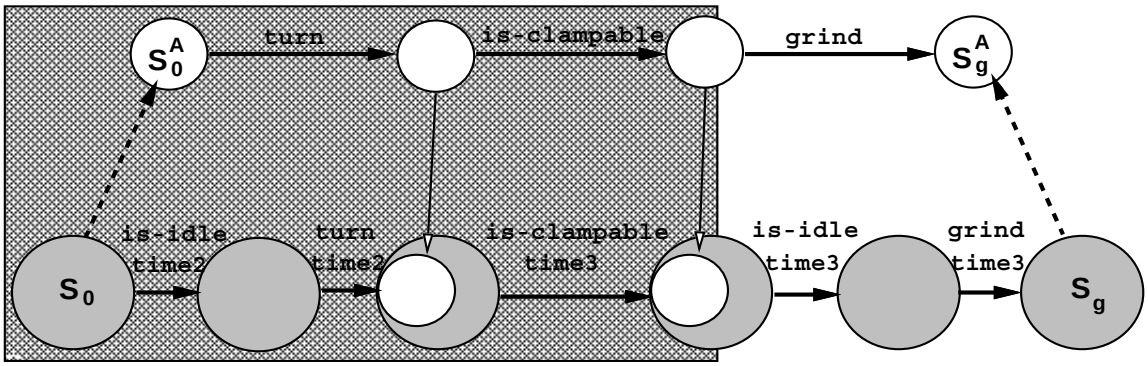


Figure 3.16: Hierarchical Refinement

points that have not been explored. Each of the individual traces for the subproblems are connected with the appropriate links so that backtracking and replanning can be done efficiently. The amount of space required to maintain these links is small compared to the size of the overall search tree.

## 3.6 Discussion

This chapter presented an approach to hierarchical problem solving, showed that this method could provide an exponential-to-linear reduction in search, and described the implementation of hierarchical problem solving in PRODIGY. The chapter defined abstraction spaces and hierarchies, but did not describe what makes one abstraction better than another. In practice, the choice of an abstraction hierarchy for a problem is critical in determining the effectiveness of hierarchical problem solving. A hierarchical problem solver uses plans produced at each level of abstraction to constrain the search at the next level of detail. However, these constraints only force the final plan to go through certain intermediate states and do not actually constrain how much work is done between these states. There is nothing in the problem solving method that prevents the refinement of one part of the abstract plan from undoing the conditions that were achieved in another part of the abstract plan. Eventually these conditions will have to be reachieved, but this could result in more work than if no abstraction were used. The next chapter addresses the issue of what makes a good abstraction for problem solving and presents a method for automatically generating good abstractions.

# Chapter 4

# Generating Abstractions

The previous chapter described how an abstraction hierarchy can be used for problem solving and showed that the use of abstraction can provide a significant reduction in search. However, several important questions have yet to be addressed: What are the characteristics of a useful abstraction? How do we find useful abstraction hierarchies?

This chapter identifies properties of an effective abstraction hierarchy and presents an approach for automatically generating such hierarchies. The chapter first identifies several properties of the relationship between a problem space and one or more abstractions of a problem space, then it gives the basic algorithms for generating hierarchies which satisfy these properties, next it uses the Tower of Hanoi as an example to illustrate the basic approach, and lastly, it describes the implementation of these algorithms in ALPINE.

## 4.1 Properties of Abstraction Hierarchies

In order to generate useful abstraction spaces, it is important to understand how search at the abstract levels constrains search at the ground level. This section first reviews the upward and downward solution properties [Tenenberg, 1988], which relate a problem space to the abstractions of the problem space. While both properties are useful, the downward solution property is too strong to require that it hold for every abstraction. The section then presents two weaker properties, the monotonicity and ordered monotonicity properties, which can be required of an abstraction hierarchy. These properties are first described informally and then defined formally.[1] Along with these definitions, this section provides restrictions on the possible abstraction

[1] The properties and algorithms described in this section are my own [Knoblock, 1990c, Knoblock, 1990b], but the formal definitions in this section were joint work with Josh Tenenberg and Qiang Yang and are also described in [Knoblock *et al.*, 1990, Knoblock *et al.*, 1991b].

hierarchies that are sufficient to guarantee these properties. These restrictions are used in the following section as the basis of an algorithm for automatically generating abstraction hierarchies that satisfy the properties.

### 4.1.1 Informal Description

Tenenberg[1988] identified the upward and downward solution properties, which relate a problem space to an abstract space. The upward solution property is defined as follows:

**Upward Solution Property:** the existence of a ground-level solution implies the existence of an abstract-level solution.

Since any solution at the more constrained ground level will also be a solution in any of the less constrained models, it is clear that an abstraction space will exhibit the upward solution property. The contrapositive of this property is the *downward failure property* [Weld and Addanki, 1990], which states that if there is no solution in the abstract space, then there is no solution in the ground space. This property is useful for determining efficiently that a problem is unsolvable since only the abstract space needs to be searched to prove unsolvability.

The inverse of the upward solution property is the downward solution property, which is defined as follows:

**Downward Solution Property:** the existence of an abstract-level solution implies the existence of a ground-level solution.

Unfortunately, there are few abstraction spaces for which the downward solution property will hold. Since an abstraction space is formed by dropping conditions from the problem space, information is lost and operators in an abstract space can apply in situations in which they would not apply in the original space. In general, using an abstraction space formed by dropping information it is impossible to guarantee this property. The same problem arises in the use of abstraction in theorem proving, where it is called the false proof problem [Plaisted, 1981, Giunchiglia and Walsh, 1990].

Since the downward solution property does not hold in general, there is no guarantee that a refinement of the abstract solution exists. To make matters worse, there are a potentially infinite number of possible refinements of each abstract plan. In general, a refinement of an abstract solution simply requires inserting additional operators to achieve the conditions ignored at a more abstract level. There is nothing to prevent the added operators from undoing the conditions that were achieved in the abstract level and then reachieving these conditions. The result could be that the

problem effectively gets re-solved at each level of abstraction. As Tenenberg[1988, p.75] points out, if the operators in a domain are invertible, then there is no clear criterion for failure in specializing a plan, and a planner could specialize a plan *ad infinitum* simply by inserting increasingly longer solutions between the steps of the abstract plan.

Since the downward solution property is too strong to guarantee, this section defines a weaker property that constrains the refinement of an abstract solution. This property, called the monotonicity property, is defined as follows:

**Monotonicity Property:** the existence of a ground-level solution implies the existence of an abstract-level solution that can be refined into a ground-level solution while leaving the literals *established* in the abstract plan unchanged.

The monotonicity property states that if a solution exists it can be found without modifying an abstract plan in the process of refining that plan. This property captures the idea that an abstract solution should serve as an outline to a ground solution and thus should not be modified in the refinement process.

The monotonicity property is useful because it provides a criterion for backtracking that does not sacrifice completeness. Whenever a problem solver would undo a literal established in an abstract plan while refining the plan, the system can backtrack to a more abstract level instead since the property states that if a problem is solvable, an abstract solution exists that can be refined leaving the abstract plan unchanged. This imposes a strong constraint on how an abstract plan is refined at a lower level. Thus, given an abstraction it provides an approach for using the abstraction space more effectively. In fact, Yang and Tenenberg [1990] designed a nonlinear, least-commitment problem solver that uses the monotonicity property to constrain the search for a refinement of an abstract plan. While the property is useful for constraining the refinement process, it is still rather weak. The next section proves that every abstraction space has this property. As such, it does not provide a criterion for generating useful abstractions.

A restriction of the monotonicity property, called the ordered monotonicity property, does provide a useful criterion for generating abstraction spaces. This property is defined as follows:

**Ordered Monotonicity Property:** *Every* refinement of an abstract plan leaves *all* the literals that comprise the abstract space unchanged.

The ordered monotonicity property is more restrictive than the monotonicity property because it requires that not only does there exist a refinement of an abstract plan that leaves the literals in the abstract plan unchanged, but every refinement of an abstract plan leaves all the literals in the abstract space unchanged. Thus, the property

partitions the problem space such that a plan for achieving the literals at one level will not interact with the literals in a more abstract level. The ordered monotonicity property is useful because it captures a large class of useful abstractions, and ordered monotonic abstractions can be generated from just the initial definition of a problem space.

The ordered monotonicity property requires that every refinement leaves the literals in an abstract space unchanged. One way to construct hierarchies of abstraction spaces that have this property is to partition the literals of a problem space into levels such that any plan to achieve a literal at one level will not interact with literals in a more abstract level. Which literals will potentially interact with other literals can be determined from the operators that define a problem space. A set of constraints can be extracted from the operators that require those literals that could possibly be changed in the process of achieving some other literal to be placed lower or at the same level in the abstraction hierarchy. This set of constraints is sufficient to guarantee the ordered monotonicity property.

Since the interactions between literals depend on the problem, the usefulness of a given abstraction hierarchy not only varies from one domain to another, but also from one problem to another. Thus, instead of attempting to find a single abstraction hierarchy that can be used for all problems in a domain, an alternative approach is to select each abstraction hierarchy based on a problem or class of problems to be solved. As described in the following sections, the ordered monotonicity property can be used to produce finer-grained abstraction hierarchies if the property is guaranteed relative to a given problem instead of for an entire domain.

### 4.1.2 Refinement of Abstract Plans

This section defines *establishment*, *justification*, and *refinement*. These definitions are then used to formally define the monotonicity and ordered monotonicity properties.

#### Establishment

Abstract planning is usually done in a top-down manner. An solution is first found in the most abstract version of the problem space, and then it is refined to account for successive levels of detail. This notion is formalized by first defining the concept of "operator establishment." Intuitively, an operator $\alpha$ establishes a precondition of another operator $\beta$ in a plan, if it is the last operator before $\beta$ in the plan that achieves that precondition.

More precisely, an operator $\alpha$ establishes precondition $p$ of operator $\beta$ whenever $\alpha$ precedes $\beta$, $p$ is an effect of $\alpha$ and a precondition of $\beta$, and there are no operators between $\alpha$ and $\beta$ that have $p$ as an effect. This is formalized as follows, where

$\alpha \prec_\Pi \beta$ means that operator $\alpha$ precedes operator $\beta$ in plan $\Pi$, and $Ops(\Pi)$ is the set of instantiated operators in plan $\Pi$. Recall from Section 2.1 that $P_\beta$ refers to the preconditions of operator $\beta$ and $E_\alpha$ refers to the effects of operator $\alpha$.

**Definition 4.1 (Establishment)** *Let $\Pi$ be a correct plan, $\alpha, \beta \in \text{Ops}(\Pi)$, and $p \in E_\alpha, P_\beta$. Then $\alpha$ establishes $p$ for $\beta$ in $\Pi$ (*$\text{Establishes}(\alpha, \beta, p, \Pi)$*) if and only if*

1. $\alpha \prec_\Pi \beta$,
2. $\forall \alpha' \in \text{Ops}(\Pi)$, *if* $\alpha \prec_\Pi \alpha' \prec_\Pi \beta$, *then* $p \notin E_{\alpha'}$.

The first condition states that $\alpha$ must precede $\beta$ in the plan. The second condition states that $\alpha$ must be the last operator that precedes $\beta$ and adds precondition $p$. Since $\Pi$ is a correct plan, this implies that there is additionally no operator between $\alpha$ and $\beta$ that undoes $p$.

### Justification

An operator in a plan is justified with respect to a goal if it contributes, directly or indirectly, to the satisfaction of that goal. This condition holds when an operator achieves a literal that is either a goal or a precondition of a subsequent justified operator. Justification is used in the definition of a refinement below.

**Definition 4.2 (Justification)** *Let $\Pi$ be a correct plan, $\alpha \in \text{Ops}(\Pi)$, and $S_g$ a goal. $\alpha$ is justified with respect to $S_g$ in $\Pi$ (*$\text{Justified}(\alpha, \Pi, S_g)$*) if and only if there exists $u \in E_\alpha$ such that either:*

1. $u \in S_g$, *and* $\forall \alpha' \in \text{Ops}(\Pi)$, *if* $(\alpha \prec_\Pi \alpha')$ *then* $u \notin E_{\alpha'}$, *or*
2. $\exists \beta \in \text{Ops}(\Pi)$ *such that $\beta$ is justified with respect to $S_g$, and* $\text{Establishes}(\alpha, \beta, u, \Pi)$.

The justification definition can be extended to plans as follows: $Justified(\Pi, S_g)$ if and only if for every operator $\alpha \in Ops(\Pi)$, $Justified(\alpha, \Pi, S_g)$.

Any operator that is not justified is not needed to achieve the goal and can be removed. Thus, an unjustified plan $\Pi$ (one for which *Justified* is false) that achieves $S_g$ can be justified by removing all unjustified operators. $JustifyPlan(\Pi, S_g)$ is used to denote the justified version of $\Pi$. Under the above definitions, for any correct plan $\Pi$ that achieves goal $S_g$, $Justified(JustifyPlan(\Pi, S_g), S_g)$ holds.

Recall from Section 3.1.2, $\mathcal{M}_s^i(s)$ is a state mapping function that maps a ground-level state $s$ to a state at level $i$, and $\mathcal{M}_o^i(\alpha)$ is an operator mapping function that maps a ground-level operator $\alpha$ to an operator at level $i$. Both of these functions

perform the mapping simply by dropping the conditions that are not in abstraction level $i$. Similarly, we can define a *plan mapping* function $\mathcal{M}_p^i(\Pi)$ that maps a ground-level plan $\Pi$ to a plan at level $i$ by replacing each operator $\alpha$ in $\Pi$ by $\mathcal{M}_o^i(\alpha)$. We can also define a *problem mapping* function $\mathcal{M}_\rho^i(\rho)$ that maps a problem $\rho = (S_0, S_g)$ to the corresponding abstract problem $\rho = (\mathcal{M}_s^i(S_0), \mathcal{M}_s^i(S_g))$ at level $i$. In the remainder of this section, the subscript will be dropped from the mapping functions since it is clear from the context which mapping function is required.

By the above definitions, $\mathit{JustifyPlan}(\mathcal{M}^i(\Pi), \mathcal{M}^i(S_g))$ denotes the abstract plan that corresponds to the ground-level plan $\Pi$ justified at level $i$ with respect to goal $S_g$.

The definition of justification can now be used to show that for any plan that achieves a goal in the base space, the abstract version of that plan with all the unjustified operators removed achieves the goal at the abstract level.

**Lemma 4.1** *If* $\Pi$ *is a plan that solves* $\rho = (S_0, S_g)$ *at the base level of an abstraction hierarchy, then* JustifyPlan$(\mathcal{M}^i(\Pi), \mathcal{M}^i(S_g))$ *is a plan that solves* $\mathcal{M}^i(\rho)$ *on level* $i$ *of the hierarchy.*

The proof of an analogous lemma can be found in [Tenenberg, 1988, pg.69]. The idea is that since conditions involving certain literals are eliminated in ascending the abstraction hierarchy, one can eliminate from plans those operators included solely to satisfy these eliminated conditions. For example, if the `OpenDoor` condition is eliminated at level $i$, then those plan steps from levels below $i$ that achieve `OpenDoor` can be removed. Note that the upward solution property holds as a trivial corollary of this lemma.

### Refinement

With the notion of justification, we can now define the "refinement" of an abstract plan. Intuitively, a plan $\Pi$ is a refinement of an abstract plan $\Pi^{\mathcal{A}}$, if all operators and their ordering relations in $\Pi^{\mathcal{A}}$ are preserved in $\Pi$, and the new operators have been inserted for the purpose of satisfying the re-introduced preconditions.

**Definition 4.3 (Refinement)** *A plan* $\Pi$ *at level* $i-1$ *is a* refinement *of an abstract plan* $\Pi^{\mathcal{A}}$ *at level* $i$*, if*

1. $\Pi$ *is justified at level* $i-1$*, and*
2. *there is a 1-1 function* $c$ *(a* correspondence function*) mapping each operator of* $\Pi^{\mathcal{A}}$ *into* $\Pi$*, such that*
   (a) $\forall \alpha \in \mathrm{Ops}(\Pi^{\mathcal{A}}), \mathcal{M}^i(c(\alpha)) = \alpha$,

*(b)* *if* $\alpha \prec_{\Pi^{\mathcal{A}}} \beta$, *then* $c(\alpha) \prec_{\Pi} c(\beta)$,

*(c)* $\forall \gamma \in \text{Ops}(\Pi), \forall \alpha \in \text{Ops}(\Pi^{\mathcal{A}})$, *if* $c(\alpha) \neq \gamma$, *then* $\exists \beta \in \text{Ops}(\Pi)$ *with precondition* $p$ *such that* $\text{Justified}(\gamma, \Pi, p)$ *and* $\text{Level}(p) = i - 1$.

If $\Pi$ is a refinement of $\Pi^{\mathcal{A}}$, then we say that $\Pi^{\mathcal{A}}$ refines to $\Pi$. This formal definition captures the notion of plan refinement used in many different planners, including ABSTRIPS [Sacerdoti, 1974], NOAH [Sacerdoti, 1977], SIPE [Wilkins, 1984], and ABTWEAK [Yang and Tenenberg, 1990].

### 4.1.3 Monotonic Abstraction Hierarchies

In Lemma 4.1, the relationship between $\Pi$ and its justifications at successive levels of abstraction reveals that not only are operators being eliminated from a plan in ascending the abstraction hierarchy, but that for those preconditions still present at a given level, the establishment relationships from the higher levels are preserved. Thus, if a planner can find this abstract-level plan, this plan could be expanded at successively lower levels by inserting operators that do not violate the abstract establishment structure. This section first defines a monotonic refinement and then uses this definition to define a monotonic abstraction hierarchy.

A monotonic refinement of an abstract plan is a refinement that preserves all of the establishment relations.

**Definition 4.4 (Monotonic Refinement)** *Let* $\Pi^{\mathcal{A}}$ *be an abstract plan that solves* $\mathcal{M}^i(\rho)$ *at level* $i$, $i > 0$ *and is justified relative to* $\mathcal{M}^i(S_g)$. *A level* $i-1$ *plan* $\Pi$ *is a* monotonic refinement *of a level* $i$ *plan* $\Pi^{\mathcal{A}}$ *if and only if*

1. $\Pi$ *is a refinement of* $\Pi^{\mathcal{A}}$,
2. $\Pi$ *solves* $\mathcal{M}^{i-1}(\rho)$ *at level* $i-1$, *and*
3. $\text{JustifyPlan}(\mathcal{M}^i(\Pi), \mathcal{M}^i(S_g)) = \Pi^{\mathcal{A}}$.

An abstraction hierarchy is monotonic if every solvable problem has an abstract solution that has a monotonic refinement at each lower level.

**Definition 4.5 (Monotonic Abstraction Hierarchy)** *A* $k$*-level abstraction hierarchy is monotonic, if and only if, for every problem* $\rho = (S_0, S_g)$ *solvable at the ground level, there exists a sequence of plans* $\Pi^{k-1}, \ldots, \Pi^0$ *such that* $\Pi^{k-1}$ *is a justified plan for solving* $\mathcal{M}^{k-1}(\rho)$ *at level* $k-1$, *and for* $0 < i < k$, $\Pi^{i-1}$ *is a monotonic refinement of* $\Pi^i$.

An important feature of the monotonicity property is its generality:

**Theorem 4.1** *Every abstraction hierarchy is monotonic.*

**Proof**: This will be proven for a two-level hierarchy, but can be easily extended to $k$ levels by induction. Let $\rho$ be a problem at the ground level, and let $\Pi$ be a ground-level plan that solves $\rho$. By Lemma 4.1, there exists a abstract plan $\Pi^{\mathcal{A}} = \mathit{JustifyPlan}(\mathcal{M}^1(\Pi), \mathcal{M}^1(S_g))$ that solves the abstract problem $\mathcal{M}^1(\rho)$. By definition, $\Pi^{\mathcal{A}}$ is justified. It follows from Definition 4.4 that $\Pi$ is a monotonic refinement of $\Pi^{\mathcal{A}}$. Thus, for every problem $\rho$ in the ground space, there exists an abstract plan that solves $\rho$ in the abstract space and has a monotonic refinement. □

This property is useful because it means that the completeness of a hierarchical problem solver can be maintained while only considering the monotonic refinements of an abstract plan.

### 4.1.4 Ordered Monotonic Abstraction Hierarchies

This section first defines a more restrictive refinement, called an *ordered refinement*, and then uses this definition to define an *ordered monotonic abstraction hierarchy.*

An ordered refinement of an abstract plan $\Pi^{\mathcal{A}}$ is a refinement $\Pi$ in which no literals in the abstract level are changed by the operators inserted to refine the abstract plan.

**Definition 4.6 (Ordered Refinement)** *Let $\Pi^{\mathcal{A}}$ be a justified plan that solves $\mathcal{M}^i(\rho)$ at level $i$, $i > 0$. A level $i-1$ plan $\Pi$ is an* ordered refinement *of a level $i$ plan $\Pi^{\mathcal{A}}$ if and only if*

1. *$\Pi$ is a monotonic refinement of $\Pi^{\mathcal{A}}$, and*
2. *$\forall \alpha \in \mathrm{Ops}(\Pi)$, if $\alpha$ adds or deletes a literal $l$ with $\mathrm{Level}(l) \geq i$, then $\exists \alpha' \in \mathrm{Ops}(\Pi^{\mathcal{A}})$ such that $\alpha = c(\alpha')$.*

The first condition requires that $\Pi^{\mathcal{A}}$ is a monotonic refinement of $\Pi$. The second condition above states that in plan $\Pi$, the only operators that add or delete literals at level $i$ or above are refinements of the operators in $\Pi^{\mathcal{A}}$.

The definition of an ordered refinement is now used to define the *ordered monotonicity property.*

**Definition 4.7 (Ordered Monotonic Abstraction Hierarchy)** *An abstraction hierarchy is* ordered monotonic *if and only if, for all problems $\rho$ and for all justified plans $\Pi^{\mathcal{A}}$ that solve $\mathcal{M}^i(\rho)$ at level $i$, for $i > 0$, every refinement of $\Pi^{\mathcal{A}}$ at level $i-1$ is an ordered refinement.*

This property guarantees that every possible refinement of an abstract plan will leave the conditions established in the abstract plan unchanged. In contrast, the monotonicity property requires explicit protection of these conditions. By ensuring that

every refinement is ordered, the ordered monotonicity property guarantees that no violation of the monotonic property will ever occur during plan generation.

Unlike the monotonicity property, not all abstraction hierarchies satisfy the ordered monotonicity property. It is therefore important to explore conditions under which a hierarchy satisfies this property. The following restriction defines a set of constraints that are sufficient but not necessary to guarantee the ordered monotonicity property. The constraints specify a partial ordering of the literals in an abstraction hierarchy.

**Restriction 4.1** *Let $O$ be the set of operators in a domain. $\forall \alpha \in O, \forall p \in P_\alpha$ and $\forall e, e' \in E_\alpha$,*

1. $\text{Level}(e) = \text{Level}(e')$, *and*
2. $\text{Level}(e) \geq \text{Level}(p)$.

The first condition constrains all the literals in the effects of an operator to be at the same abstraction level. The second condition constrains the preconditions of an operator to either be at the same or lower level as the effects. As proved below, these two conditions are sufficient to guarantee the ordered monotonicity property of an abstraction hierarchy.

**Lemma 4.2** *If an abstraction hierarchy satisfies Restriction 4.1, then any justified plan for achieving a literal $l$ does not add or delete any literal whose level is higher than* $\text{Level}(l)$.

**Proof**: Let $\Pi$ be a justified plan at level $i$ that achieves $l$. Since $\Pi$ is justified, every operator in $\Pi$ is used either directly or indirectly to achieve $l$. Thus, the establishment relations in $\Pi$ form a directed, acyclic *proof graph* in which $l$ is the root. The operators form the nodes and the establishment relations form the arcs of the graph. The depth of a node in the proof graph is the minimal number of arcs to the root $l$. Below, we prove by induction on the depth of the proof graph that $\forall \alpha \in \mathit{Ops}(\Pi), e \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e)$. This condition will guarantee that no operator in $\Pi$ affects any literal higher than $\mathit{Level}(l)$ in the hierarchy.

For the base case, consider the operator $\alpha$ at depth 1. Since $\Pi$ achieves $l$ and $\mathit{Justified}(\Pi, l)$, then $l \in E_\alpha$. From Restriction 4.1, $\forall e \in E_\alpha$, $\mathit{Level}(l) = \mathit{Level}(e)$.

For the inductive case, assume that for each operator $\beta$ at depth $i$, $\forall e \in E_\beta$, $\mathit{Level}(l) \geq \mathit{Level}(e)$. Let $\alpha$ be an operator at depth $i+1$. Since $\Pi$ is justified, there exists an operator $\beta$ at depth $i$ with $p \in P_\beta$, such that $p \in E_\alpha$. From Restriction 4.1, $\forall e \in E_\beta$, $\mathit{Level}(e) \geq \mathit{Level}(p)$. From the inductive hypothesis, $\mathit{Level}(l) \geq \mathit{Level}(e)$. Therefore, $\mathit{Level}(l) \geq \mathit{Level}(p)$. From Restriction 4.1, $\forall e' \in E_\alpha$, $\mathit{Level}(p) = \mathit{Level}(e')$. Thus, $\forall e' \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e')$. □

**Theorem 4.2** *Every abstraction hierarchy satisfying Restriction 4.1 is an ordered monotonic hierarchy.*

**Proof**: From Definition 4.7 we need to show that every refinement of a justified plan $\Pi^{\mathcal{A}}$ is an ordered refinement. By way of contradiction, assume that there exists a plan $\Pi$ that is a refinement of $\Pi^{\mathcal{A}}$ at level $i-1$, but is not an ordered refinement. It follows from Definition 4.6 that an operator $\alpha$ in $\Pi$ changes a literal $l$, with $Level(l) \geq i$, but the corresponding abstract operator $\mathcal{M}^i(\alpha)$ is not in $\Pi^{\mathcal{A}}$. Since $\Pi$ is a refinement, it follows from Definition 4.3 that $\Pi$ is justified. Since $\Pi$ is justified and $\alpha \in Ops(\Pi)$, $\alpha$ must achieve some condition $p$ and be justified with respect to that condition. In addition, since $\mathcal{M}^i(\alpha)$ is not in $\Pi^{\mathcal{A}}$, it follows from Definition 4.3 that $Level(p) = i-1$. But $\alpha$ also achieves $l$, where $Level(l) \geq i$, which contradicts lemma 4.2. □

In general, the ordered monotonicity property is quite restrictive since it requires that the property hold for every problem in the domain. A natural extension, which allows finer-grained abstraction hierarchies, is to only require that an abstraction hierarchy have the ordered monotonicity property relative to a given problem. This extension is straightforward and is based on the definitions and results in the previous section. Associated with this property is a restriction on the assignment of literals to levels that is sufficient to insure this property for a given problem instance.

**Definition 4.8 (Problem-Specific Ordered Monotonic Hierarchy)** *An abstraction hierarchy is* ordered monotonic *relative to a specific problem $\rho$, if and only if for all justified plans $\Pi^{\mathcal{A}}$ that solve $\mathcal{M}^i(\rho)$ at level $i$, for $i > 0$, every refinement of $\Pi^{\mathcal{A}}$ at level $i-1$ is an ordered refinement.*

A problem-specific, ordered monotonic hierarchy can be formed by considering which operators of a domain could be used to solve a given goal. In particular, only some of the operators would actually be relevant to achieving a given goal. And, of those operators, only some of their effects would be relevant to achieving the goal. These are called the "relevant effects". The relevant effects of an operator $\alpha$ relative to a goal $S_g$ (denoted $Relevant(\alpha, S_g)$) are those effects of $\alpha$ that are either in $S_g$, or are preconditions of operators that have relevant effects with respect to $S_g$.

**Definition 4.9 (Relevant Effects)** *Let $S_g$ be a goal state, and $O$ be the set of operators in a domain. Given $\alpha \in O$, $e \in E_\alpha$, $e$ is a* relevant effect *of $\alpha$ with respect to $S_g$ (or $e \in$ Relevant$(\alpha, S_g)$) if and only if*

1. *$e \in S_g$, or*
2. *$\exists \beta \in O$,* Relevant$(\beta, S_g) \neq \emptyset$ *and $e \in P_\beta$.*

The following restriction defines a set of constraints on an abstraction hierarchy that are sufficient to guarantee the ordered monotonicity property of an abstraction hierarchy for a specific problem.

**Restriction 4.2** *Let $\rho = (S_0, S_g)$ be a problem instance and $O$ be the set of operators. $\forall \alpha \in O$, $\forall e, e' \in E_\alpha$, $p \in P_\alpha$, if $e \in$ Relevant$(\alpha, S_g)$ then*

1. Level$(e) \geq$ Level$(e')$,
2. Level$(e) \geq$ Level$(p)$.

The restriction requires that all the relevant effects of an operator $\alpha$ to be at the same or higher levels of abstraction than both the other effects and the preconditions of $\alpha$.

**Lemma 4.3** *If an abstraction hierarchy satisfies Restriction 4.2, then any justified plan for achieving a literal $l$ does not add or delete any literal whose level is higher than* Level$(l)$.

**Proof**: The proof is analogous to the proof of Lemma 4.2. As above, let $\Pi$ be a justified plan at level $i$ that achieves $l$, where the establishment relations in $\Pi$ form a directed, acyclic *proof graph* in which $l$ is the root. The proof is by induction on the depth of the proof graph and shows that $\forall \alpha \in Ops(\Pi), e \in E_\alpha$, $Level(l) \geq Level(e)$.

For the base case, consider the operator $\alpha$ at depth 1. Since $\Pi$ achieves $l$ and $Justified(\Pi, l)$, then $l \in E_\alpha$. From Restriction 4.2, since $l \in Relevant(\alpha, l)$, $\forall e \in E_\alpha$, $Level(l) \geq Level(e)$.

For the inductive case, assume that for each operator $\beta$ at depth $i$, $\forall e \in E_\beta$, $Level(l) \geq Level(e)$. Let $\alpha$ be an operator at depth $i+1$. Since $\Pi$ is justified, there exists an operator $\beta$ at depth $i$ with precondition $p \in P_\beta$, such that $p \in E_\alpha$. From Restriction 4.2, $\forall q \in Relevant(\beta, S_g)$, $Level(q) \geq Level(p)$. From the inductive hypothesis, $Level(l) \geq Level(q)$. Therefore, $Level(l) \geq Level(p)$. Since $p \in Relevant(\alpha, l)$, from Restriction 4.2, $\forall e' \in E_\alpha$, $Level(p) \geq Level(e')$. Thus, $\forall e' \in E_\alpha$, $Level(l) \geq Level(e')$. □

**Theorem 4.3** *Every abstraction hierarchy satisfying Restriction 4.2 with respect to a problem $\rho$ is a problem-specific ordered monotonic hierarchy with respect to $\rho$.*

**Proof**: The proof is the same as the proof of Theorem 4.7 with Lemma 4.2 replaced by Lemma 4.3. □

## 4.2 Generating Abstraction Hierarchies

The restrictions described in the last section can be used as the basis for constructing ordered monotonic abstraction hierarchies. Hierarchies that have this property are desirable because they partition the literals in a domain such that a condition at one level in the hierarchy can be achieved without interacting with conditions higher in the hierarchy. The construction of such a hierarchy requires finding a sufficient set of constraints on the placement of the literals in a hierarchy such that this property can be guaranteed. This section first presents algorithms for finding both problem-independent and problem-specific constraints that are sufficient to guarantee the ordered monotonicity property. Then it describes the top-level algorithm for constructing an abstraction hierarchy given a set of constraints. To simplify the description of the algorithms, this section assumes that the operators are fully-instantiated. Section 4.4.2 describes how the algorithms handle operators with variables.

### 4.2.1 Determining the Constraints on a Hierarchy

This section presents two algorithms for generating ordering constraints on an abstraction hierarchy. The first algorithm produces a set of problem-independent constraints that guarantee the ordered monotonicity property. The second algorithm produces a set of problem-specific constraints, where the constraints are sufficient to guarantee the ordered monotonicity property for a given problem. The ordering constraints generated by the algorithms are placed in a directed graph, where the literals form the nodes and the constraints form the edges. Each literal at a node represents both that literal and the negation of the literal since it is not possible to change one without changing the other. A directed edge between two nodes in the graph indicates that the literals of the first node cannot occur lower in the abstraction hierarchy than the literals of the second node.

#### Problem-Independent Constraints

A set of problem-independent constraints can be generated for a problem space based on Restriction 4.1. This restriction requires that all the effects of each operator must be placed in the same abstraction level and the preconditions of each operator cannot be placed in a higher level in the abstraction hierarchy than the effects of the same operator. The algorithm in Table 4.1 finds exactly this set of constraints and records them in a directed graph. For each operator, the algorithm arbitrary selects an effect and then adds directed edges in both directions between that effect and all the other effects. It also adds directed edges between the selected effect and all of the preconditions of the operator.

**Input**: The operators that define the problem space.
**Output**: Sufficient constraints to guarantee ordered monotonicity.

```
function Find_Constraints(graph,operators):
     for each op in operators
          select lit1 in Effects(op)
               begin
               for each lit2 in Effects(op)
                    begin
                    Add_Directed_Edge(lit1,lit2,graph);
                    Add_Directed_Edge(lit2,lit1,graph)
                    end;
               for each lit2 in Preconditions(op)
                    Add_Directed_Edge(lit1,lit2,graph)
               end;
     return(graph);
```

Table 4.1: Problem-Independent Algorithm for Determining Constraints

The complexity of this algorithm is $O(d)$, where $d$ is the length of the encoding of a problem space (i.e., the number of literals in the preconditions and effects of all the operators). To find the constraints, the algorithm only scans through the preconditions and effects of each operator once.[2]

While this algorithm generates a sufficient set of constraints for the ordered monotonicity property, many of the constraints will not be necessary to guarantee the property. As such, the algorithm will only produce abstractions for a limited class of problem spaces. The next section describes a problem-specific version of this algorithm, which will produce useful abstractions for a wider class of problem spaces.

### Problem-Specific Constraints

Restriction 4.2 can be used to generate a set of problem-specific constraints for a given problem space and problem. This restriction provides a sufficient set of constraints to guarantee the ordered monotonicity property for a given problem. An algorithm that implements this restriction is shown in Table 4.2. The algorithm is similar to the problem-independent one, but forms the constraints based on a particular set of goals to solve.

[2]Thanks to Charles Elkan for pointing out that my original $O(d^2)$ algorithm [Knoblock, 1990a] could be transformed into a linear algorithm.

The algorithm is given the operators and the goals of the problem to be solved and it returns a directed graph of the constraints on the abstraction hierarchy. It scans through each of the goal literals and first checks to see if the constraints for the given literal have already been added to the graph (lines 1-2). If not, it scans through each of the operators and finds those operators that could be used to achieve the given goal (lines 3-4). The algorithm then adds constraints between any effect that matches the goal and the other effects and preconditions of the operator (lines 5-9). The algorithm is called recursively on the preconditions of the operator since these could arise as subgoals during problem solving (line 10). The algorithm will terminate once it has considered all of the conditions that could arise as goals or subgoals during problem solving.

---

**Input**: The operators of the problem space and the goals of a problem.
**Output**: Sufficient constraints to guarantee ordered monotonicity for the given problem.

```
function Find_Constraints(graph,operators,goals):
1.   for each goal in goals do
2.        if Not(Constraints_Determined(goal,graph)) then
3.             for each op in operators do
4.                  if goal in Effects(op) do
                         begin
5.                       for each effect in Effects(op) do
6.                            Add_Directed_Edge(goal,effect,graph);
7.                       preconds ← Preconditions(op);
8.                       for each precond in preconds do
9.                            Add_Directed_Edge(goal,precond,graph);
10.                      Constraints_Determined(goal,graph) ← true;
11.                      graph ← Find_Constraints(graph,operators,preconds)
                         end;
11.  return(graph)
```

---

Table 4.2: Problem-Specific Algorithm for Determining Constraints

An important advantage of the problem-specific abstractions is that the algorithm only produces the constraints that are relevant to the particular problem to be solved. Thus, it can produce finer-grained hierarchies than could be produced for the entire problem domain. In many cases the abstraction hierarchy produced by the problem-independent algorithm collapses into a single level, while the problem-specific algorithm produces a useful abstraction hierarchy.

The complexity of determining the constraints, and thus the complexity of creating the problem-specific abstraction hierarchies, is $O(n \cdot o \cdot l)$, where $n$ is the number

of different literals in the graph, $o$ is the maximum number of operators relevant to achieving any given literal, and $l$ is the maximum length (total number of preconditions and effects) of the relevant operators. In the worst case, the algorithm must loop through each literal, and for each relevant operator scan through the body of the operator and add the appropriate constraints. This cost is insignificant compared to problem solving since its complexity is polynomial in the size of the problem space, while the complexity of problem solving is exponential in the solution length.

### 4.2.2 Constructing A Hierarchy

This section describes the algorithm for constructing an abstraction hierarchy. The algorithm is given the operators that define a problem space and, optionally, the goals of a problem to be solved, and it produces an ordered monotonic abstraction hierarchy. The algorithm partitions the literals of a domain into classes and orders them such that the literals at one level will not interact with the literals in a more abstract level. The final hierarchy consists of an ordered set of abstraction spaces, where the highest level in the hierarchy is the most abstract and the lowest level is the most detailed.

Table 4.3 defines the `Create_Hierarchy` procedure for building ordered monotonic abstraction hierarchies. The procedure is given the domain operators and, depending on the definition of `Find_Constraints`, may also be given the goals of the problem to be solved. Without using the goals, `Create_Hierarchy` produces a problem-independent abstraction hierarchy, which can be used for solving any problem in a domain. Using the goals, the algorithm produces an abstraction hierarchy that is tailored to the particular problem to be solved.

---

**Input**: Operators of a problem space and, optionally, the goals of a problem.
**Output**: An ordered monotonic abstraction hierarchy.

```
procedure Create_Hierarchy(operators[,goals]):
1.   graph ← Find_Constraints({},operators[,goals]);
2.   components ← Find_Strongly_Connected_Components(graph);
3.   partial_order ← Construct_Reduced_Graph(graph,components);
4.   total_order ← Topological_Sort(partial_order);
5.   abs_hierarchy ← Construct_Abs_Hierarchy(total_order);
6.   return(abs_hierarchy)
```

---

Table 4.3: Algorithm for Creating an Abstraction Hierarchy

- Step 1 of the algorithm produces a set of constraints on the order of the literals in an abstraction hierarchy using the algorithms in either Table 4.1 or Table 4.2. By Theorems 4.2 and 4.3, the constraints are sufficient to guarantee that a hierarchy built from these constraints will have the ordered monotonicity property.

- Step 2 finds the strongly connected components of the graph using a depth-first search [Aho *et al.*, 1974]. Two nodes in a directed graph are in the same strongly connected component if there is a path from one node to the other and back again. Thus, any node in a strongly connected component can be reached from any other node within the same component. As such, this step partitions the graph into classes of literals where all the literals in a class must be placed in the same abstraction level.

- Step 3 constructs a reduced graph where the nodes that comprise a connected component in the original graph correspond to a single node in the reduced graph. There is a constraint between two nodes in the reduced graph if there was a constraint between the corresponding nodes in the original graph. The literals within a node in the reduced graph must be placed in the same abstraction space and the constraints between nodes define a partial order of the possible abstraction hierarchies.

- Step 4 transforms the partial order into a total order using a topological sort [Aho *et al.*, 1983]. The total order defines a single ordered monotonic abstraction hierarchy. There may be a number of possible total orders for a given partial order and one order may be better than another. Section 4.4.4 describes the set of heuristics used to choose between the possible total orders.

- Step 5 uses the total order to construct an abstraction hierarchy. The most abstract level in the hierarchy contains only the literals that are first in the total order. Each successive level of the abstraction hierarchy contains the literals from the previous level combined with the next element of the total order. The most detailed level in the hierarchy contains all of the literals in the graph.

The complexity of steps 2-5 in the algorithm above is linear in the size of the graph. The complexity of both finding the strongly connected components of a directed graph and performing the topological sort is $O(max(e, v))$ [Aho *et al.*, 1974], where $e$ is the number of edges (constraints) and $v$ is the number of vertices (literals). Creating the reduced graph is also $O(max(e, v))$ since the new graph can be created by scanning through each of the vertices and edges once. The complexity of forming the abstraction hierarchy is $O(v)$ since this step only needs to scan the vertices of

the sorted graph once to build the hierarchy. Thus, the complexity of steps 2-5 is $O(max(e, v))$.

Using the problem-independent algorithm for finding the constraints, the complexity of building an abstraction hierarchy is linear in the length of the encoding. Since finding the constraints is $O(d)$, where $d$ is the length of the encoding, and the number of possible constraints, $e$, and the number of possible literals, $v$, is bounded by $O(d)$, the complexity of the entire algorithm is $O(d)$.

As described above, the complexity of the problem-specific algorithm for finding the constraints is $O(n \cdot o \cdot l)$, so the complexity of building a problem-specific abstraction hierarchy is also $O(n \cdot o \cdot l)$ ($n$ is the number of different literals, $o$ is the number of operators relevant to achieving each literal, and $l$ is the length of each relevant operator). The complexity of the graph algorithms is bounded by the complexity of finding the constraints since the number of vertices, $v$ is the number of literals $n$, and the number of edges $e$ must be bounded by $n \cdot o \cdot l$ since this is the complexity of the algorithm for finding the constraints, which are the edges in the graph.

## 4.3 Tower of Hanoi Example

This section applies the approach to generating abstractions described in this chapter to the Tower of Hanoi puzzle. The algorithms presented in the previous section can be used to automatically generate the abstraction of the Tower of Hanoi described in the previous chapter, where the disks are partitioned into separate abstraction levels. This section describes how the algorithm generates this abstraction, shows the intermediate results at each step in the algorithm, and then provides an intuitive explanation for why the ordered monotonicity property provides a good decomposition of this problem.

Given a three-disk Tower of Hanoi problem in either representation described in Section 2.2, both the problem-independent and problem-specific versions of the algorithm generate a three-level abstraction hierarchy. The two algorithms differ in that for a problem involving only the two smallest disks, the problem-specific algorithm would generate only a two-level hierarchy, while the problem-independent version would still generate a three-level hierarchy since it does not take the problem into account.

The first step of the algorithm for constructing an abstraction hierarchy is to find a set of constraints that are sufficient to guarantee the ordered monotonicity property. Both versions of the `find-constraints` algorithm would produce the directed graph of constraints shown in Figure 4.1, where `diskC` is the largest disk and `diskA` is the smallest. The problem-independent algorithm would consider each operator and first add constraints that force all the effects of each operator to be in the same abstraction

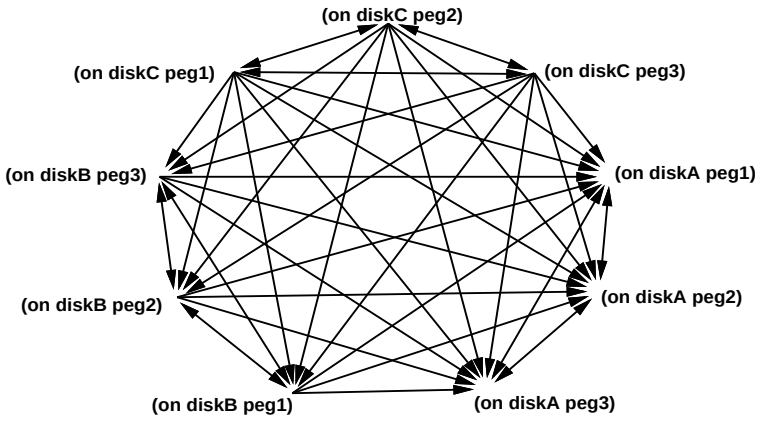


Figure 4.1: Constraints on the Literals in the Tower of Hanoi

level and then add constraints that force the precondition of an operator to be lower (or at the same level) than the effects.

For example, consider the constraints generated by the algorithm from the operator shown in Table 4.4 (additional constraints would be generated from the other operators). First, it would add constraints based on the effects, which would generate a constraint between `(on diskC peg1)` and `(on diskC peg3)`, as well as a constraint between the same literals in the opposite direction. Then the algorithm would consider the preconditions and add constraints between one of the effects and each of the preconditions of that operator. For example, it would add a constraint that required `(on diskC peg3)` to be higher or at the same level as `(on diskB peg1)`. (Note that a literal and a negation of a literal are considered the same literal for purposes of abstraction and thus placed at the same level since obviously one cannot be changed without changing the other.)

```
(Move_DiskC_From_Peg1_to_Peg3
  (preconds (and (on diskC peg1)
                 (not (on diskB peg1))
                 (not (on diskA peg1))
                 (not (on diskB peg3))
                 (not (on diskA peg3))))
  (effects ((del (on diskC peg1))
            (add (on diskC peg3)))))
```

Table 4.4: Instantiated Operator in the Tower of Hanoi

The second step in creating the abstraction hierarchy is to find the strongly connected components. Two literals are in the same connected component if and only if there is a cycle in the directed graph that contains both literals. Figure 4.2 shows the three connected components in the graph, where the literals involving each disk form a component. Each of these components contains a set of literals that must be placed in the same abstraction level.

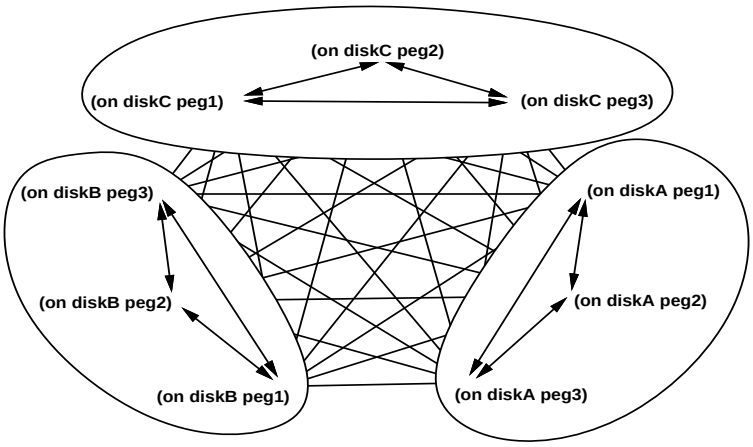


Figure 4.2: Connected Components in the Tower of Hanoi

The third step in the algorithm is to combine the literals within each connected component into a single node to form a reduced graph. The reduced graph for the Tower of Hanoi, which is shown in Figure 4.3, reduces the original graph to a graph with three nodes and only a few constraints between the nodes. The arrows between the nodes in a reduced graph specify the constraints on the order in which the literal classes can be removed to form an abstraction hierarchy.

The fourth step in the algorithm converts the partially-ordered directed graph into a total order using a topological sort. In the case of the Tower of Hanoi there is only one possible order, where the disks are ordered by size. The resulting total order is shown in Figure 4.4

In the last step, the total order is converted into the abstraction hierarchy shown previously in Figure 3.7. For the three-disk puzzle, the highest abstraction level includes literals involving only the largest disk, the next level includes both the largest and middle size disk, and the third level includes all three disks. It is possible to divide the disks into separate abstraction levels since the steps necessary to move a given disk can always be inserted into an abstract plan without interfering with the position of any larger disks. For an n-disk problem, the algorithm would produce a n-level abstraction hierarchy. Section 5.1 presents empirical results on using these abstraction hierarchies for problem solving.

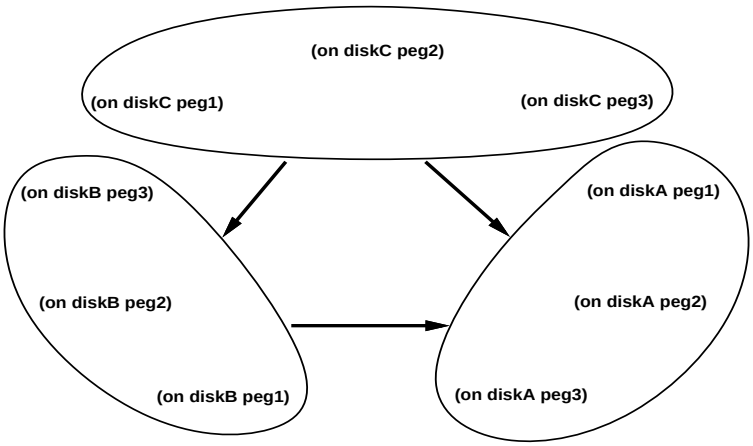


Figure 4.3: Reduced Graph in the Tower of Hanoi

In general, abstraction hierarchies are useful because they form reduced state spaces that can be searched more effectively than the original state space. The reduced state spaces for the three-disk puzzle are shown in Figure 4.5. The state space on the left shows the result of removing the smallest disk. The nodes that differ only by the location of the smallest disk are collapsed into a single node, reducing the state space from 27 states to 9 states. The operators that are not relevant to a given state

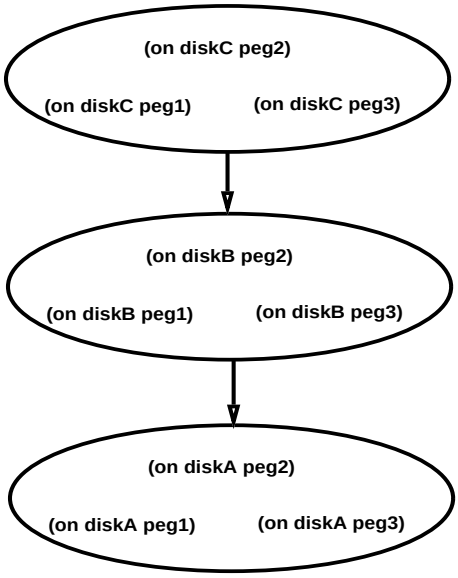


Figure 4.4: Total Order in the Tower of Hanoi

space are replaced with dotted lines. Removing the middle-sized disk, as well as the smallest disk, produces a state space with only 3 states, which is shown on the right of Figure 4.5.

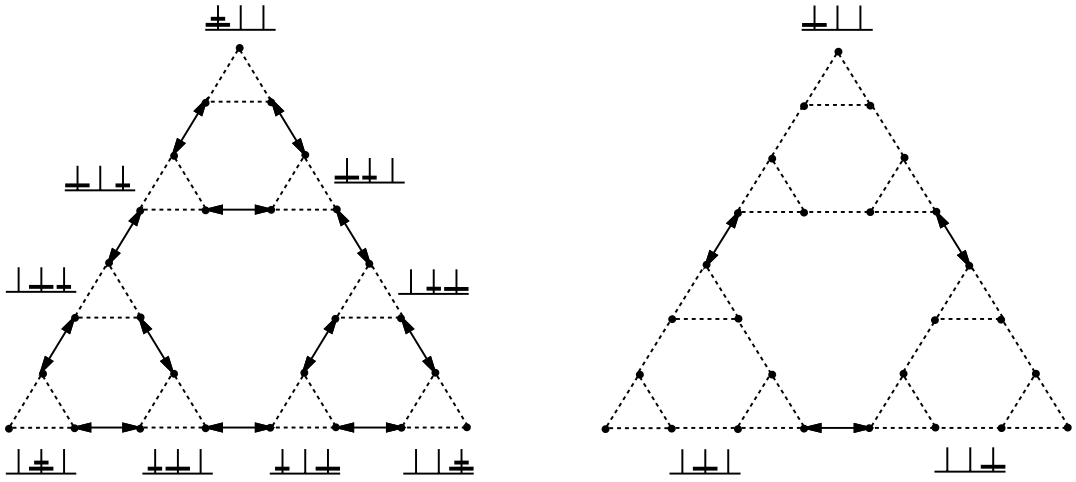

Figure 4.5: Reduced State Spaces in the Tower of Hanoi

The abstraction hierarchy for the Tower of Hanoi has the ordered monotonicity property since it satisfies restriction 4.2. Since this property requires that every refinement of an abstract plan leaves the conditions achieved in the abstract space unchanged, it follows that the work done at each abstraction level is never undone in the process of refining the plan. In the case of the Tower of Hanoi, a solution in the most abstract space produces a plan that moves the largest disk to the goal peg. Since the abstraction hierarchy has the ordered monotonicity property, at the next level only steps for moving the medium disk can be inserted. Thus, the abstraction hierarchy partitions the state space into 3 smaller spaces and any subproblem must be solved within one of these smaller state spaces. At the final level the hierarchy partitions the state space into 9 separate state spaces. Each subproblem at the base level is then solved in one of these 9 spaces.

## 4.4 Generating Abstractions in ALPINE

ALPINE is a fully implemented system that generates abstraction hierarchies for PRODIGY. As shown in Figure 4.6, ALPINE is given a problem space specification and a problem to be solved and it produces a problem-specific abstraction hierarchy for the given problem. The abstraction hierarchy is then used by PRODIGY for hierarchical problem solving.

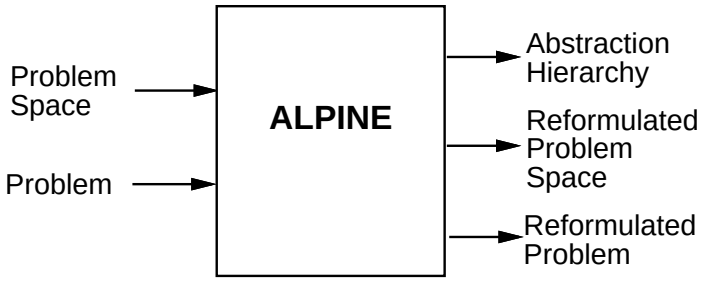


Figure 4.6: The Input and Output of ALPINE

ALPINE uses an extended version of the problem-specific algorithm described in Section 4.2 to produce abstraction hierarchies. Since the abstractions are to be used for the particular hierarchical problem-solving method described in the previous chapter, ALPINE employs several extensions that allow it to produce finer-grained abstraction hierarchies, but still preserve the ordered monotonicity property for the given problem solving method. Using this extended algorithm, ALPINE is able to produce abstraction hierarchies for a variety of domains, including the Tower of Hanoi, the STRIPS robot planning domain, an extended version of the STRIPS domain, and a machine-shop scheduling domain. These results are described in Chapter 5.

To illustrate the ideas in this section, examples from the extended robot planning domain are used. This domain is an augmented version of the original STRIPS robot planning domain [Fikes and Nilsson, 1971]. In the original domain a robot can move among rooms, push boxes around, and open and close doors. In the augmented version, the robot can both push and carry objects and lock and unlock doors. The robot may have to fetch keys as well as move boxes, and may have to contend with doors that cannot be opened.

The description of ALPINE is divided into four sections. The first section describes the problem-space definition that serves as the input to ALPINE. The second section presents the representation language of the abstraction hierarchies. The third section describes the reformulation of the problem and problem space that are performed to produce finer-grained abstraction hierarchies. The last section describes the PRODIGY-specific extensions to the basic algorithm for generating the abstraction hierarchies.

### 4.4.1 Problem Space Definition

ALPINE is given a problem-space specification that consists of three components: a set of PRODIGY operators, a type hierarchy for the operator representation language,

and a set of axioms that state invariants about the states of a problem space.[3].

### Operators

The first component of a problem space is a set of PRODIGY operators. As described in Section 2.3, each operator is composed of a set of preconditions and effects. The preconditions can include conjunctions, disjunctions, negations, and both universal and existential quantifiers. The effects can be conditional, which means that whether or not an effect is realized depends on the state in which the operator is applied. Table 4.5 shows an example operator for pushing a box between rooms in the robot planning domain.

```
(Push_Box_Thru_Dr
  (preconds
      (and (connects door room.x room.y)
           (dr-open door)
           (next-to box door)
           (next-to robot box)
           (pushable box)
           (inroom box room.y)))
  (effects (
      (del (inroom robot room.y))
      (del (inroom box room.y))
      (add (inroom robot room.x))
      (add (inroom box room.x)))))
```

Table 4.5: Example Operator in the Robot Planning Domain

In addition to specifying the preconditions and effects of an operator, some of the effects may be designated as primary. The problem solver is only permitted to use an operator to achieve a goal if the desired effect is listed as a primary effect. By default, every effect of an operator is considered primary. Thus, unless the primary effects are specified, the `push_box_thru_dr` operator (Table 4.5) can be used to move either the robot or a box. In fact, the primary effect for this operator is `(in-room` *box room.x*`)`. Thus, the problem solver would not attempt to use the `push_box_thru_door` operator to move the robot to another room; it would use it only to move the box.

[3]A problem space in PRODIGY can also include a set of control rules, but ALPINE need not consider these to create the abstraction hierarchies. The use of control rules in an abstract space is described in Section 3.5.2

Of course, when a box is moved, the robot would still be moved as a secondary effect. The primary effects are implemented in PRODIGY by generating a set of control rules that select only the operators whose primary effect matches a goal.

### Type Hierarchy

The second component of the problem-space specification is a type hierarchy, which specifies the types of all the constants and variables in a problem domain. The type hierarchy is used to differentiate literals with the same predicate but different argument types. If no type hierarchy is given, then all constants and variables are considered to be of the same type. In the example operator, the type hierarchy allows the system to differentiate between (`inroom robot` *room*) from (`inroom` *box room*). The type hierarchy for the robot planning domain is shown in Figure 4.7. The types, shown in boldface, are on the interior nodes of the tree and the instances are on the leaves. This particular hierarchy was implicit in the original definition of the problem space.

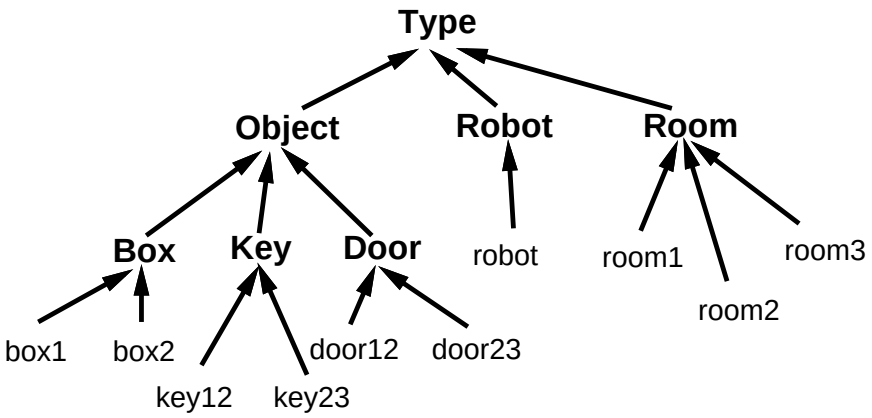


Figure 4.7: Type Hierarchy in the Robot Planning Domain

### Axioms

The third component of the problem-space specification is a set of axioms that describe facts that hold for every state of a problem space. For example, one such axiom for the robot planning domain states that if a door is open then it must be unlocked. A list of example axioms for this domain is shown in Table 4.6. These facts cannot be derived from the operators because they are axioms about conditions that hold in every state. The operators may make assumptions about these conditions, but these assumptions are not usually stated explicitly.

| |
|---|
| (dr-open *door*) → (unlocked *door*) |
| (locked *door*) → (dr-closed *door*) |
| (not (dr-closed *door*)) → (and (dr-open *door*)(unlocked *door*)) |
| (not (dr-open *door*)) → (dr-closed *door*) |
| (not (locked *door*)) → (unlocked *door*) |
| (not (unlocked *door*)) → (and (locked *door*)(dr-closed *door*)) |
| (not (arm-empty)) → (holding *object*) |
| (not (holding *object*)) → (arm-empty) |
| (next-to *box1 box2*) → (and (inroom *box1 room*)(inroom *box2 room*)) |
| (next-to robot *box*) → (and (inroom *box room*)(inroom robot *room*)) |

Table 4.6: Example Axioms in the Robot Planning Domain

### 4.4.2 Representation Language of Abstraction Spaces

The algorithms and examples up to this point have implicitly assumed that the literals in the domain are all represented at the same level of granularity. For example, in the Tower of Hanoi all the literals were completely instantiated ground literals. However, the operators of a domain are usually expressed as operator schemas, where each instantiation of the schema corresponds to an operator. A schema can contain both instantiated and uninstantiated literals. Since the algorithms for generating the abstractions are based on analyzing the potential interactions between the literals used in the operators, the operator representation limits the representation of the abstractions.

To deal with the problem that some literals may be instantiated while others are uninstantiated or partly instantiated, ALPINE associates a type with each literal. It could assume that two literals with the same predicate are of the same type, but this would severely restrict the possible abstractions of a domain. In the Tower of Hanoi, all of the "on" conditions would be forced into the same abstraction level and there would be no abstraction. Instead the type of each literal is determined by both the predicate and the argument types. The type of each literal is easily determined by the type hierarchy described in the last section. Each constant and variable has an associated type, so from each literal, instantiated or uninstantiated, it is possible to determine the argument types. Literals of different types are initially placed in distinct nodes in the graph. For example, in the robot planning domain, (inroom robot *room*) and (inroom *box room*) are of distinct types since they differ by the second argument.

Currently only literal types that are immediately above the leaves of the type hierarchy can be used to represent a literal in an abstraction hierarchy. For example, in

the robot planning domain, the type "object" is a type at an interior node in the tree, so it is not possible to have the literal `(inroom object room)` in the final abstraction hierarchy, only the finer-grained conditions that refer to subtypes of object, `(inroom box room)` and `(inroom key room)`, can be used in the abstraction hierarchy. There is no inherent reason for this restriction except that it simplified the implementation.

### 4.4.3 Problem and Operator Reformulation

The abstraction process described so far involves dropping conditions from a problem space to form a more abstract problem space. The abstractions that are formed by this process will depend heavily on the initial formalization of both the problems and the problem spaces. The original problem space can be reformulated to improve the likelihood of generating useful abstractions.

A straightforward approach to reformulating a problem space is to augment both the goals of a problem and the preconditions of the operators with conditions that will necessarily hold. Since the axioms described above provide invariants that hold for all states in a problem, they can be used to perform this augmentation. Consider a problem that requires achieving some goal $P$. In the problem space that is to be used for solving this goal, imagine there is an axiom which states that $P$ implies $Q$. Using the axiom, the original goal $P$ can be replaced with the goal $P \wedge Q$, since $Q$ will necessarily hold if $P$ holds. At first glance this might appear to make the problem harder. However, by augmenting the goal, it may now be possible to drop $P$ from the problem space. Without the reformulation of the problem, it may not have been possible to drop $P$ and still produce an ordered monotonic abstraction. The augmentation and subsequent abstraction of the problem has the effect of replacing the problem of achieving $P$ with the more abstract problem of achieving $Q$. $P$ will still need to be achieved when the abstract solution is refined, but it may be considerable easier to achieve it once $Q$ has been achieved.

Consider an useful reformulation that occurs in the robot planing domain. The goal is to get `box1` and `box2` next to each other and to place `box1` in `room2`:

`(and (next-to box1 box2)(inroom box1 room2)).`

This problem space has an axiom which states that if two boxes are next to each other then they must be in the same room:

`(next-to` *box1 box2*`)` $\rightarrow$ `(and (inroom` *box1 room*`)(inroom` *box2 room*`)).`

Using this axiom, the original goal would be augmented with the condition that `box2` must also be in `room2`:

`(and (next-to box1 box2)(inroom box1 room2)(inroom box2 room2)).`

The augmentation is important because it allows the system to transform the problem into an abstract problem that would not be possible without the augmentation. In this case, without reformulating the problem ALPINE would find that there is a potential interaction between the next-to condition and the inroom condition and would not put the conditions in separate levels. By augmenting the goal, ALPINE uses the other goal conditions to show that no interaction is possible. (This process is described in the next section.) The reformulation allows the original problem of getting the two boxes next to each other to be replaced by the more abstract problem of getting the two boxes into the same room. Once the abstract problem has been solved, the additional steps for moving the two boxes next to each other can be inserted when the plan is refined.

ALPINE augments the preconditions of operators in exactly the same way. Since the preconditions of operators arise as goals this also allows the system to produce finer-grained abstraction hierarchies that ensure the ordered monotonicity property. The operator `Push_Box_Thru_Dr` would be augmented as shown in Table 4.7. The boxed conditions in the table are the ones added by the axioms. These augmentations allow ALPINE to form an abstraction of this problem space by dropping the `(dr-open` *door*) conditions from the problem space. It makes this abstraction possible since whether the door is open is a detail as long as the door is not locked and the robot is in the appropriate room to open the door.

```
(Push_Box_Thru_Dr
  (preconds
      (and (connects door room.x room.y)
           (dr-open door)
           [(dr-unlocked door)]
           (next-to box door)
           (next-to robot box)
           (pushable box)
           (inroom box room.y)
           [(inroom robot room.y)]))
  (effects (
      (del (inroom robot room))
      (del (inroom box room))
      (add (inroom robot room.x))
      (add (inroom box room.x)))))
```

Table 4.7: Reformulated Operator in the Robot Planning Domain

The reformulations of both the problems and the operators are important for two reasons. First, they allow the system to form abstractions that could not otherwise be guaranteed to have the ordered monotonicity property. Second, they can transform a problem into a problem that can be solved more easily.

### 4.4.4 Abstraction Hierarchy Construction

The algorithm described in Section 4.2 presented a general approach to finding ordered monotonic abstraction hierarchies. This algorithm consists of five steps. First, find the constraints that are sufficient to guarantee the ordered monotonicity property. Second, combine the strongly connected components of the constraint graph to form the abstraction spaces. Third, construct a reduced graph by combining the nodes that comprise the strongly connected components. Fourth, perform the topological sort to order the abstraction spaces. Fifth, convert the total order into an abstraction hierarchy. ALPINE employs this basic algorithm for constructing abstraction hierarchies, but uses refinements of the steps for selecting the constraints and ordering the abstraction spaces to produce better hierarchies.

#### Constraint Generation

The algorithm presented earlier for finding a sufficient set of constraints to guarantee the ordered monotonicity is conservative and will often produce constraints that are unnecessary to guarantee the property. The unnecessary constraints can lead to cycles in the constraint graph, which in turn can collapse the graph and reduce the granularity of the abstraction hierarchies. In addition, the operator language used in PRODIGY is more expressive than the language assumed for the algorithm described earlier. To deal with both of these issues, ALPINE generates abstraction hierarchies that are tailored to PRODIGY and that exploit the particular hierarchical problem solving method used in PRODIGY.

ALPINE's algorithm (Table 4.8) for finding a sufficient set of constraints to guarantee ordered monotonicity exploits two additional sources of knowledge that are specific to the particular problem solving method. First, it uses information about the primary effects of operators to reduce the constraints on the effects. Second, it exploits the fact that although every precondition of an operator can be subgoaled on, in practice there are situations where the preconditions hold and will not become subgoals. Recall that the purpose of these constraints is to guarantee that the literals at one level in a hierarchy will not interact with literals in a more abstract level. The extensions to the basic algorithm, which are described in detail below, preserve the ordered monotonicity property for the given problem solving method, but allow the system to form finer-grained hierarchies than would otherwise be possible.

**Input**: Domain operators and a problem to be solved.
**Output**: Sufficient constraints to guarantee ordered monotonicity for the given problem.

```
function Find_Constraints(graph,operators,goals):
     return Find_Graph_Constraints(graph,operators,goals,goals);

function Find_Graph_Constraints(graph,operators,goals,context):
1.   for each goal in goals do
3.        for each op in operators do
4.             if goal in Primary_Effects(op) do
                    begin
5.                  for each effect in Effects(op) do
6.                       Add_Directed_Edge(goal,effect,graph);
7.                  preconds ← Preconds(op);
8.                  subgoals ← Subgoalable_Preconds(op,goal,preconds,context);
9.                  for each precond in subgoals do
10.                      Add_Directed_Edge(goal,precond,graph);
11.                 Find_Graph_Constraints(graph,operators,subgoals,preconds)
                    end;
12.  return(graph)
```

Table 4.8: Alpine's Algorithm for Determining Constraints

ALPINE avoids unnecessary constraints based on the effects by using the primary effects to determine which operators can actually be used to achieve a goal. Since PRODIGY uses the primary effects to determine which operators can be used to achieve a given goal, it would never consider using an operator that did not have a matching primary effect. Thus, as long as the problem solver only subgoals on operators with matching primary effects, the algorithm will still guarantee the ordered monotonicity property.

ALPINE avoids unnecessary constraints based on the preconditions by determining which preconditions can actually arise as subgoals. In general, if an operator is used to achieve a given goal, it may be necessary to subgoal on any of the preconditions of the operator. However, in some cases the preconditions of an operator are guaranteed to hold and would thus not be subgoaled on. Instead of adding constraints on *all* of the preconditions, the extended algorithm only adds constraints on the preconditions that could actually be subgoaled on in a given context. This extension preserves the ordered monotonicity property since the purpose of these constraints is to avoid subgoaling on conditions that are higher in the abstraction hierarchy.

ALPINE is conservative and assumes everything can be subgoaled on unless it can prove otherwise. There are three ways in which the system can show that a given precondition could not be subgoaled on.

1. If a precondition is static, which means that it cannot be changed by any operators, then it could never be subgoaled on. In the example operator described earlier, the condition `connects` is static since it describes the room connections, which are invariant for a given problem.

2. If the precondition will necessarily hold in the context in which the operator is used, then it could not be subgoaled on. There are two situations in which this can occur:

   (a) If a precondition of an operator is also a precondition of the parent operator, then the precondition must hold at the time the operator is inserted into the plan. For example, a precondition of the `Push_Box_Thru_Dr` operator is that the door is open, and a precondition of the `Open_Door` operator is that the robot is in the room that is next to the door. This second precondition is also a precondition of the `Push_Box_Thru_Dr` operator, so when the `Open_Door` operator is used in the process of pushing a box thru a door, then the system can prove that given this context the `Open_Door` operator will not require getting the robot into the room.

   (b) If a precondition of an operator is the negation of the goal the operator is used to achieve, then it would not be subgoaled on since the negation must already hold or the operator would not have been considered. For example, the `Open_Door` operator has the precondition that the door is closed, however, this condition would not be subgoaled since if this condition is false, (i.e, the door is open) there is no point in considering the operator.

All of this analysis is performed in a preprocessing step that only needs to be done once for a domain. When a hierarchy is created the algorithm calls the function `Subgoalable_Preconds`, which looks up the potential subgoals in a table given a goal and context. This function also keeps track of which conditions in which context have already been considered, so that the algorithm will terminate.

ALPINE handles the full PRODIGY language, but does so simply by assuming the worst case. Disjunctions are treated as conjunctions, conditional effects are treated as unconditional effects, and both universal and existential quantifiers are considered for every possible binding. In addition, ALPINE adds constraints to ensure that if a precondition of an operator generates bindings for a variable that are then used by other preconditions in the same operator, then the precondition literal that generates the bindings cannot occur lower in the hierarchy than the literals that use the bindings.

These assumptions and constraints preserve the ordered monotonicity property and ensure that the abstractions generated by ALPINE will work correctly in PRODIGY.

### Abstraction Hierarchy Selection

Once ALPINE builds the directed graph and combines the strongly connected components, the next step is to convert the partial order of abstraction spaces into a total order. The algorithm shown in Table 4.3 uses a topological sort to produce an abstraction hierarchy. However, in general, the total order produced by the topological sort is not necessarily unique, and two abstraction hierarchies that both have the ordered monotonicity property for a given problem will differ in their effectiveness at reducing search. This section describes the approach ALPINE uses in selecting among the possible ordered monotonic abstraction hierarchies for a problem.

Each potential abstraction space is comprised of a set of literals that have one or more of the following properties:

**Goal Literal** A literal that matches one of the top-level goals.

**Recursive Literal** A literal that could arise as a goal where the plan for achieving that goal could require achieving a subgoal of the same type.

**Static Literal** A literal that is not changed by the effects of any of the operators.

**Binding Literal** A literal that serves as a generator and does not occur in the primary effects of any operators. A generator is any literal that generates bindings for variables in the preconditions of an operator. While a binding literal cannot be subgoaled on, it can generate a set of possible bindings for an operator.

**Plain Literal** A literal that does not have any of the properties above.

The types of the literals that comprise an abstraction space are used to determine the ordering of the levels and which levels should be combined.

ALPINE employs the following set of heuristics to select the final abstraction hierarchy for problem solving:

1. Place the static literals in the most abstract space. By definition there is no operator that adds or deletes any static literal so they can be placed at any level in the hierarchy without risk of a monotonicity violation. If a static literal is false, then it is better to find out as early as possible to avoid wasted work.

2. Place levels involving goal literals as high as possible in the abstraction hierarchy. Thus, whenever there is a choice of placing one set of literals before another in the hierarchy and one set matches a goal literal and the other one does not,

then place the one involving the goal literal above the other. Since goals are sometimes unachievable, it is better to find out as early as possible.

3. Combine levels that involve only plain literals, when the levels could be adjacent in the final hierarchy. Each additional abstraction level in the hierarchy incurs a cost in the refinement process and combining them will reduce this cost. In the domains that have been studied, most of the search occurs in the levels involving the goal and recursive literals.

4. Place levels involving binding literals as low as possible in the abstraction hierarchy and combine these levels with the levels directly below that involve only plain literals. Since the binding literals do not occur in the primary effects of any operators, they cannot be directly achieved. However, they can be used to generate the bindings of variables. The selection of an appropriate set of bindings may require some search, so it is better to delay consideration of these literals as long as possible. In the machine-shop domain, this type of literal is used to perform the actual scheduling.

Figure 4.8 shows how the heuristics would transform an example partial order into a total order. This set of heuristics creates abstraction hierarchies where each separate abstraction level serves some purpose. The goal literals are placed at separate levels because it both orders the top-level goals and partitions the goals of a problem into separate levels. The recursive literals, even if they are not top-level goals, can involve a fair amount of search, and placing them in a separate level can reduce this search by removing some of the lower level details.[4] The levels that contain only plain literals separate the details from the more important aspects of a problem. The levels involving binding literals delay the generation of bindings as long as possible, which can reduce backtracking.

## 4.5 Discussion

This chapter identified the monotonicity and ordered monotonicity properties and presented an algorithm for automatically generating ordered monotonic abstraction hierarchies. The monotonicity property is useful because it provides a criterion for pruning the search while still maintaining completeness. The ordered monotonicity property is useful because it captures a large class of interesting abstractions and it is tractable to find abstraction hierarchies that have this property. The algorithm for generating abstractions is implemented in ALPINE, which automatically generates

[4]The idea of separating out the recursive literals was inspired by the work of Etzioni [1990], which identified the importance of nonrecursive explanations for explanation-based learning.

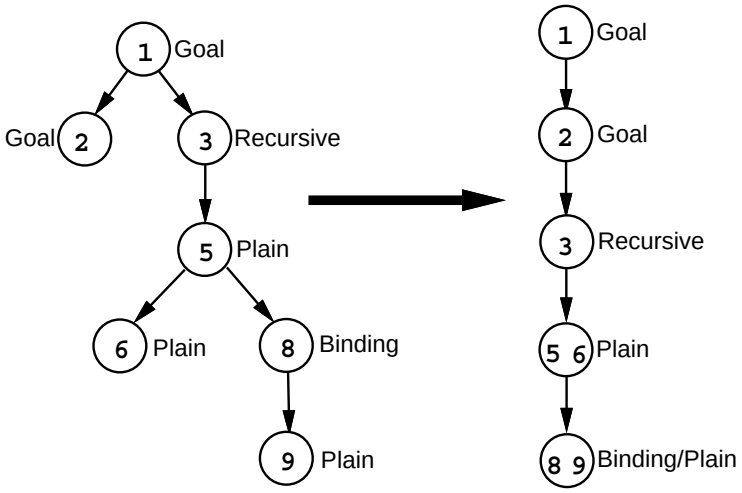


Figure 4.8: Selecting a Total Order from a Partial Order

abstractions for Hierarchical PRODIGY. While the abstractions produced by ALPINE are intended to be used by Hierarchical PRODIGY, the basic approach to generating abstractions is applicable to any operator-based problem solver.

The next chapter describes results in four problem-solving domains and shows that ALPINE produces effective abstraction in all four domains. There results serve to demonstrate both that the ordered monotonicity property does capture a useful class of abstractions and that these abstractions can produce significant reductions in search.

# Chapter 5

# Empirical Results

This chapter describes the results of both generating and using abstractions for problem solving. The abstractions are generated by ALPINE and then used in the hierarchical version of PRODIGY. The chapter is divided into four sections. The first section demonstrates empirically, in the Tower of Hanoi, that abstraction can produce an exponential-to-linear reduction in search. The second section describes empirical results in several more complex domains to show that ALPINE can generate effective abstraction hierarchies for a variety of problem domains. The third section compares the performance of ALPINE's abstractions with both hand-coded and automatically generated search control knowledge. The last section compares ALPINE and ABSTRIPS in the original STRIPS domain and shows that ALPINE produces abstractions that have a considerable performance advantage over those generated by ABSTRIPS.

## 5.1 Search Reduction: Theory vs. Practice

As described in Section 4.3, ALPINE generates a hierarchy of abstraction spaces for the Tower of Hanoi that separates the various sized disks into separate abstraction levels. In the most abstract level it plans the moves for the largest disk, at the next level it adds the moves for the next largest disk, and so on. Section 3.4 showed analytically that this abstraction produces an exponential-to-linear reduction in the size of the search space. This section provides empirical confirmation of this result and then explores the search reduction when the problem solver uses a "nonadmissible" search procedure (a search procedure that is not guaranteed to produce shortest solutions).

The degree to which the use of abstraction reduces search depends on the amount of search required to find a solution without using abstraction. Admissible search procedures such as breadth-first search or depth-first iterative-deepening are guaranteed to produce the shortest solution and to do so will usually search most of the

search space. However, these methods are impractical in more complex problems, so this section also examines the use of the abstractions with a depth-first search. Not surprisingly, these results show that the actual reduction in search largely depends on how much of the search space is actually searched by the problem solver without using abstraction. In addition, the problem solver can sometimes trade off solution quality for solution time by producing longer solutions rather than searching for better ones.

To evaluate empirically the use of ALPINE's abstraction in the Tower of Hanoi, PRODIGY was run both with and without the abstractions using a depth-first iterative-deepening search and depth-first search. The experiments compare the CPU time required and the length of the solutions on problems that range from one to seven disks. In the CPU time comparisons, the time required to create the abstraction hierarchies is included in the problem-solving time. The graphs below measure the problem size in terms of the optimal solution length, not the number of disks, since the solution to a problem with $n$ disks is twice as long as the solution to a problem with $n-1$ disks. For example, the solution to the six-disk problem requires 63 steps and the solution to the seven-disk problem requires 127 steps.

Figure 5.1 compares PRODIGY with and without abstraction using a depth-first iterative-deepening search to solve each of the subproblems. As the analytical results predict, the use of abstraction with an admissible search procedure produces an exponential reduction in the amount of search. Without the use of abstraction, PRODIGY was unable to solve the four-disk problem within the 600 CPU second time limit. The results are plotted with the problem size along the x-axis and the CPU time used to solve the problem along the y-axis. In the Tower of Hanoi, the use of an admissible search produces optimal (shortest) solutions both with and without abstraction.

Admissible search procedures such as breadth-first search or depth-first iterative-deepening are guaranteed to produce the shortest solution[1] and to do so usually requires searching most of the search space. However, these methods are impractical in more complex problems, so this section also examines the use of hierarchical problem solving with a nonadmissible search procedure. Figure 5.2 compares the CPU times and solution lengths with and without abstraction using depth-first search. As the graphs shows, the use of abstraction produces only modest reductions in search times and solution lengths. This is because, using depth-first search, neither configuration is performing much search. When the problem solver makes a mistake it simply proceeds adding steps to undo the mistakes. Thus, the amount of search performed by each configuration is roughly linear in the length of the solutions found. Problem solving with abstraction performed better because the abstraction provides some guidance on which goals to work on first and thus produces shorter solutions

[1] If an admissible search procedure is used to solve each of the subproblems in hierarchical problem solving, then the solution to each subproblem will be the shortest one possible, but the solution to the entire problem may still be suboptimal.

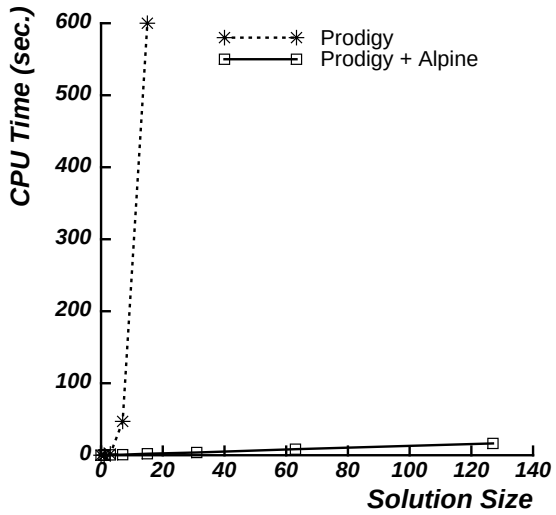


Figure 5.1: Depth-First Iterative-Deepening Search in the Tower of Hanoi
ALPINE reduces search time exponentially.

by avoiding some unnecessary steps.

The small difference between depth-first search with and without using abstraction is largely due to the fact that the problems can be solved with relatively little backtracking. To illustrate this point, consider a variant of the Tower of Hanoi problem that has the additional restriction that no disk can be moved twice in a row

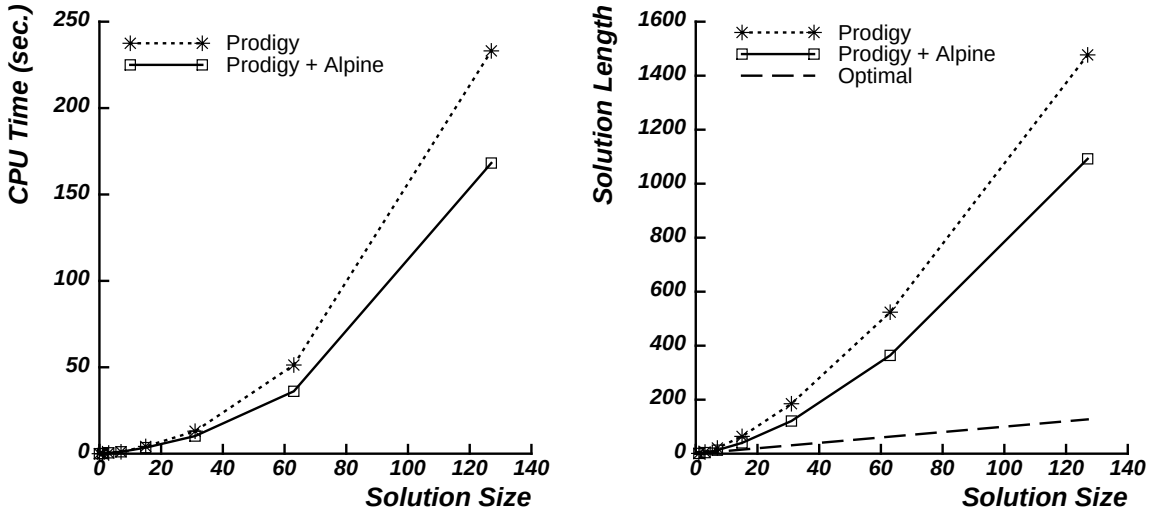


Figure 5.2: Depth-First Search in the Tower of Hanoi
ALPINE produces modest improvements in both search time and solution quality.

[Anzai and Simon, 1979, Amarel, 1984, VanLehn, 1989]. This constrains the problem considerably since the suboptimal plans in the previous graph were caused by moving a disk to the wrong peg and then moving the same disk again. By imposing additional structure on the problem, the problem solver is forced to search a larger portion of the search space to find a solution and as a result the abstraction will provide a greater reduction in search. Figure 5.3 compares the CPU time and the solution lengths for the two configurations on this variant of the domain. This small amount of additional structure enables the abstract problem solver to produce the optimal solution in linear time, while PRODIGY produces a suboptimal solution that requires significantly more problem-solving time.

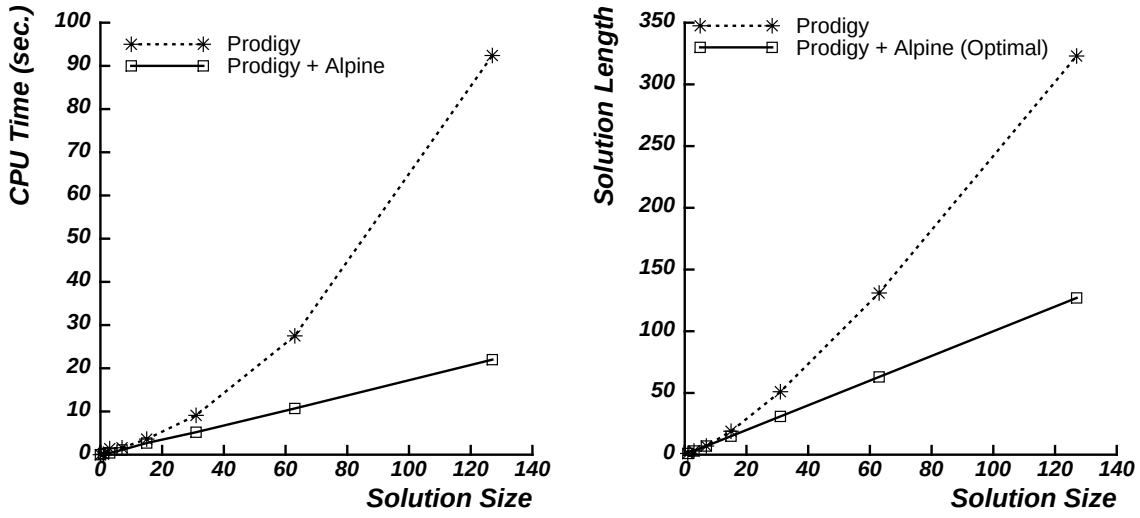


Figure 5.3: Depth-First Search in a Variant of the Tower of Hanoi
ALPINE significantly reduces the search time and produces optimal solutions.

The Tower of Hanoi is perhaps a bit unusual in that the structure of the search space allows the problem solver to undo its mistakes by simply inserting additional steps. In domains that are more constrained, the problem solver may be forced to backtrack and search a fairly large portion of the search space to find a solution. In those domains the use of abstraction will provide more dramatic search reductions with a depth-first search. In the less constrained domains, the problem solver can simply trade solution quality for search time. Thus, using a depth-first search, abstraction can reduce both the search and the solution length, and the reduction of each depends on the structure of the problem domain.

## 5.2 Empirical Results for ALPINE

ALPINE generates abstraction hierarchies for a variety of problem solving domains. This section describes the abstractions generated by ALPINE on two problem solving domains and the use of these abstractions in PRODIGY to reduce search. These domains were previously described in [Minton, 1988a], where they were used to evaluate the effectiveness of the explanation-based learning (EBL) module in PRODIGY. The first domain is an extended version of the original STRIPS robot planning domain and the second domain is the machine-shop planning and scheduling domain that was described in Section 2.3.

### 5.2.1 Extended STRIPS Domain

This section describes the abstraction hierarchies generated by ALPINE for the extended version of the robot planning domain. This domain is an extended version of the STRIPS robot planning domain [Fikes and Nilsson, 1971] and includes locks, keys, and a robot that can both push and carry objects. These extensions make the domain considerably more complex since there are multiple ways to achieve the same goals and there are many potential dead-end search paths because of locked doors and potential unavailability of keys. The version of the domain used for the experiments differs syntactically from the original extended STRIPS domain [Minton, 1988a]. These minor syntactic differences allow ALPINE to produce finer-grained abstraction hierarchies. Appendix B describes the differences and provides a complete specification of the problem space.

The definition of the problem space includes some control information that was not included in the original problem space. As described in Section 4.4.1, the problem space definition includes a specification of the primary effects, which ALPINE uses to construct the abstraction hierarchies. To avoid an unfair bias in the favor of ALPINE, the primary effects are translated into control knowledge, which is given to all of the systems in the comparisons. The inclusion of this control knowledge improves the problem-solving performance in this domain, allowing the problem solver to solve considerably more complex problems, even in the absence of abstractions.

ALPINE produces abstractions in this domain that both reduce search time and produce shorter solutions. As described earlier, each abstraction hierarchy is automatically tailored to the particular problem based on the parts of the domain that are relevant to solving the particular problem. For example, an abstraction hierarchy for a problem that simply involves moving the robot into a particular room can completely disregard any conditions involving boxes. Similarly, whether a door is open or closed may or may not be a detail depending on whether or not it occurs in the goal statement. If it is in the goal, it is no longer a detail since the plan may require

additional steps to get the robot to a room in which the goal can be achieved.

In constructing the abstraction hierarchies for this domain, ALPINE uses 33.9 CPU seconds to perform the one-time preprocessing of the domain. To construct the abstraction hierarchies for each of the test problems requires an average of 1.5 CPU seconds and ranges from 0.4 to 4.5 CPU seconds. The problem-solving times reported in this chapter include the time required to construct an abstraction hierarchy, but not the time required to perform the preprocessing since that only needs to be done once for the entire domain.

Consider a problem that was taken from the set of randomly generated test problems for this domain. The problem consists of moving three boxes into a configuration that satisfies the following goal:

```
(and (next-to a d)(in-room b room3)(in-room a room4)).
```

The randomly generated initial configuration is shown in Figure 5.4. The complete specification of this problem (#173) is given in Appendix B. Boxes and keys are scattered among the set of rooms and the doors between the rooms can be either open (OP), closed (CL), or locked (LO). The names of the keys are based on the rooms they connect (e.g., K36 is the key to the door connecting `room3` and `room6`). This particular problem is difficult for two reasons. First, box `A` has two constraints that must be satisfied in the goal statement: box `a` must be next to box `d` and box `a` must be in `room4`. Second, some of the doors in the initial state are locked and the

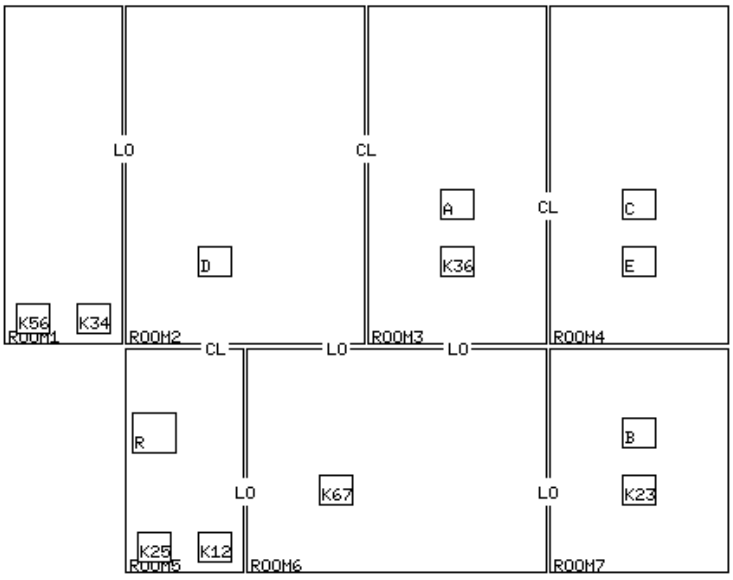


Figure 5.4: Initial State for the Robot Planning Problem

robot, which starts out in `room5`, will need to go through at least two of the locked doors to solve the problem.

To construct an abstraction hierarchy for this problem, ALPINE first augments the goal using the axioms described in Section 4.4.1 and then finds an ordered monotonic abstraction hierarchy for the augmented problem. The example problem would be augmented as follows:

```
(and (next-to a d)(in-room b room3)(in-room a room4)(in-room d room4))
```

where there is an added condition that box `d` is in `room4`. This follows from the axiom that states that if two boxes are next to each other then they must be in the same room. The system constructs the abstraction hierarchy using the algorithm described in the previous chapter. The resulting three-level abstraction hierarchy is shown in Figure 5.5. The first level in the hierarchy deals with getting all of the boxes into the correct room. The second level considers the location of both the robot and the keys, whether doors are locked or unlocked, and getting the boxes next to each other. The third level contains only details involving moving the robot next to things and opening and closing doors.

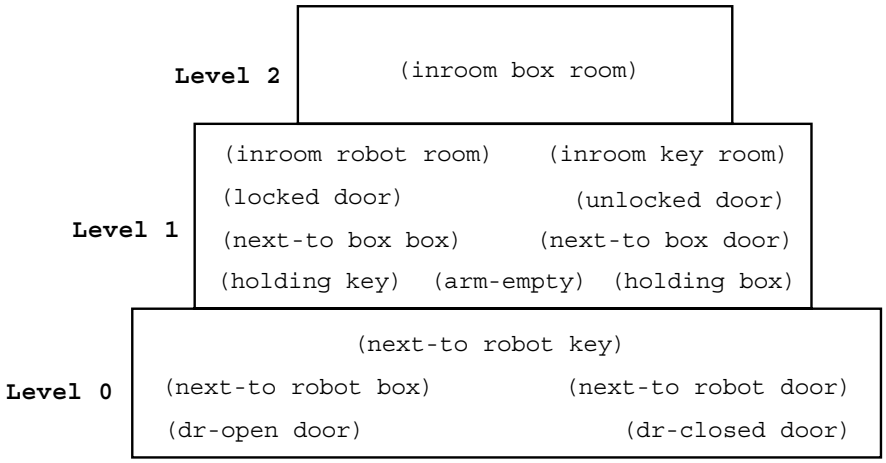


Figure 5.5: Abstraction Hierarchy for the Robot Planning Problem

The abstraction hierarchy for this problem has several important features. First, the problem of getting the boxes into the final rooms is solved before moving the boxes next to each other. Thus, the planner will not waste time moving two boxes next to each other only to find that one or both of the boxes needs to be placed in a different room. Second, the conditions at the second level can require a fair amount of search – doors may need to be unlocked and thus keys must be found – but achieving these conditions will not interfere with the more abstract space that

deals with the location of the boxes. Note, however, that it may not be possible to refine the abstract plan because some door cannot be unlocked. This does not violate the ordered monotonicity property, but may require returning to the abstract space to formulate a different abstract plan. Third, the conditions at the final level in the hierarchy are details that can be solved independently of the higher level steps and inserted into the abstract plan. Once conditions like whether doors are locked or unlocked are considered, it will always be possible to open and close the doors.

Before comparing the overall performance of the hierarchical problem solver using ALPINE's abstractions to problem solving without any use of abstraction, consider the results of using the abstraction hierarchy for this problem. As Table 5.1 shows, the use of the abstraction hierarchy produces a ten-fold speedup in solution time, reduces the amount of search by a factor of twenty, and produces a solution that is almost half the length of the solution produced without abstraction.

| System | CPU Time (sec.) | Nodes Searched | Solution Length |
|---|---|---|---|
| Prodigy | 194.6 | 4069 | 76 |
| Prodigy + Alpine | 19.2 | 194 | 45 |
| Ratio: | 10.1 : 1 | 21.1 : 1 | 1.7 : 1 |

Table 5.1: Performance Comparison for the Robot Planning Problem

Table 5.2 shows the problem-solving search and solution steps at each level in the abstraction hierarchy. Level two, the most abstract level, produces a five-step plan for moving the boxes into the correct rooms. This level requires little search since it only requires finding paths to the destination rooms. Level one requires an additional twenty-one steps to find the keys and unlock the doors, move the robot to the necessary places for moving the boxes, and move the boxes next to each other. This level requires the most search because of the difficulty of finding paths through rooms while dealing with doors that may be locked. The final level inserts an additional nineteen steps, but effectively requires no search (every step in the solution requires two nodes if there is no search). Note that while these steps are individually easy

| | Level 2 | Level 1 | Level 0 | Total |
|---|---|---|---|---|
| Nodes Searched | 15 | 141 | 38 | 194 |
| Solution Lengths | 5 | 21 | 19 | 45 |

Table 5.2: Breakdown of the Abstract Search for the Robot Planning Problem

to achieve, separating them from the level above considerably simplifies the problem solving in the more abstract space.

The use of ALPINE's abstractions does not always improve performance and, in some cases, can actually degrade the performance compared to problem solving without using abstraction. There are three possible ways in which ALPINE can degrade the performance on a particular problem. First, the added cost of constructing and using the abstraction hierarchy can dominate the problem-solving time on problems that can be solved easily without using abstraction. Second, since PRODIGY uses a depth-first search, the use of abstraction could lead the problem solver down a different path than the default path that would have been explored first without using abstraction, which can result in more search to find a solution. Third, the use of a particular abstraction could degrade performance by producing abstract plans that cannot be refined and require backtracking across abstraction levels to find alternative abstract plans. Despite these problems, the use of abstraction still produces significant performance improvements overall.

To evaluate the abstraction hierarchies produced by ALPINE, this section compares problem solving with ALPINE's abstractions to problem solving in PRODIGY with no control knowledge and problem solving in PRODIGY with a set of hand-coded control rules (HCR). The comparison was made on a set of 250 randomly generated problems. Of these problems, 100 were used in Minton's experiments [Minton, 1988a] to test the EBL module. The hand-coded control rules correspond to the ones that were used in the EBL experiments. Because of the additional information about primary effects used in this comparison, these problems proved quite easy for the problem solver even without the use of abstraction. Thus, an additional set of 150 significantly more complex randomly generated problems was also used in the comparison. The harder problems were generated by increasing the number of goal conjuncts. The experiment compared PRODIGY running without the use of abstraction to the hierarchical version of PRODIGY using an abstraction for each problem generated by ALPINE. The different configurations were allowed to work on each problem until it was solved or the 600 CPU second time limit was exceeded.

Comparing the results of the different configurations on the set of test problems is complicated by the fact that some of the problems cannot be solved within the time limit. Similar comparisons in the past have been done using cumulative time graphs [Minton, 1990], but Segre et al. [1991] argue that such comparisons could be misleading because changing the time limit can change the results. To avoid this problem, the total time expended solving all of the problems is graphed against the CPU time bound. The resulting graph illustrates three things. First, each curve on the graph shows the total time expended on all the problems as the time bound is increased. Second, the slope at each point on a curve indicates the relative portion of the problems that remain unsolved. A slope of zero means that all of the problems

have been solved (no more time is required to solve any of the problems). Third, the shape of the curve can be extrapolated to estimate the relative performance of the systems being compared as the time bound is increased.

Figure 5.6 provides the time-bound graphs for the test problems in the extended robot planning domain. The graphs separate the solvable problems from the unsolvable problems (those problems that have no solution). Unsolvable problems can be considerably harder since the problem solver may have to explore every possible alternative to prove that a problem has no solution. The graph on the left contains the 206 solvable problems and the one on the right contains the remaining 44 unsolvable problems. On the solvable problems, PRODIGY with abstraction can solve all the solvable problems in less than 200 CPU seconds. In contrast, both PRODIGY alone and PRODIGY with the hand-coded control rules have still not solved some of the problems after 600 CPU seconds. In addition, the total time spent by PRODIGY is over three times that of using abstraction. On the unsolvable problems the difference between the use of abstraction and no abstraction is less dramatic, although ALPINE has solved more of the problems in considerably less time than PRODIGY.

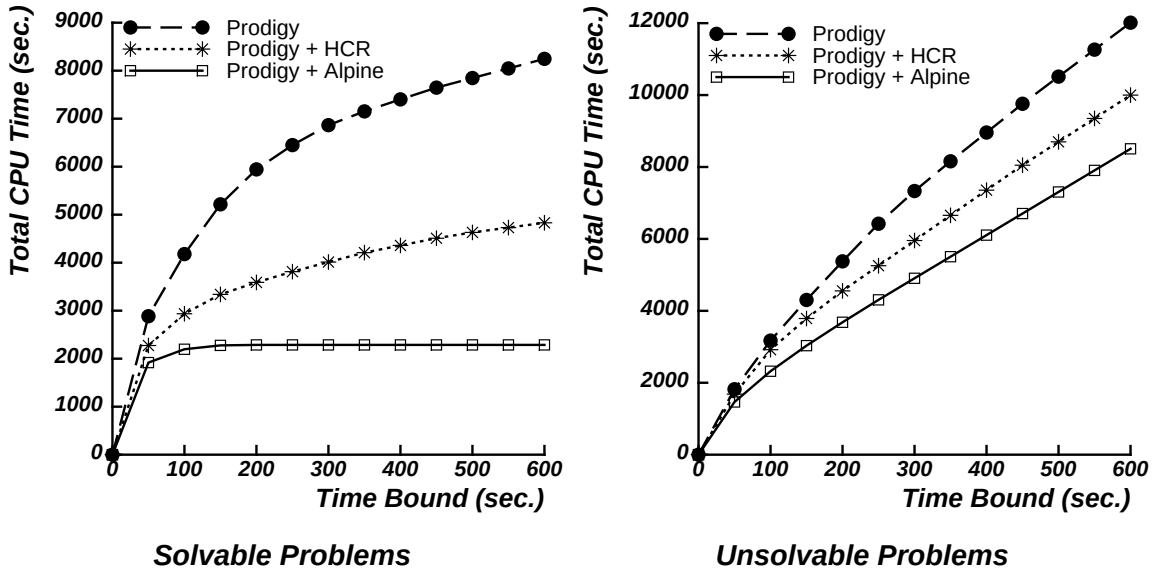


Figure 5.6: Total CPU Times in the Robot Planning Domain

While the time-bound graph provides a good comparison of the overall performance of several configurations, it does not provide any comparison of how the configurations compare on different sized problems or how different configurations will scale as the problems get harder. To provide such a comparison, the average solution time and average solution length are graphed against increasing solution size. The actual difficulty of a problem depends on many factors, including the size of the search

space, the solution length, the number of goal conjuncts, the size of the initial state, the degree of interaction among the goals, etc, and a more thorough analysis should probably take into account all of these aspects. However, since there is no widely accepted measure of problem complexity and there is a reasonably close correspondence between solution length and problem difficulty, these graphs use the average solution length of the three different configurations as the measure of the complexity of a problem. To reduce the variation in problem difficulty, the problems of different sizes are partitioned into larger sets. Thus, the problems that have a solution length between 1 and 20 are grouped into one set, and the problems with solution lengths between 21 and 40 are grouped into one set, and so on. The average time and solution length are then computed on these sets of problems.

Figure 5.7 shows the average solution time and average solution length as the problems increase in complexity. The graph on the left shows the average solution time on the 206 problems that could be solved by any system, and the graph on the right shows the average solution length on the 202 problems that could be solved by all systems. (Including problems that exceeded the time bound in the average solution time underestimates the average, but provides a better indication of overall performance than if these problems were excluded.) The average solution-time graph shows that on simple problems PRODIGY, both with and without control rules, performs about the same or slightly better than ALPINE, but as the problems become harder the use of abstraction clearly pays off. PRODIGY's better performance on the simpler problems is due to the added overhead of selecting and using the abstractions on problems for which the abstraction provides little benefit. The average solution-length graph shows that overall ALPINE produces solutions that are slightly better than PRODIGY (on average up to 10% shorter), but they are quite close and in some cases the solutions are worse.

### 5.2.2 Machine-Shop Scheduling Domain

This section describes the abstractions generated by ALPINE in the machine-shop process planning and scheduling domain. As described in Section 2.3, the domain involves planning and scheduling a set of machining operators on a set of parts being manufactured. The complete specification of this domain can be found in Appendix C. There are a few minor syntactic difference between the problem space used in the experiments and Minton's original definition of the problem space. These differences are described in the appendix.

ALPINE finds two useful types of abstraction in this domain. First, in many cases it can separate the top-level goals into separate abstraction levels, which reduces the search for a valid ordering of the operations. Second, it separates the process planning (the selection and ordering of the operations on the parts) from the actual scheduling

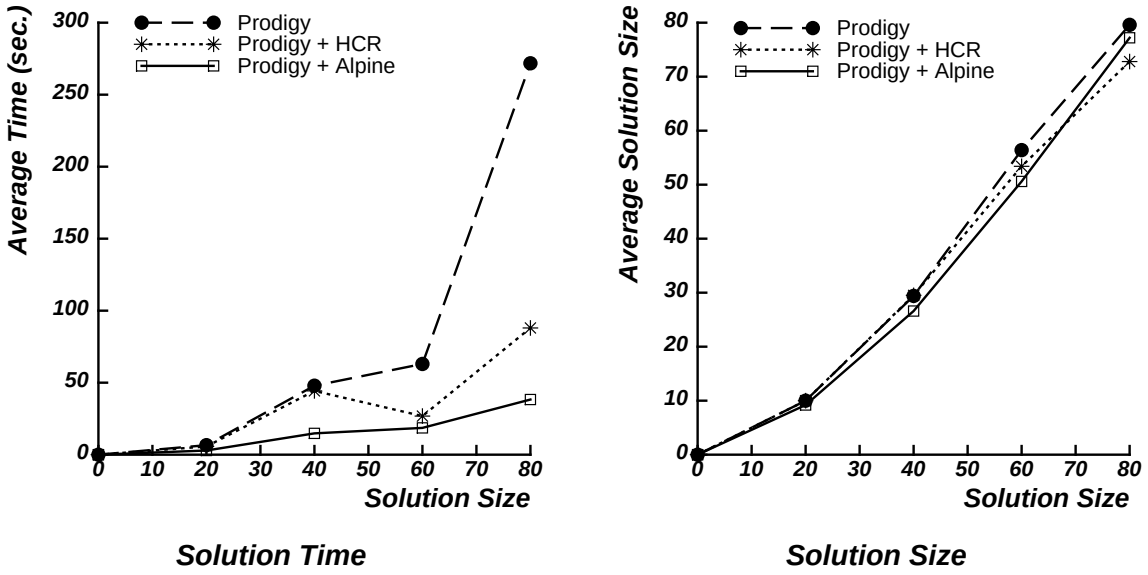


Figure 5.7: Average Solution Times and Lengths in the Robot Planning Domain

of the operations (only one part can be assigned to one machine at a given time). This allows the problem solver to find a legal ordering of the operators before it even considers placing the operations in the schedule.

In constructing the abstraction hierarchies for this domain, ALPINE uses 14.9 CPU seconds to perform the one-time preprocessing of the domain. To construct the abstraction hierarchies for each of the test problems requires an average of 1.4 CPU seconds and ranges from 0.4 to 3.8 CPU seconds. The problem-solving times reported in this chapter include the time required to construct an abstraction hierarchy, but not the time required to perform the preprocessing.

Consider the following problem in the scheduling domain, which involves making two parts:

```
(and (has-hole d (4 mm) orientation-4)
     (shape d cylindrical)
     (surface-condition e smooth)
     (painted d (water-res white)))
```

Part `d` needs a hole and needs to be made cylindrical and painted white. Part `e` simply needs to be made smooth. The complete specification of this problem (#181) is given in Appendix C. The resulting abstraction hierarchy for this problem is shown in Figure 5.8. The hierarchy separates the selection and ordering of the various operations and performs the scheduling last.

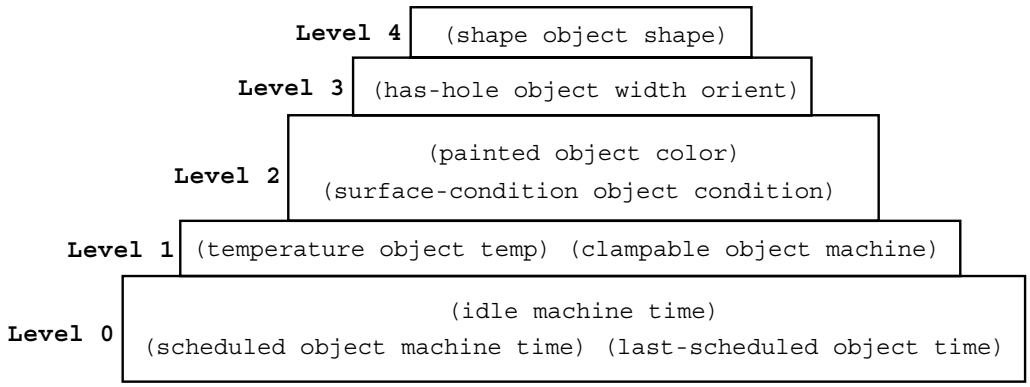


Figure 5.8: Abstraction Hierarchy for the Machine-Shop Problem

This abstraction produces a considerable improvement in problem-solving performance. The results are shown in Table 5.3, where ALPINE solves the problem significantly faster with much less search. The problem solving is broken up into five levels, where the distribution of search and solution length is as shown in table 5.4.

| System | CPU Time (sec.) | Nodes Searched | Solution Length |
|---|---|---|---|
| Prodigy | 164.7 | 5150 | 9 |
| Prodigy + Alpine | 7.0 | 39 | 9 |
| Ratio: | 23.4 : 1 | 132.1 : 1 | 1 : 1 |

Table 5.3: Performance Comparison for the Machine-Shop Problem

| | Level 4 | Level 3 | Level 2 | Level 1 | Level 0 | Total |
|---|---|---|---|---|---|---|
| Nodes Searched | 6 | 8 | 12 | 5 | 8 | 39 |
| Solution Lengths | 1 | 1 | 2 | 1 | 4 | 9 |

Table 5.4: Breakdown of the Abstract Search for the Machine-Shop Problem

This section provides a comparison analogous to the one for the extended robot planning domain described in the last section. It compares the performance of ALPINE to PRODIGY with no control knowledge, and PRODIGY with a set of hand-coded control rules. The hand-coded rules are the same rules that were used in the original comparisons with the EBL system [Minton, 1988a]. All the configurations were run

on 250 randomly generated problems including the 100 problems used for testing the EBL system.

The first comparison, shown in Figure 5.9, graphs the total time against an increasing time bound for the solvable and the unsolvable problems. On the 186 solvable problems, ALPINE performs better than PRODIGY both with and without control knowledge. On the 64 unsolvable problems, ALPINE performs better than PRODIGY without control knowledge. With control knowledge PRODIGY can quickly show for most of the problems that the problems have no solution. After 600 CPU seconds ALPINE and PRODIGY with control knowledge have used the same total time, but the slopes of the lines at 600 seconds show that ALPINE has completed more of the problems. This can be explained by the fact that control knowledge can often immediately determine that a problem is unsolvable, while the use of abstraction requires completely searching at least the most abstract space to determine that a problem is unsolvable. If there is no control rule to identify an unsolvable problem, then PRODIGY without using abstraction would have to search the entire space. Thus, the control knowledge can determine that a problem is unsolvable quickly, but the use of abstraction produces better coverage.

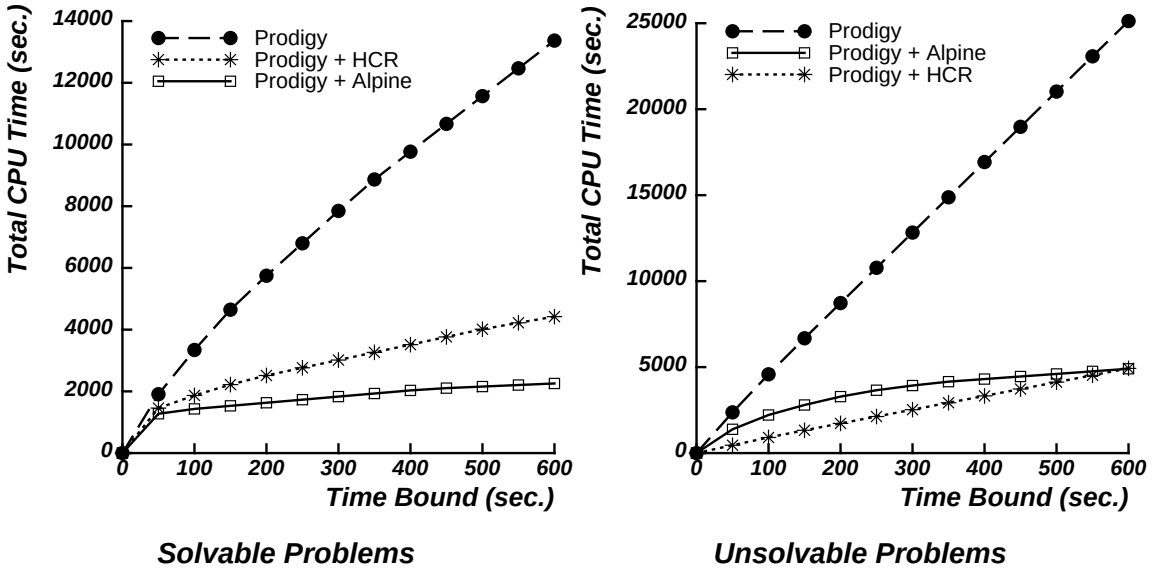


Figure 5.9: Total CPU Times in the Machine-Shop Domain

The second comparison, shown in Figure 5.10, graphs the average solution time and average solution length against increasing problem size. On the average solution length for the 163 problems that could be solved by all the configurations, ALPINE produces slightly shorter solutions than PRODIGY with and without control knowledge. On the average solution time for the 186 solvable problems, PRODIGY does well on

the easier problems, but performs poorly as the problems get harder. PRODIGY performs slightly better on the simplest problems because of the overhead of generating and using the abstraction hierarchies. Using control knowledge improves PRODIGY's performance considerably, but ALPINE still performs better than the other two configurations. ALPINE has trouble with a few of the more difficult problems because as the problems get larger, there are more constraints on the abstraction hierarchy and this results in fewer abstraction levels. In many cases the abstraction hierarchy is overconstrained and there are better ordered monotonic abstraction hierarchies that ALPINE does not produce. An extension to ALPINE that avoids this problem is described in Section 7.2.1.

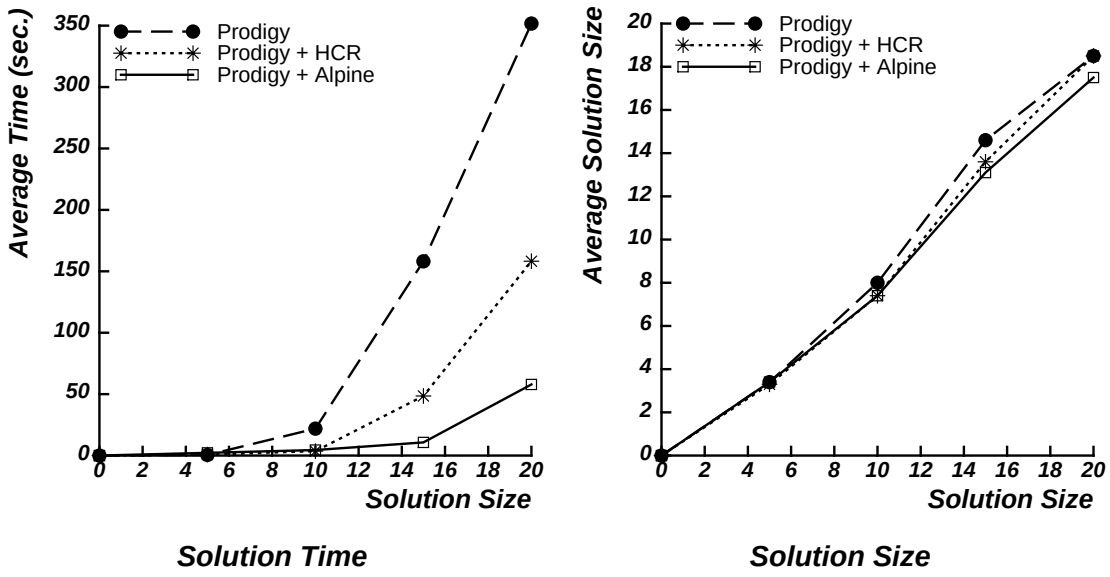


Figure 5.10: Average Solution Times and Lengths in the Machine-Shop Domain

## 5.3 Comparison of ALPINE and EBL

A significant amount of work in PRODIGY has focused on learning search control to reduce search. Minton [1988a] developed a system called PRODIGY/EBL that learns search control rules using explanation-based learning. More recently, Etzioni [1990] developed a system called STATIC that generates control rules using partial evaluation. This section compares the use of the abstractions generated by ALPINE to these two systems for learning search control knowledge.

The learning systems are compared in the machine-shop scheduling domain that was described in the previous section. The comparisons below mirror the ones de-

scribed in the last section. In addition to PRODIGY alone, with the hand-coded control rules (HCR), and with ALPINE's abstractions, the graphs also include PRODIGY with the control rules produced by EBL, with the control rules produced by STATIC, and the combination of ALPINE's abstractions and the hand-coded control rules. A comparison in the extended STRIPS domain was not included because of the differences between the original domain and the one used in this thesis. Such a comparison would require rerunning both EBL and STATIC to learn new sets of control rules for the modified domain.

The first comparison, shown in Figure 5.11, graphs the total time against an increasing time bound for the solvable and unsolvable problems. On the solvable problems, ALPINE without any control knowledge performs about the same as STATIC's control rules and significantly better than the use of EBL's control rules. On the unsolvable problems, STATIC and EBL perform the same as the hand-coded control rules and use about the same total amount of time on the unsolvable problems, but ALPINE completes more of the problems after 600 CPU seconds than the other configurations.

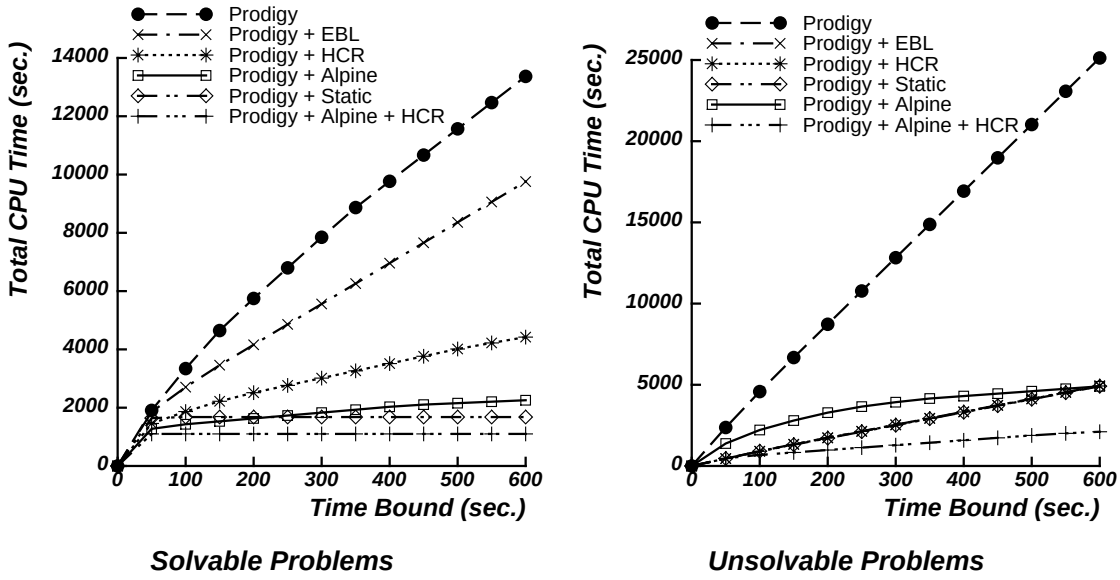


Figure 5.11: Total CPU Times for the Learning Systems in the Machine-Shop Domain

The second comparison, shown in Figure 5.12, graphs the average solution time and average solution length against increasing problem size for the solvable problems. As the problems get harder, ALPINE performs significantly better than EBL and slightly worse than STATIC on the average solution time. Both ALPINE and STATIC perform well, but have trouble solving some of the harder problems. The solutions produced by PRODIGY alone and PRODIGY with the STATIC control rules and the hand-coded control rules are slightly longer than the rest, although the differ-

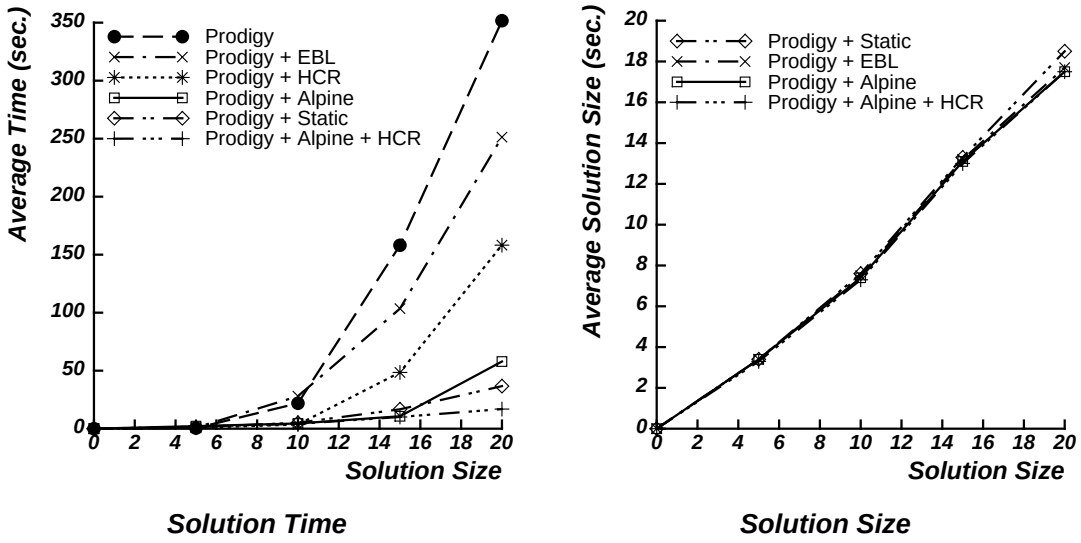


Figure 5.12: Average Solution Times and Lengths for the Learning Systems in the Machine-Shop Domain

ences are small. The obvious next step, which will be discussed in Section 7.3.3, is to combine these systems in order to learn the control knowledge to use in the abstract spaces.

The integration of the abstraction and the learning of control knowledge assumes that these approaches provide complementary sources of knowledge. While these systems have not yet been fully integrated, the figures above also graph the combination of the abstraction with the hand-coded control knowledge to demonstrate that the integration will provide improved performance. The combination of the abstraction and control knowledge, as shown in Figure 5.11, produces significantly better performance than any system alone on both the solvable and unsolvable problems. For the solvable problems, Figure 5.12 shows that as the complexity of the problems increase, the combined system allows the problem solver to solve the problems in time linear to the problem complexity. This combination improves performance because the control rules provide search guidance within an abstraction level and the use of abstraction provides better coverage at a lower cost than just using the control rules.

## 5.4 Comparison of ALPINE and ABSTRIPS

This section compares the abstractions generated by ALPINE to those generated by ABSTRIPS and shows that ALPINE produces better abstractions with less specific domain knowledge than ABSTRIPS. ABSTRIPS was the first system that attempted to

automate the construction of abstraction hierarchies for problem solving and was only applied to the STRIPS robot planning domain. The resulting abstraction hierarchies were then used for problem solving in an extended version of the STRIPS planner [Fikes and Nilsson, 1971]. This section compares the abstraction hierarchy generated by ABSTRIPS to the dynamically-tailored abstraction hierarchies generated by ALPINE in the STRIPS domain. The STRIPS domain is a simpler version of the robot planning domain described in Section 5.2.1 and consists of a robot that can push boxes around and between a set of rooms. The abstractions generated by each system are then tested empirically in the PRODIGY problem solver.

ABSTRIPS is given an initial partial order of the predicates for a domain and then performs some analysis on the domain to assign criticality values to the preconditions of each of the operators. The criticalities specify which preconditions of each operator should be ignored at each abstraction level. Since the abstractions are formed by dropping preconditions, the resulting abstraction spaces are relaxed models, as described in Section 3.1.2. The technique used to construct the abstraction hierarchy is described in detail in Section 6.2.1. The basic idea is to separate those preconditions that could not be achieved in isolation by a short plan and then use the given partial order to assign criticalities to the remaining preconditions.

The abstractions generated by ALPINE differ from ABSTRIPS in several important ways. First, ALPINE completely automates the construction of the abstraction hierarchies from only the initial definition of the problem space, while ABSTRIPS requires an initial partial order to form the abstractions. Second, ALPINE forms abstractions that are tailored to each problem, and ABSTRIPS constructs a single abstraction hierarchy for the entire domain. Third, ALPINE forms reduced models where each level in the abstraction hierarchy is an abstraction of the original problem space, while ABSTRIPS forms relaxed models.

The best way to compare the abstractions generated by the two systems is to consider an example. The example below is taken from one of the 200 randomly generated test problems used to compare the systems. The goal state consists of five goal conjuncts as follows:

```
(and (in-room a room1)
     (status door56 closed)
     (status door12 closed)
     (in-room robot room3)
     (in-room b room6))
```

The initial state for the problem is shown in Figure 5.13. The complete specification of this problem (#88) is given in Appendix D. This problem is difficult because the doors must be closed after the boxes have been placed in the correct rooms and the robot must be on the correct side of the door when it is closed.

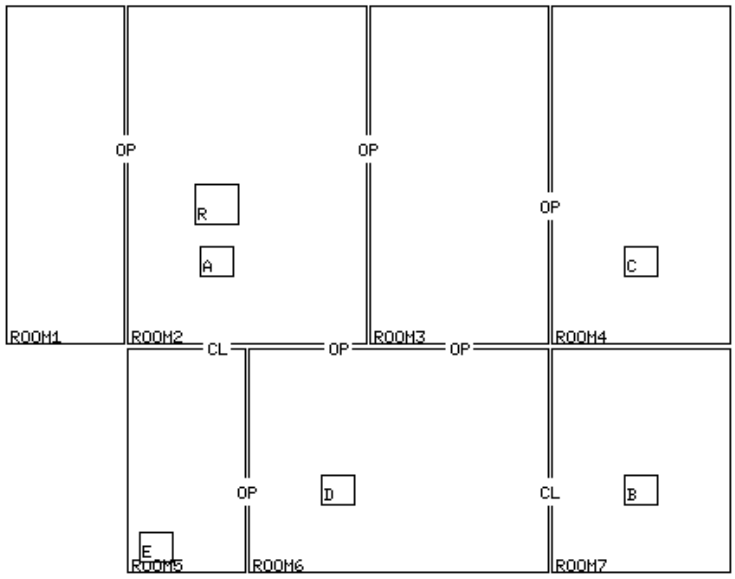


Figure 5.13: Initial State for the STRIPS Problem

The abstraction hierarchies generated by each system are shown in Figure 5.14. For the entire problem domain, ABSTRIPS uses the same four-level abstraction hierarchy. The most abstract space consists of all the static predicates (the predicates that cannot be changed), the second level consists of the preconditions that cannot be achieved by a short plan. This includes all of the `in-room` preconditions, and some of the `next-to` and `status` preconditions. The third level consists of the remaining `status` preconditions that can be achieved by a short plan, and the fourth level contains the remaining `next-to` conditions.

ALPINE can build finer-grained hierarchies using the type hierarchy (Section 4.4.1)

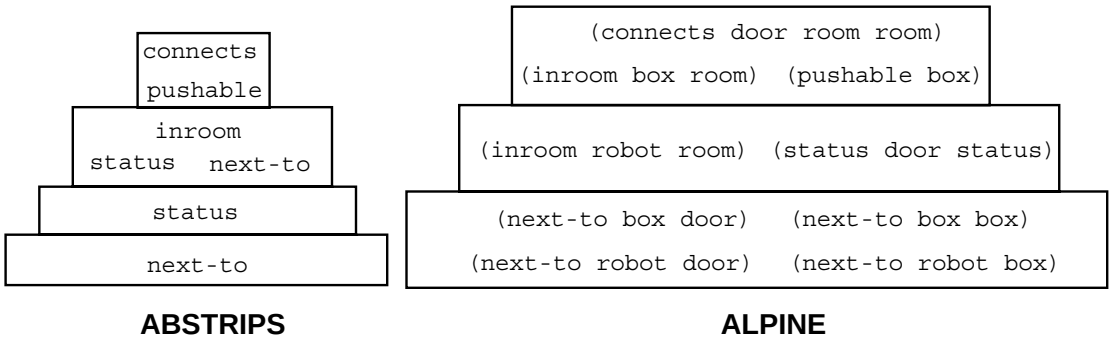


Figure 5.14: Abstraction Hierarchies Generated by ABSTRIPS and ALPINE

to separate literals with the same predicate but different argument types. The abstraction hierarchy built by ALPINE for this problem consists of a three-level abstraction hierarchy (the abstraction hierarchy selection heuristics described in Section 4.4.4 combine the bottom two levels into a single level). The most abstract space consists of all the static literals and the `(in-room box room)` literals. The next level contains both the `(in-room robot room)` and the `(status door status)` literals. These two sets of literals get combined to satisfy the ordered monotonicity property since it may be necessary to get the robot into a particular room to open or close a door. Finally, the last level contains the `next-to` literals for both the robot and the boxes. ALPINE uses 12.3 CPU seconds for the one-time preprocessing of this domain. The time required to construct an abstraction hierarchy for each problem ranges from 0.3 to 2.8 CPU seconds and is 1.2 CPU seconds on average.

The example problem illustrates a problem with the abstraction hierarchies that are formed by ABSTRIPS. Problem solving using this abstraction hierarchy proceeds as follows. Since ABSTRIPS only drops preconditions and not effects of operators, all the goals are considered in the abstract space. The system constructs a plan to move box `a` into `room1`, closes the door to the room, and then moves the robot through the closed door. When the system is planning at this abstraction level it ignores all preconditions involving door status, so it does not notice that it will later have to open this door to make the plan work. When the plan is refined to the next level of detail the steps are added to open the door before moving the robot through the door, deleting a condition that was achieved in the abstract space (which is a violation of the monotonicity property). At this point the problem solver would need to either backtrack or insert additional steps for closing the door again. In fact, the original ABSTRIPS system would not have even noticed that it had violated the precondition, and would simply produce an incorrect plan [Joslin and Roach, 1989, pg.100].

ALPINE would first solve this problem in the abstract space by generating the plan for moving the boxes into the appropriate rooms. At the next level it would deal with both closing the doors and moving the robot. If it closed the door from the wrong side and then tried to move the robot to another room, it would immediately notice the interaction since these goals are considered at the same abstraction level. After producing a plan at the intermediate level it would refine this plan into the ground space by inserting the remaining details, which consists of the conditions involving `next-to`.

To illustrate the difference between ALPINE's and ABSTRIPS's abstractions, the use of these abstractions are compared in PRODIGY. Strictly speaking, this is not a fair comparison because the abstraction hierarchies generated by ABSTRIPS were intended to be used by the STRIPS problem solver. STRIPS employed a best-first search instead of a depth-first search, so the problem of expanding an abstract plan that is then violated during the refinement of that plan would be less costly. Nevertheless,

the comparison emphasizes the difference between the abstraction hierarchies generated by ALPINE and ABSTRIPS and demonstrates that a poorly chosen abstraction hierarchy can degrade performance rather than improve it.

First consider the results on the example problem described above. Table 5.5 shows the CPU time, nodes searched and solution length. ALPINE produces a small performance improvement over PRODIGY and generates shorter solutions. In contrast ABSTRIPS takes almost 6 times longer than PRODIGY, although it too produces the same length solution as ALPINE.

| System | CPU Time (sec.) | Nodes Searched | Solution Length |
|---|---|---|---|
| Prodigy | 14.5 | 259 | 25 |
| Prodigy + Alpine | 10.2 | 114 | 19 |
| Prodigy + Abstrips | 83.0 | 1,631 | 19 |

Table 5.5: Performance Comparison for the STRIPS Problem

Figure 5.15 provides a comparison of the performance of PRODIGY without using abstraction, using the abstractions produced by ABSTRIPS, and using the abstractions produced by ALPINE on 200 randomly generated problems in the robot planning domain. PRODIGY was run in each configuration and given 600 CPU seconds to solve each of the problems. Out of the 200 problems, 197 of the problems were

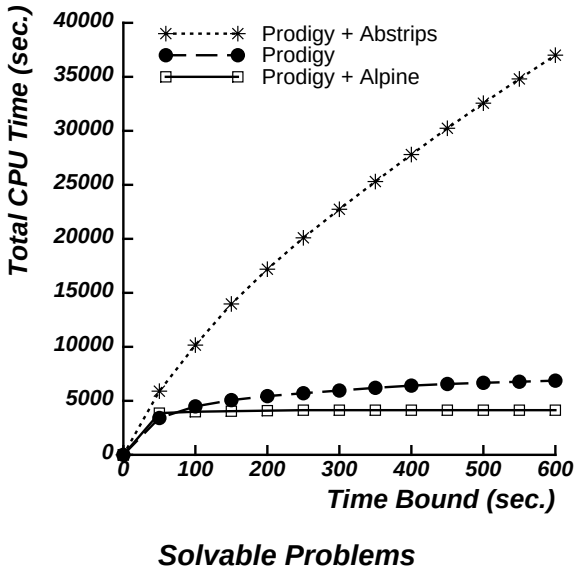


Figure 5.15: Total CPU Times in the STRIPS Domain

solvable in principle. The graph plots the total time spent on the solvable problems against increasing time bounds. It is clear from the graph that PRODIGY performs quite well on these problems even without abstraction. This is largely due to the fact that problems in this domain are much like the Tower of Hanoi, in that the problem solver only needs to search a small portion of the search space since most mistakes can be undone by adding additional steps. The graphs shows that the use of ABSTRIPS' abstractions significantly degrades performance, while ALPINE's abstractions improve performance enough that all 197 solvable problems are solved within 150 CPU seconds. (After 600 CPU seconds, PRODIGY has still not solved two of the problems.) None of the systems were able to determine that the three unsolvable problems were unsolvable.

Figure 5.16 shows the average solution times for the 197 solvable problems and the average solution lengths for the 153 problems that were solved by all three configurations. These graphs show that ALPINE produces shorter solutions in less time than either PRODIGY or ABSTRIPS. It is worth noting that while ABSTRIPS' abstractions significantly increased the problem solving time, they did improve the quality of the solutions. With respect to problem-solving time PRODIGY performed quite well, but as in the Tower of Hanoi, PRODIGY achieved this performance by trading solution quality. On the hardest set of problems, PRODIGY produces solutions that were on average fifty percent longer than ALPINE.

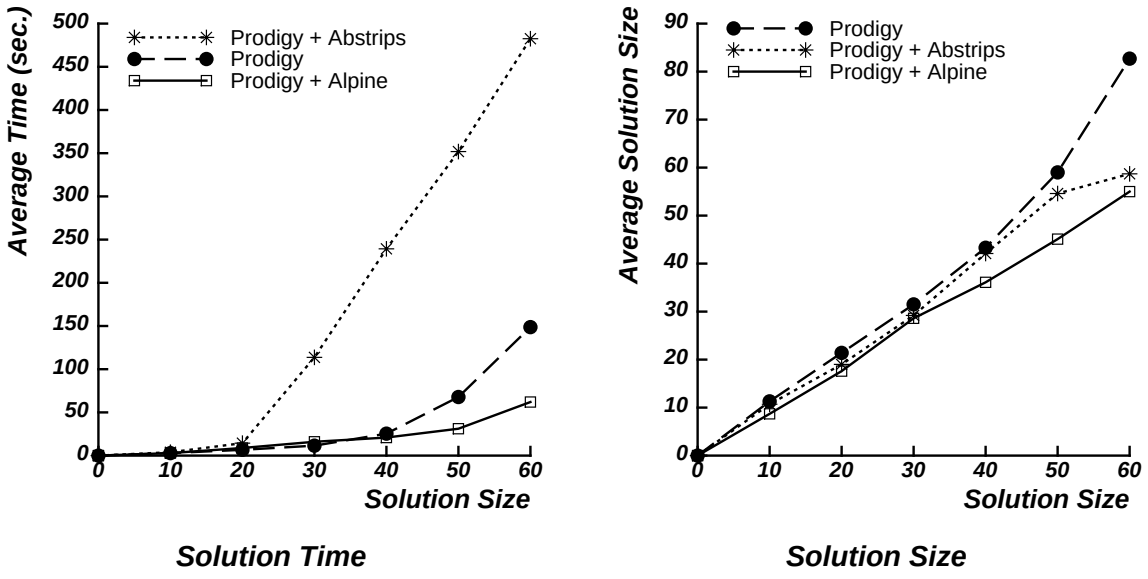


Figure 5.16: Average Solution Times and Lengths in the STRIPS Domain

# Chapter 6

# Related Work

This chapter describes the related work on abstraction in problem solving. The first section compares the approaches to problem solving using abstraction that are most closely related to the one described in this thesis. The second section compares the approach implemented in ALPINE for generating abstractions to other related techniques. The third section describes the related work on properties of abstractions.

## 6.1 Using Abstractions for Problem Solving

There are a variety of ways in which abstractions can be used in problem solving. Chapter 3 described an approach to using abstractions for hierarchical problem solving. This section compares this technique to the most closely related problem-solving techniques.

### 6.1.1 Abstract Problem Spaces

One approach to hierarchical problem solving, which was presented in Chapter 3, is to employ a set of well-defined abstraction spaces to solve problems at different levels of abstraction. An abstraction space can be either a simplification or reformulation of the original problem space, such as the relaxed and reduced model described in Section 3.1.2. A problem is usually solved in an abstract space and then refined at successively more detailed levels.

Planning GPS [Newell and Simon, 1972] was the first system to employ this approach to problem solving. The system was applied to the domain of propositional logic problems. In this logic domain, GPS is given an abstract problem space that ignores the logical connectives in the formulas. A problem is first solved in this abstract space and then refined into the ground space to replace the abstract operations with the appropriate operations on the connectives. In contrast to the abstract problem

spaces generated by ALPINE, the abstraction of the propositional logic problems does not just ignore conditions, but provides a different representation of the problem. While GPS provided the first automated use of abstraction for problem solving, it did not automate the construction of the abstractions.

ABSTRIPS [Sacerdoti, 1974] also employs abstract problem spaces for hierarchical problem solving. The abstraction spaces for a problem domain are represented by assigning criticalities, numbers indicating the relative difficulty, to the preconditions of each operator. The system uses these criticalities to plan abstractly. First, an abstract plan is found that satisfies only the preconditions of the operators with the highest criticality values. The abstract plan is then refined by considering the preconditions at the next level of criticality and inserting steps into the plan to achieve these preconditions. The process is repeated until the plan is expanded to the lowest level of criticality. If a solution to a subproblem cannot be found, the system starts over and reruns the problem solver to find a different abstract solution. Hierarchical PRODIGY uses the same basic approach to problem solving, but there are two significant differences. First, Hierarchical PRODIGY uses reduced models instead of simply dropping preconditions, which allows it to simplify the goals of a problem in an abstract space. Second, it maintains the dependency structure of the problem-solving trace so that the problem solver can efficiently backtrack across abstraction levels.

The motivation behind this general approach to problem solving is to use the abstractions to divide up a problem into a number of smaller subproblems. A problem is first solved in a simpler abstraction space and the abstract solutions can then be used to form a set of subproblems at the next level of abstraction. The subproblems are then solved at that level and the solutions in turn form subproblems at the next level. This process is repeated until the problem is solved in the original problem space. Polya [1945] was one of the first to describe this idea of decomposing a problem into a number of simpler subproblems. Since then several people have shown that dividing a problem into subproblems can produce significant reductions in search [Newell *et al.*, 1962, Minsky, 1963]. These analyses implicitly assume a problem can be divided into small, equal-sized, independent subproblems that can be solved without backtracking. Section 3.3 both formalizes this analysis and identifies a set of sufficient conditions to achieve significant reductions in search.

### 6.1.2 Abstract Operators

Another approach to hierarchical problem solving is to first build a plan out of a set of abstract operators and then refine the plan by selectively expanding individual operators into successively more detailed ones. The refinement is done using a set of *action reductions* [Yang, 1989], which specify the relationship between an abstract operator and the refinements of that operator. This approach differs from the previ-

ous one in that it does not require that a plan is expanded completely at one level of abstraction before refining it to the next level. Instead there are a set of abstractions for each operator, and each instance of an operator in an abstract plan can be expanded to a different level of detail. Since these systems do not employ a set of well-defined abstraction spaces, there is no notion of solving a problem at one level before refining the plan to the next level. This approach has been used extensively in least-commitment problem solvers and the systems that employ this approach include NOAH [Sacerdoti, 1977], MOLGEN [Stefik, 1981], and NONLIN [Tate, 1976]. In all of these problem solvers, the abstractions must be hand-crafted for each problem domain.

A problem with the use of action reductions for hierarchical problem solving is that it only provides a heuristic about the order in which preconditions of various operators should be expanded and does not necessarily provide a coherent abstraction of a problem. A problem solver may expand one part of the plan down to a given level and then work on a different part of the plan that then undoes conditions that were needed in the part of the problem already solved. This is equivalent to violating the monotonicity property. As Rosenschein [1981] points out, seemingly correct plans at one level can be expanded into incorrect plans at lower levels. This undermines the rationale for hierarchical planning of reducing complexity through factorization since "unexpected global interactions" can arise. NOAH dealt with this problem by maintaining a *hierarchical kernel*, which records for each node the conditions that were tested in order to apply the operator at that node. Then, before any node is expanded further, the hierarchical kernel for that node is tested to see if the appropriate conditions still hold. In the case where the hierarchical kernel does not hold, NOAH had to reachieve the missing conditions or backtrack to the point it undid one or more of these conditions. In general, reachieving the deleted conditions or finding another way to solve the problem that does not violate the conditions is a difficult problem.

There is nothing inherent in least-commitment problem solvers that prevents them from using action reductions to implement abstract problem spaces. SIPE [Wilkins, 1984, Wilkins, 1986] uses a more explicit encoding of the abstractions where the domain is partitioned into literals at different abstraction levels and operators for achieving those literals. While SIPE can still expand the operators in a plan to different levels of abstraction, the domains in SIPE are designed such that it will never undo some condition in a more abstract space in the process of refining some part of the plan.

However, Wilkins [1986] identified another problem that can arise with least-commitment problem solvers even when using abstraction spaces as in SIPE. The problem (called *hierarchical inaccuracy* [Yang, 1990]) is that since the planner can expand the operators in a plan to different levels of detail, it may expand one part

of the plan to too detailed a level before another part of the same plan is expanded at all. The result is that state inconsistencies can arise. For example, in one part of the plan the system may decide to move a robot from one room to another, while an earlier part of the plan, which has not been expanded, may also require moving the robot. Since the earlier part of the plan has not been expanded to the level of detail before the later part of the plan, the system may plan to move the robot from a room that the robot will have already left. Wilkins proposed several solutions to this problem, but they all require noticing when these situations can arise and annotating the operators in order to prevent the expansion at too detailed a level before earlier parts of the plan have been expanded to that level. Yang [1990] proposed an automated approach to avoid this problem that involves preprocessing a domain to find a set of syntactic restrictions on the action reductions. Problem solvers that employ abstraction problem spaces, such as Hierarchical PRODIGY and ABSTRIPS, avoid this problem by always expanding the plan at each level in a left-to-right order.

### 6.1.3 Macro Problem Spaces

Another problem-solving method, similar to the use of abstract problem spaces, is the use of macro problem spaces. Instead of abstracting operators to form an abstract problem space, operators are combined into macro operators to form a macro problem space [Korf, 1987]. This approach is similar to using abstraction spaces in that a problem is mapped into an abstract space, which is defined by a set of macro operators, and then solved in the abstract space. However, unlike the use of abstract problem spaces, once a problem is solved in the macro space, the problem is completely solved since the macros are defined by operators in the original problem space.

Korf [Korf, 1987] shows that a single level of abstraction can reduce the total search time from $O(n)$ to $O(\sqrt{n})$, and he shows that multiple levels of abstraction can reduce the search time from $O(n)$ to $O(logn)$, where n is the number of states and the time is proportional to the number of states searched. These results are based on an average case analysis that assumes the distribution remains constant over different levels of abstraction and the number and ratio of the sizes of the abstraction spaces are optimal.

Korf's analysis assumes that the mapping between an abstract plan and a specialization of the abstract plan is trivially known and a specialization always exists. In contrast, a hierarchical problem solver may expend a great deal of work searching for an appropriate specialization and in some cases no corresponding specialization exists. Consider the route planning domain that Korf describes in [Korf, 1987]. The problem is to plan a route between any two points where the operators are used to move between adjacent intersections. By building up a set of macro operators he creates abstract operators that move between more distant points. Planning involves

mapping the original problem into an abstract problem and then finding a route between the points in the abstract space. Once this abstract solution is found, the solution is found. Each macro operator is composed of the individual operators that achieve the macro so there is no additional work to map an abstract solution into a detailed solution. The difficultly with this approach is in finding a good set of macro operators. For the domains in which a set of macros can be defined, this technique will be useful, but there are few domains that are like the route planning domain and have a sufficiently regular structure to define a set of macro spaces.

## 6.2 Generating Abstractions for Problem Solving

This section compares the automated approach to generating abstractions described in this thesis to other related techniques. The related work is compared along three dimensions. First, what is the form of the knowledge produced by the technique (e.g., abstractions, aggregations, problem reductions, goal orderings). Second, how is this knowledge used for problem solving (e.g., subgoals, control rules, evaluation functions). Third, what is the approach to producing this knowledge (e.g., analysis of interactions, partial evaluation). This section is loosely organized by the type of knowledge produced by the various approaches.

### 6.2.1 Abstractions

ABSTRIPS [Sacerdoti, 1974] was the first attempt to automate the formation of abstraction hierarchies for hierarchical planning. However, the system only partially automates this process. The user provides the system with an initial partial order of predicates, which is used to assign criticalities automatically to the preconditions of ground-level operators. ABSTRIPS places the *static literals*, literals whose truth value cannot be changed by an operator, in the highest abstraction space. It places literals that cannot be achieved with a "short" plan in the next highest abstraction space. And it places the remaining literals at lower levels corresponding to their place in the user-defined partial order. It determines whether a short plan exists by assuming that the preconditions higher in the partial order hold and attempts to show the remaining conditions can then be solved in a few steps. The same literal in the preconditions of two different operators can be placed at two different levels because the difficulty of achieving a particular precondition depends on the other preconditions of the operator in which it occurs.

The essence of the approach in ABSTRIPS is the short-plan heuristic, which separates the details from the important information. The system automatically produces a three-level abstraction hierarchy, with the static literals at the top of the hierarchy,

the "important" literals next, and the details at the bottom. Any further refinement of levels comes from the user-defined abstraction hierarchy. In contrast, ALPINE can create a hierarchy of abstractions in which the levels of the hierarchy are successively more detailed. As shown in Section 5.4, ALPINE produces more effective abstractions with less knowledge than ABSTRIPS.

Christensen [1990] developed a system called PABLO that also generates hierarchies of abstractions for hierarchical problem solving. The system uses a technique called *predicate relaxation*, where it determines the number of steps needed to achieve each predicate by partially evaluating the operators. The problem solver solves a problem by focusing at each point on the part of the problem that requires the greatest number of steps. This general approach is similar to ABSTRIPS in that the abstractions are based on the number of steps needed to achieve the different conditions, but this approach allows successively more detailed abstraction spaces. A potential limitation, however, is that it may be expensive to partially evaluate the operators in more complex problem spaces, especially ones that involve recursive operators. While PABLO bases the abstractions on the length of the plan to achieve a goal, ALPINE forms abstractions based on partitioning the problem such that the conditions at one level do not interact with the conditions at a more abstract level.

Unruh and Rosenbloom [1989, 1990] present a weak method for SOAR [Laird *et al.*, 1987] that dynamically forms abstractions by dropping applicability conditions of the operators. If SOAR is working on a goal and reaches an impasse, a point in the search where it does not know how to proceed, then it performs a look-ahead search to resolve this impasse. Since this search can be quite expensive, an alternative is to perform a look-ahead search that ignores all of the unmatched preconditions. The choices made in the look-ahead search can then be stored by SOAR's chunking mechanism and the chunks are then used to guide the search in the original space. If the look-ahead search cannot distinguish between two choices, the unmatched conditions are iteratively expanded. This approach forms abstractions based on the heuristic that the more operators there are between a condition and the goal, the more likely it is to be a detail. When a useful abstraction is found it will be stored by the chunking mechanism. The system dynamically abstracts the operators, not to form abstract problem spaces, but to learn control heuristics.

A potential problem with the approach implemented in SOAR is that since the abstractions are formed by ignoring the preconditions that were unmatched in solving one particular problem, an abstraction that is effective in one situation could degrade performance in other situations. In contrast, since ALPINE constructs ordered monotonic abstraction hierarchies based on the potential interactions between the literals in a problem space, ALPINE's abstractions are more likely to ignore the appropriate conditions. The more stringent requirements on the abstractions formed by ALPINE, however, prevent it from finding abstractions in problem spaces in which SOAR can

produce abstractions (e.g., the eight-puzzle).

Anderson [1988] developed a system called PLANEREUS that automatically generates hierarchies of abstract operators and objects. The system constructs operator hierarchies by examining the operators that share common effects and forming new abstract operators that contain only the shared preconditions. Similarly, object hierarchies are formed by adding a new abstract object type when two operators perform the same operations on different objects. The operator and object hierarchies are then used to construct abstract macros by generalizing a particular plan as far as possible without losing the dependency structure of the plan. The resulting macros are then added to the operator hierarchy as new abstract operators which can be used to solve analogous problems in the future. The abstract operators, objects, and macros are then used for least-commitment hierarchical problem solving. PLANEREUS differs from ALPINE in at least two important ways. First, PLANEREUS generates operator and object hierarchies and macro operators, while ALPINE forms abstract problem spaces. Second, PLANEREUS forms abstract operators by ignoring the differences between operators without regard to the difficulty of achieving those differences, while ALPINE drops conditions based on an interaction and dependency analysis of the entire problem space.

ABSOLVER [Mostow and Prieditis, 1989] employs a set of abstraction transformations to create abstractions of a problem. The resulting abstract models are then tested to see if they provide useful abstractions for use in an admissible search procedure [Pearl, 1984, Kibler, 1985]. The reduced or relaxed models are used to compute lower bounds and check solvability. The abstraction transformations include dropping preconditions, dropping goals, and dropping predicates from the problem space. The abstractions are selected using a generate-and-test procedure. Since ALPINE employs a more principled approach in deciding which conditions to ignore, it is more likely to find useful abstractions and could be used to select the abstractions for ABSOLVER. However, the set of abstractions that could be generated by ALPINE may be more restrictive than needed for producing admissible search heuristics.

Hansson and Mayer [1989] use relaxed models to find intermediate subgoals to solve problems, such as the eight puzzle. They describe a system that drops conditions of the operators at random and the resulting relaxed models are then used to create intermediate subgoals. These subgoals are then used to simplify the original problem by searching to achieve each of the intermediate subgoals. Assuming that the abstract plan generated legal intermediate states, this approach reduces the search by replacing the original problem with a number of smaller ones. In contrast, ALPINE provides a more principled approach for determining which conditions should be dropped from a problem space, but ALPINE does not find a useful abstraction of the eight puzzle.

### 6.2.2 Aggregations

An alternative to constructing abstractions for problem solving is to construct aggregations. An aggregation is formed by combining the primitive elements of a problem space into larger elements. For example, sequences of operators can be combined to form macro operators and objects can be combined into aggregate objects. Aggregations provide a different type of abstraction that could be combined with the type of abstractions produced by ALPINE.

A number of systems have explored the formation of macro operators for problem solving [Fikes *et al.*, 1972, Korf, 1985b, Minton, 1985, Laird *et al.*, 1986, Shell and Carbonell, 1989, Iba, 1989, Riddle, 1990, Guvenir and Ernst, 1990]. A macro operator is constructed by combining frequently used sequences of operators into a single operator. In most systems, these macro operators are then added to the original space. Adding macros to the original problem space can reduce the solution depth, but it has the side-effect of increasing the branching factor, which can reduce the overall performance [Minton, 1985].

There are also systems that construct aggregate objects for problem solving. The idea is to reduce the complexity of problem solving by reasoning about larger-grained objects. Benjamin et al. [1990] present an approach to constructing aggregate objects by identifying equivalence classes over certain features in the state space. The approach is applied to the Tower of Hanoi puzzle to combine disks into macro objects. For example, the three-disk problem can be transformed into the two-disk problem by combining the two smallest disks into a single aggregate disk. This aggregation is equivalent to the abstraction of the Tower of Hanoi described in Section 3.4, but it is generated by a very different means.

There has also been work on automatically generating aggregations for chess. One system, called CHUNKER [Campbell, 1988], groups pawn configurations into chunks and then plans in the abstract space by reasoning about the interactions between pawn chunks. Another system, called PLACE [Flann, 1989], forms aggregate objects, operators, and goals using an explanation-based generalization approach.

### 6.2.3 Problem Reductions

In many systems, abstraction spaces are used to find an abstract solution, which can then be used to divide a problem into a number of subproblems. Problem reduction is a related technique, where a problem is replaced by a number of easier to solve subproblems [Amarel, 1968].

Riddle [1990] developed a system that automates this type of problem reformulation by analyzing example solutions and identifying the "critical" subgoals of a problem, which correspond to the most constrained subgoals. This approach success-

fully automates some of the reformulations of the Missionaries and Cannibals problem that were first described by Amarel [1968].

Ruby and Kibler [Ruby and Kibler, 1989] developed a system called SteppingStone that also learns sequences of subgoals. SteppingStone employs the learned subgoal sequences only when the basic means-ends analysis problem solver fails to find a solution that does not involve undoing a previously achieved goal. If there are no appropriate subgoals stored in memory, SteppingStone employs a brute-force search for a solution and uses the solution to learn a new sequence of subgoals.

Both of these systems learn sequences of subproblems through experience and then apply them to future problem solving episodes. Although the use of problem reductions avoids the search for an abstract solution, each problem might require a different set of problem reductions. In fact, hierarchical problem solving can be viewed as a dynamic method for generating problem reductions.

### 6.2.4 Goal Ordering

There has been a variety of work on ordering goals for problem solving. This section reviews only the work that employs techniques for goal ordering that are related to the abstraction generation techniques described in this thesis.

GPS [Ernst and Newell, 1969] is a means-ends analysis problem solver, which employs a table of differences to select relevant operators and thus focus the search. The problem solving proceeds by attempting to reduce the differences between the initial state and goal. The problem of finding good orderings of the differences has been extensively explored in GPS [Eavarone, 1969, Goldstein, 1978, Ernst and Goldstein, 1982]. The criterion for ordering the differences is to attempt to find an ordering such that achieving one difference will not affect a difference reduced by operators selected earlier in the ordering. This is related to the analysis performed by ALPINE, except the ordering of differences in GPS is based only on the effects of operators, while the construction of abstraction hierarchies in ALPINE is based on analysis of both the effects and preconditions of the operators. The constraints on a good difference ordering in GPS are necessary, but not sufficient for the ordered monotonicity property. For example, in the Tower of Hanoi the techniques for producing good difference orders for GPS is only able to identify the positions of the different sized disks as good differences, but cannot produce a useful ordering of the disks. [Eavarone, 1969] presents a program that produces 24 possible difference orderings for the four-disk problem without any preferences among them. In contrast, ALPINE produces a single hierarchy for the four-disk problem, which orders the disks from largest to smallest.

Irani and Cheng [Irani and Cheng, 1987, Cheng and Irani, 1989] present an approach to both ordering goals and augmenting the goals with additional information. The goal orderings are based on necessary interactions determined statically from the

operator definitions. For each problem the goal orderings are determined by back-propagating the goals through the operators to determine which of the other goals must already hold to apply the relevant operators. The goal conditions are first augmented with additional conditions that must also hold when the goal conditions are achieved. The augmented and ordered goals are then used in an admissible heuristic evaluation function. The augmentation of the goals is similar to the goal augmentation performed in ALPINE (Section 4.4.3), but the approach to ordering the goals is much more similar to the analysis in PABLO [Christensen, 1990]. In addition, the use of the goal orderings is more similar to the way abstractions are used in ABSOLVER [Mostow and Prieditis, 1989].

Etzioni [Etzioni, 1990] developed a system called STATIC, which statically analyzes the problem space definition to identify potential interactions. Based on these interactions, STATIC generates a set of search control rules for PRODIGY to guide the problem solving. The analysis is done by proving that a particular condition will necessarily interact with another condition and then constructing a control rule to avoid such an interaction. This analysis differs from the analysis performed by ALPINE, since the control rules are based on necessary interactions, while the abstractions are based on possible interactions. Also, the control rules are used to guide the search in the original space, while the abstractions are used for hierarchical problem solving.

## 6.3 Properties of Abstractions

Banerji and Ernst [1977a, 1977b] compared three similar problem solvers, GPS [Ernst, 1969], Planning GPS [Newell and Simon, 1972], and ABSTRIPS [Sacerdoti, 1974]. They developed a formal model of these systems that makes some additional assumptions not actually present in the three problem solvers. It is interesting to note that the additional assumptions correspond to enforcing the ordered monotonicity property. In the case of GPS, this means that after a given difference is solved, the problem solver is prevented from reintroducing that difference. In the case of ABSTRIPS, states are abstracted in the same way as the preconditions of operators, and when solving a problem at criticality level $i$, the problem solver rejects any problem with criticality level greater than $i$.

Using the formal models of these problem solvers, Banerji and Ernst then showed that all three of these systems can solve the same class of problems – those that are *well-stratified*. A well-stratified problem is one that, for a given abstraction hierarchy, can be divided up into subproblems and solved strictly in the order imposed by the hierarchy. They go on to show that if a problem is well-stratified, then it has a *totally ordered solution* [Ernst, 1969], which requires that once a difference is reduced it need not be reintroduced to solve the problem. Ernst [1969] showed that the combination of

a good difference ordering and the existence of a totally ordered solution are sufficient for GPS to solve a problem. Since the constraints on a good difference ordering are subsumed by the constraints on the ordered monotonicity property (Section 6.2.4), it follows that the ordered monotonicity and the existence of a totally ordered solution are sufficient for Hierarchical PRODIGY to solve a problem. This is not surprising since the restrictions on a problem that are needed to guarantee completeness (Section 3.2) are equivalent to requiring that a problem is well-stratified.

In Korf's work on generating macros [Korf, 1985b], he identified a property called *serial decomposability*, which is sufficient to guarantee that a set of macros can serialize a problem. A problem is said to be *serializable* if there exists an ordering among the goals, such that once a goal is satisfied, it need never be violated in order to satisfy the remaining goals. A problem space is serially decomposable if there exists an ordering of the operators such that the effect of each operator only depends on the state variables (e.g., location of a tile in the eight puzzle) that precede it in the ordering. If a problem space is serially decomposable, then there exists a set of macros that can make any problem serializable. Serializability is a property of goals, while the ordered monotonicity property is a property of an abstraction hierarchy. However, serializability is related to ordered monotonicity in that if a set of goals is serializable, then there exists a corresponding ordered monotonic abstraction hierarchy. But the ordered monotonicity is weaker than serializability since the converse does not hold. The ordered monotonicity property does not guarantee that once a goal is satisfied, it need never be violated. It only guarantees that once a goal is satisfied at one level, it will not be violated while refining a plan at a lower level, but it may be necessary to backtrack to the more abstract level if it cannot be refined.

# Chapter 7

# Conclusion

This thesis identified several useful properties of abstraction hierarchies, presented a completely automated approach to generating abstractions based on these properties, and described how the abstractions can be used for problem solving. The thesis showed analytically that under an ideal decomposition of a problem the use of hierarchical problem solving can produce an exponential-to-linear reduction in search, and it provided comprehensive empirical results which demonstrate that the generated abstractions produce better solutions with significantly less search in several different problem domains.

While the techniques are effective in generating useful abstractions for a variety of problem solving domains, they are not without their limitations. This chapter describes some of the limitations of both the theory and approach for generating abstractions, presents some ideas about how to produce better abstraction hierarchies automatically, and describes how the abstractions could be used for learning as well as planning. The chapter is divided into four sections. The first three sections describe the limitations and extensions of the theory, the generation of better abstractions, and the use of abstraction hierarchies in problem solving and learning. The last section concludes with a discussion of where this thesis leaves off and what remains to be done.

## 7.1 Theory of Abstraction

The theory presents two properties of abstraction hierarchies that relate a problem space to the possible abstractions of that problem space. The first property, monotonicity, captures the idea that the structure of the abstract solution should be preserved as it is refined. The second property, ordered monotonicity, is a restriction of the first and requires not only that the structure is preserved, but also that the space

is partitioned and ordered such that achieving the literals at one level does not interact with the literals in a more abstract level. The algorithm presented in this thesis generates abstraction hierarchies that satisfy the ordered monotonicity property.

While the ordered monotonicity property is quite general and captures a variety of interesting abstractions, it overconstrains the possible abstractions in some cases and underconstrains them in others. The property overconstrains the abstractions in the sense that there are useful abstractions that it does not capture. In general, the notion of preserving the structure of the abstract plan is important, but there are exceptions where a useful abstract plan may require violating the structure of the abstract plan in a constrained or localized fashion, where a condition is temporarily violated to achieve some other condition and then reachieved.

The ordered monotonicity property also underconstrains the possible abstraction hierarchies in that an abstraction hierarchy is not necessarily useful if it has this property. Because an abstract space is a simplification of the original problem space there may exist plans in that abstract space that are not *realizable*, which means that there is no way to refine the abstract plan to a plan in the original problem space. If the ratio of unrealizable to realizable abstract plans is too large, the use of a particular abstract space could prove to be more expensive than no abstraction at all. The problem arises because the properties on which the abstractions are based do not take into account the difficulty of achieving the conditions that are ignored. They only consider whether the achievement of the conditions can be delayed without interfering with those parts of the problem that have already been solved.

The properties described in the thesis provide only an initial approximation of what makes a good abstraction, although a formalizable and tractable approximation. A more comprehensive theory would need to deal with the problems mentioned above. To address the problem of overconstraining the hierarchies, the theory could be weakened such that an abstraction hierarchy does not need to be strictly monotonic, but nearly monotonic, where those conditions that could be easily reachieved are allowed to be undone when necessary to achieve some other conditions. To address the problem of underconstraining the hierarchies, the theory would need to consider not only whether the ordered monotonicity property could be ensured, but also the difficulty of achieving those conditions that are ignored. This could be dealt with empirically by maintaining statistics on the costs and benefits of each abstraction and eliminating those abstractions whose cost outweigh their benefit.

## 7.2 Generating Abstractions

ALPINE generates abstraction hierarchies that have the ordered monotonicity property. The algorithm used in ALPINE guarantees that any abstraction it finds will have

this property, but it does not guarantee that all ordered monotonic abstractions will be found. If ALPINE cannot find an abstraction then the directed graph of literals will collapse into a single strongly connected component. There are two limitations of the current approach that can prevent ALPINE from generating an abstraction for a given problem space and problem. First, the representation of the operators may limit the granularity of the abstractions. Second, the algorithm may generate constraints that are unnecessary to ensure the ordered monotonicity property.

### 7.2.1 Representing the Abstraction Hierarchies

The granularity of the abstraction hierarchies is determined by the language used to express the preconditions and effects of the operators. If an operator uses a literal with variables to express a precondition or effect, then ALPINE cannot place two instances of this literal at different levels in the hierarchy. The reason for this is that the algorithm determines the interactions between literals based on the preconditions and effects of the operators.

Consider how different representations of the Tower of Hanoi problem impose different constraints on the abstraction language. The completely instantiated representation, shown in Table 2.2, does not impose any constraints on the abstraction language (although the potential interactions of the preconditions and effects of operators still impose some constraints) because the operators are defined by fully-instantiated literals. In contrast, the representation consisting of one operator for moving each disk, shown in Table 2.1 in Chapter 2, constrains the literals for each different size disk to be in the same abstraction level. For example, `(on diskC peg1)`, `(on diskC peg2)`, and `(on diskC peg3)` are forced into the same abstraction level regardless of the interactions between these literals. This is because the operators have preconditions and effects such as `(on diskC` *peg*`)`, where *peg* is a variable, which prevents the system from distinguishing between different instances of the same literal. In this particular case, ALPINE would generate the same abstraction hierarchy for either representation.

Another possible representation of the Tower of Hanoi consists of a single operator for moving any disk. This operator is shown in Table 7.1. In the other two representations the conditions referring to different size disks were explicitly represented, so it was clear which disks would interact with which other disks. In this representation there is only the condition `(on` *disk peg*`)`, so the potential interactions are not made explicit in the operator representation. Instead the interactions of the different conditions are implicitly determined by the `smaller` relation. That is, moving a particular disk will only interact with smaller disks, but this is determined when the operator is matched during planning. Thus, the algorithm for generating abstractions does not find any abstractions given this representation of the problem.

```
(Move_Disk
  (preconds
    (and (is-peg source-peg)
         (is-peg dest-peg)
         (not (equal source-peg dest-peg))
         (on disk source-peg)
         (forall (smaller-disk)(smaller smaller-disk disk)
                 (and (not (on smaller-disk source-peg))
                      (not (on smaller-disk dest-peg))))))
  (effects ((del (on disk source-peg))
            (add (on disk dest-peg)))))
```

Table 7.1: Single Operator Version of the Tower of Hanoi

One way to avoid this problem is to partially evaluate the operators in order to determine the precise interactions for any given literal in a domain. Thus, instead of grouping literals together based on the granularity of the literals in the operators, each operator is partially evaluated to determine both the potential effects and potential preconditions when the operator is used to achieve various possible instantiated literals. To perform the partial evaluation the static conditions in the initial state are used to generate the bindings for the operator preconditions. For the single-disk Tower of Hanoi representation the `smaller`, `equal`, and `is-peg` relations would be used to partially evaluate the operator.

For example, consider how partial evaluation could be used to determine the potential interactions when the operator is used to achieve the literal `(on diskB peg3)`. Since this condition matches `(on` *disk dest-peg*`)` in the effects list, *disk* would be bound to `diskB` and *dest-peg* would be bound to `peg3`. Next, the static relations are used to determine the bindings for the other variables. The variable *source-peg* could be bound to `peg1` or `peg2`, and the variable *smaller-disk* could only be bound to `diskA`. Given the variable bindings it is then possible to determine the actual preconditions and effects when the operator is used to achieve a particular literal.

Once the potential interactions are determined for each literal in the domain, the basic algorithm for generating abstractions can be used to construct the abstraction hierarchy. The difference is that instead of determining the constraints simply by examining the operators, the constraints are determined by partially evaluating the operators.

This additional capability has been implemented in an extended version of ALPINE and it allows the system to produce finer-grained abstractions in many domains. For

example, in the process planning and scheduling domain, the system can produce abstraction spaces that distinguish between the various parts. Thus, the literals `(shape a cylindrical)` and `(shape b cylindrical)` could be placed at separate levels in the abstraction hierarchy. This allows the process planning for one part to be done separately from the process planning for another part because the different parts will not interact until they are placed in the schedule and the scheduling is done last. In the robot planning domain, partial evaluation allows ALPINE to place the literals involving different doors at separate abstraction levels. Thus, some doors can be treated as details while other doors are dealt with in more abstract spaces. Such a discrimination, for instance, is useful if the status of only some of the doors are mentioned in the goal state. The partial evaluation also allows ALPINE to generate abstractions for the single-operator version of the Tower of Hanoi.

The difficulty with abstracting instances of literals is that the complexity of the algorithm is dependent on the number of literal classes and this extension significantly increases the number of literal classes. One way to reduce the number of literal classes is to expand only some of the argument types in a domain. For example, expanding only the parts in the scheduling domains would allow different parts to be placed on separate levels. Another approach to control the number of literals is to determine which literals will actually be used in solving a particular problem and only reason about those literals.

### 7.2.2 Constraints on the Abstraction Hierarchy

The most difficult problem of generating the abstraction hierarchies is finding a set of constraints that are sufficient to guarantee the ordered monotonicity problem, but do not overconstrain the possible abstraction hierarchies. ALPINE attempts to identify only those interactions that could actually occur in solving the given problem. However, since it forms the abstractions by statically analyzing the operators it must make assumptions about which operators could be used and in what context. Thus, the abstraction hierarchies are based on the possible interactions, which are a superset of the actual interactions. As a result it will in many cases overconstrain the hierarchy, thus reducing the granularity of the possible abstraction hierarchies.

The "blocks world" [Nilsson, 1980] is a domain in which ALPINE is unable to generate abstractions, although there are ordered monotonic abstractions for some problems. For example, given the problem of building a stack of blocks with `A` on `B`, `B` on `C`, and `C` on the table, an ordered monotonic abstraction hierarchy would deal with the conditions on each block in the opposite order. For this example, the abstraction hierarchy would contain three levels, with `C` in the most abstract level, `B` and `C` on the next level, and all three blocks in the final level. Thus, the problem would be solved by first getting the bottom block on the table, next stacking the block above

that one, finally placing the last block on the top of the stack. This abstraction hierarchy has the ordered monotonicity property because as the plan is refined it will never be necessary to undo any of the conditions involving a block in a more abstract space. However, ALPINE cannot generate this abstraction because simply analyzing the possible interactions of the operators, it appears that every condition will interact with all other conditions.

In order to find more subtle abstractions, such as the one in the blocks world, the system needs a deeper understanding of which conditions will actually interact with which other conditions in practice. One approach to solving this problem is to use explanation-based learning to acquire the necessary constraints by example. The system could begin with no constraints on the abstraction hierarchy and then learn a set of constraints through experience. The problem solver can easily detect a violation of the ordered monotonicity property, which occurs anytime an operator is applied at one level that changes a condition in a more abstract level. When a violation is detected the problem solver halts and invokes the EBL system to explain why the violation occurred. From the proof of the violation the system constructs a rule that constrains some literal to be placed before some other literal in the abstraction hierarchy whenever the conditions arise under which the violation occurs. The rules learned by the EBL system would then be used to constrain the selection of the abstraction hierarchy for the given problem as well as future problems in the same domain. The resulting constraints on the abstraction hierarchy would be necessary, but not sufficient to guarantee the ordered monotonicity property.

There are two potential difficulties with this approach. First, despite the generalization of the constraint rules, the number of rules that would need to be learned to cover a domain could be quite large. Thus, it could be expensive both to learn the rules and to apply the rules to select an abstraction hierarchy for a particular problem. Second, the number of levels in the resulting abstraction hierarchy could be large, which would make it expensive to use the hierarchies for problem solving.

## 7.3 Using Abstractions

This thesis presented one approach to using abstractions for hierarchical problem solving. While this particular approach produces significant reductions in search, it is by no means the only possible use of the abstractions. Since the abstractions are abstract versions of the original problem space, they are not specific to the particular problem-solving method. The abstract problem spaces could be used for other approaches to hierarchical problem solving or exploited in other ways. For example, operators and objects that are indistinguishable in an abstract space can be merged to simplify a problem space. Also, the abstract problem spaces could be combined

with other learning methods. This section outlines how these abstractions could be used for other types of problem solving, describes how operators and objects can be combined, and sketches how the abstract problem spaces could be used for learning.

### 7.3.1 Problem Solving

ALPINE takes an initial problem space and forms abstract problem spaces that could then be used for a variety of problem-solving techniques. This section describe how the abstractions generated by ALPINE could be used for both least-commitment problem solving and forward-chaining problem solving.

#### Least-Commmitment Hierarchical Problem Solving

The model of hierarchical problem solving described in Chapter 3 is based on a state-space problem solver. An alternative is to use a least-commitment approach to problem solving [Sacerdoti, 1977, Chapman, 1987], which searches through the space of plan refinements instead of searching through the state space. This approach is referred to as a least-commitment approach because ordering commitments are delayed as long as possible.

The use of the abstraction hierarchies generated by ALPINE would be a simple extension to a least-commitment problem solver. First, a problem would be solved in the most abstract space to produce a partially ordered plan. The plan would then be refined in successive abstraction spaces by considering the conditions introduced at that level and adding the necessary plan steps and ordering constraints to produce a valid plan. The use of the abstractions provides additional information on which parts of the problem to solve first, but the basic problem-solving method remains unchanged. The ordered monotonicity property provides the same advantages in this approach as it does in the state-space approach. That is, the abstract solution produced at each level provides the outline for the final solution and would not be changed in the refinement process, thus constraining the search for a refinement at each level of abstraction.

#### Forward-Chaining Problem Solving

The problem solving method presented in this thesis assumed that the goals introduced at each abstraction level will be achieved by chaining backward from the goal. However the same abstractions could also be used in a forward-chaining problem solver, such as soar [Laird *et al.*, 1987]. The only problem that arises with a forward-chaining system is that operators from more abstract levels might be applied at levels in which they should not be considered and potentially violate the ordered monotonicity property. However, this problem can be avoided simply by not allowing any

operator that occurs at a higher abstraction level to be inserted to achieve goals that arise in the refinement process. Because a backward-chaining problem solver is more goal directed and the ordered monotonicity property guarantees that the more abstract goals never arise at the lower levels, this problem never arises in a backward-chaining system.

Consider an example from the Tower of Hanoi. In the abstract space a plan is constructed for moving the largest disk. At the next level this plan is refined to also achieve the conditions involving the medium-sized disk. Using a backward-chaining system, none of the operators for moving the large disk would even be considered since the ordered monotonicity property guarantees that goals involving the large disk will not arise at this level. However, with a forward-chaining system, an operator for moving a large disk could be inserted simply because the preconditions are met. However, if the forward-chaining problem solver prevents any of the operators from the more abstract space from being applied, then the problem solver would only consider operators for moving the medium-sized disk. Thus, a forward-chaining system can use the abstraction spaces produced by ALPINE and still preserve the ordered monotonicity property.

### 7.3.2 Operator and Object Hierarchies

An advantage of forming reduced models of a problem space is that when details of a problem space are removed, it may be possible to combine operators and objects to form operator and object hierarchies. Each abstract operator or object represents an equivalence class whose members are distinguishable only at lower levels of abstraction. Two operators may differ by some detail in the original problem space, but in an abstract space, the two may be indistinguishable. If so, they can then be combined into a single abstract operator that will be refined into a concrete operator at the level in which the two are distinguishable. Similarly, objects may become indistinguishable in an abstract space, and they can be combined into an abstract object that can then be treated as a resource. The use of both operator and object hierarchies can reduce the branching factor in the abstract space since there will be fewer operators and/or fewer instantiations of operators to consider during problem solving.

Consider an example in the scheduling domain, where there are two machines that can be used to make a part cylindrical, the lathe and the milling machine. While these operators differ in some of their preconditions and effects, if these differences are ignored in an abstract space, then the commitment to a particular machine can be delayed. In the abstract problem space the two operators would be replaced by a single abstract operator. Thus a plan produced in the abstract space would only contain this abstract operator. When this plan is refined into a level where the lathe and mill operators differ, the abstract operator would then be replaced by one of the

more specific operators. At that time there may be additional knowledge to select one machine over the other (for instance, the milling machine may already be in use).

This abstract operator could be used in the example described in Section 2.3. In that example, the problem was to make a part cylindrical and polished. The abstraction hierarchy formed for this problem deals with the cylindrical goal first and then the polished goal. If the mill and lathe operators are not combined into a single abstract operator, the problem solver will be forced to arbitrarily select one of the specific operators in the abstract space and it may select one that cannot be refined. In this particular example if the mill operator is used the problem solver will find that the plan cannot be refined because the part will be too hot to polish and it will eventually be forced to backtrack to the abstract space and select the lathe operator. With a single abstract operator for making a part cylindrical, the system would create the abstract plan to make the part cylindrical, then when the plan is refined the abstract operation would be refined into either the lathe or mill operation. By delaying the commitment to the more specific operator, the choice point will be moved closer to the potential interaction, which will reduce or eliminate the backtracking. (This is the same idea used in least-commitment problem solver, where ordering commitments are delayed as long as possible.)

Object hierarchies can be used in an analogous way. If two or more objects are indistinguishable in an abstract space, they can be treated as a resource. Thus, instead of committing to a specific object, the abstract plan can simply refer to the resource. Then when the plan is refined to the level in which the objects are distinguishable, the resource would be replaced by one of the particular objects. The advantages of object hierarchies are similar to operator hierarchies in that they delay committing to a particular choice as long as possible and can thus help reduce or avoid backtracking.

The hierarchical version of PRODIGY could easily be extended to handle operator hierarchies. This would simply involve combining two or more operators that are indistinguishable in an abstract space into a single abstract operator and then replacing the abstract operator with a concrete one as the abstract plan is refined. The use of object hierarchies is a bit more complex because objects are often a limited resource. To exploit object hierarchies requires the capability of reasoning about resources. Such a capability is provided in SIPE, as described in [Wilkins, 1988], but is not yet available in PRODIGY.

### 7.3.3 Using Abstract Problem Spaces for Learning

Since the abstractions of a problem space are abstract problem spaces, the abstractions can be used for learning as well as problem solving. This section sketches approaches to combining the abstractions generated by ALPINE with both explanation-

based learning and learning by analogy.[1]

### Explanation-Based Learning

Explanation-based learning is used in PRODIGY to learn control knowledge to guide the search [Minton, 1988a]. The control knowledge learned by EBL in PRODIGY provides significant reductions in search. However, a difficulty with this approach is that the examples from which the system learns often contain an abundance of unnecessary details. In order to learn control rules, the EBL system constructs proofs about the success, failure, or interactions in a problem-solving example. Problems with lots of details make this process more complex because the proofs are considerably more complex. As a result of the details in the proofs, the EBL system may also learn control rules that are overly specific. Because the rules are more specific, it requires more rules to learn a sufficient set of control knowledge to solve problems efficiently in a given domain. The more rules in the system, the more time it will spend matching the rules, reducing the overall benefit of the control knowledge.

One possible approach to combining explanation-based learning and abstraction is to apply the control rule learning within each abstraction space. This would simplify the learning process and result in more general control rules since the proofs in an abstract space would contain fewer details. The fewer, more general rules would be cheaper to match and thus provide better performance.

To illustrate the synergistic effect of ALPINE and EBL, consider their integration in the Tower of Hanoi domain. As described in Section 5.1, the use of abstraction in the Tower of Hanoi provides a significant reduction in search, but using a depth-first search it still produces suboptimal solutions and requires some search. EBL can be applied to the Tower of Hanoi to learn control rules to reduce search. For the two-disk problem, the EBL system produces a set of control rules such that the system makes the correct decision at each choice point and produces the optimal solution. However, because the proofs become more complex as the problems get larger, the system does not produce a complete set of rules for anything larger than the two-disk problem.

Combining abstraction and EBL in the Tower of Hanoi reduces the problem to one that can be solved without search (i.e., the correct decision is made at every choice point). The abstraction simplifies the problem such that the only search involves moving a disk out of the way of another disk that needs to be moved. If a disk needs to be moved in order to move another disk, there are only two places to move the disk, one of which is the "right" place and the other will interfere with the placement of another disk. EBL is particularly good at recognizing this type of interaction, called a

[1] ALPINE could also be combined with STATIC [Etzioni, 1990], which performs static analysis of a problem space to produce control knowledge. The integration of ALPINE and STATIC would be analogous to combining ALPINE with EBL.

*prerequisite violation*, and learning control rules to avoid them. Thus, the EBL system can learn a set of rules that allow the problem solving to make the correct choices at each level in the hierarchy.

An example rule that was learned by the EBL system in an abstract problem space is shown in Table 7.2. This rule states that if the goal is to get `diskA` out of the way of moving `diskB`, the problem solver should move `diskA` someplace other than the place where it is planning to move `diskB`. Otherwise, `diskA` will immediately need to be moved again. While this rule is more specific than necessary and will require learning a set of these rules to cover all the cases, the EBL system can learn more general rules if the operators are parameterized.

```
(if (and (current-node node)
         (current-goal node (noton diskA peg1))
         (candidate-op node move-disk-A-peg-1-2)
         (alt-on-deck node (on diskB peg2) move-disk-B-peg-1-2)
         (candidate-op node op)
         (not-equal move-disk-A-peg-1-2 op)))
(then (prefer operator op move-disk-A-peg-1-2))
```

Table 7.2: Control Rule Learned by EBL in an Abstract Space

The combination of the two techniques produces performance improvements that neither system can achieve independently [Knoblock *et al.*, 1991a]. In the Tower of Hanoi the abstraction module can reduce the search from exponential-to-linear in the solution length, but it cannot completely eliminate the search within each abstraction level. The EBL module can learn rules for the simple Tower of Hanoi problems, but it is unable to learn a set of rules that completely solves problems with more than two disks. However, the combination of the two approaches can both eliminate any search from the problem and produce the optimal solution.

### Learning by Analogy

Analogy can also used to guide problem solving in PRODIGY [Veloso and Carbonell, 1990]. The analogy engine stores problem solving episodes in a case library and then retrieves them to guide the search in similar problems. There are several difficulties that arise in the use of analogy in problem solving. First, the analogy engine can get mired down in indexing and selecting the relevant stored plans. Second, the number of stored plans can become quite large, incurring significant storage and retrieval costs.

Similar to combining abstraction and EBL, abstraction and analogy can be integrated by applying analogy in the abstract problem spaces. This integration will simplify the indexing of new problems to previously solved problems since the abstraction spaces will separate the important aspects from the details. Since analogy is employed in simpler abstract problem spaces, it will store a smaller set of more general past solutions and will thus reduce the storage and retrieval costs.

Consider the integration of abstraction and analogy in the Tower of Hanoi. As described previously, the use of abstraction partitions the problem such that each abstraction space requires inserting the steps to move a particular disk. Analogy would be used to store the plans for moving the disks at each level in the abstraction hierarchy. Then, instead of searching for a solution to a subproblem, the analogy system would retrieve a similar previously solved problem and use that to guide the search. The integration simplifies the indexing and retrieval since the cases will be partitioned by the abstraction levels. Thus, the number of possible plans will be much smaller and it will be easier to find one that is relevant to a given problem.

## 7.4 Discussion

The construction of abstract problem spaces is a type of reformulation, where the original problem space is replaced by a more abstract one. The work presented in this thesis takes the first steps towards automatically reformulating problems for problem solving. The role of reformulation has long been recognized as central to problem solving [Amarel, 1968, Korf, 1980, Hobbs, 1985, Subramanian and Genesereth, 1987, Subramanian, 1989], but much of this work has focused on identifying and representing the reformulations. As described in Section 6.2, more recently has work begun to address the problem of how to automate these processes.

The key to solving a problem is understanding the problem. Some problem solvers forge ahead blindly hoping to stumble across a solution by focusing on one part of the problem and when that has been achieved focusing on another part. Other problem solvers interleave the work on the various parts of a problem but spend an excessive amount of time delaying commitments and verifying constraints. A better approach is to step back and understand a problem. What are the hard parts? What are the details? How can the problem be decomposed? This thesis presented an approach to do exactly that. It takes a problem and based on the problem reformulates the initial problem space into a hierarchy of abstract problem spaces that can then be used to solve the problem. This allows the problem solver to focus on the difficult parts first, decomposing the problem into simpler subproblems and gradually reintroducing the details that were ignored.

In general, reformulating problems and solving them "intelligently" requires much

more knowledge than is usually provided to problem solvers. When people attack a problem they bring a vast amount of knowledge to bear on the problem. When computers solve problems they are limited by inflexible methods and very shallow theories of the problem solving domains. Consider the mutilated checkerboard problem [McCarthy, 1964], where the problem is to cover a mutilated checkerboard, which has two opposite corners removed, with a set of dominoes (each covering two squares) or prove that the problem is impossible. In turns out that the problem is impossible since there will be two fewer squares of one color than the other color and each domino can only cover one black and one white square. To solve this problem does not require search in either the state space or the plan space, but search through the space of possible problem spaces [Kaplan and Simon, 1990]. It requires changing the problem space from one in which all possible arrangements of the dominoes on the board are considered to one that uses the parity of the squares on the board to show that it would be futile to even begin to arrange the dominoes.

While the particular reformulation of the mutilated checkerboard problem may not have very general applicability, it does illustrate the approach needed to solve more difficult problems. That is, a problem solver should be able to take a problem represented at some level of detail and reformulate it into a problem that captures the "essence" of the problem. An important step in this process is determining which conditions to focus on and which conditions to ignore. However, to build an intelligent problem solver will require more than the ability to ignore some of the details. It will also require the ability to reformulate a problem into a completely different representation of a problem. This thesis has achieved that first step of focusing problem-solving attention on the most relevant and difficult aspects first, and then progressively reintroducing more peripheral information to construct a complete solution to a problem.

# Appendix A

# Tower of Hanoi

This section includes the PRODIGY code for the Tower of Hanoi and provides the experimental results described in Chapter 5. The three representations of the Tower of Hanoi that are described in the thesis are presented in this section. An example problem is included for each of the three problem-space representations.

## A.1 Single-Operator Representation

```
(MOVE-DISK
  (params (<disk> <peg.from> <peg.to>))
  (preconds
    (and (is-peg <peg.from>)
         (is-peg <peg.to>)
         (not-equal <peg.from> <peg.to>)
         (on <disk> <peg.from>)
         (forall (<disk.sm>)(smaller <disk.sm> <disk>)
                 (and (~ (on <disk.sm> <peg.from>))
                      (~ (on <disk.sm> <peg.to>)))))))
  (effects ((del (on <disk> <peg.from>))
            (add (on <disk> <peg.to>)))))

Goal: '(and (on diskA peg3)(on diskB peg3)(on diskC peg3))

Initial State: '((on diskA peg1)(on diskB peg1)(on diskC peg1)
                 (smaller diskA diskB)
                 (smaller diskA diskC)
                 (smaller diskB diskC)
                 (is-peg peg1)(is-peg peg2)(is-peg peg3)
                 (is-disk diskC)(is-disk diskB)(is-disk diskA)))
```

## A.2 Instantiated-Disk Representation

```
(MOVE-DISK-A
  (params (<peg.from> <peg.to>))
  (preconds
    (and (on diskA <peg.from>)
         (is-peg <peg.to>)
         (not-equal <peg.from> <peg.to>)))
  (effects ((del (on diskA <peg.from>))
            (add (on diskA <peg.to>)))))

(MOVE-DISK-B
  (params (<peg.from> <peg.to>))
  (preconds
    (and (on diskB <peg.from>)
         (is-peg <peg.to>)
         (not-equal <peg.from> <peg.to>)
         (~ (on diskA <peg.from>))
         (~ (on diskA <peg.to>))))
  (effects ((del (on diskB <peg.from>))
            (add (on diskB <peg.to>)))))

(MOVE-DISK-C
  (params (<peg.from> <peg.to>))
  (preconds
    (and (on diskC <peg.from>)
         (is-peg <peg.to>)
         (not-equal <peg.from> <peg.to>)
         (~ (on diskB <peg.from>))
         (~ (on diskA <peg.from>))
         (~ (on diskB <peg.to>))
         (~ (on diskA <peg.to>))))
  (effects ((del (on diskC <peg.from>))
            (add (on diskC <peg.to>)))))

Goal: '(and (on diskA peg3)(on diskB peg3)(on diskC peg3))

Initial State: '((on diskC peg1)(on diskB peg1)(on diskA peg1)
                 (is-peg peg1)(is-peg peg2)(is-peg peg3))
```

## A.3 Fully-Instantiated Representation

```
(MOVE-DISK-A-PEG-1-2
  (preconds (on diskA peg1))
  (effects ((del (on diskA peg1))
            (add (on diskA peg2)))))
```

```
(MOVE-DISK-A-PEG-2-1
  (preconds (on diskA peg2))
  (effects ((del (on diskA peg2))
            (add (on diskA peg1)))))

(MOVE-DISK-A-PEG-1-3
  (preconds (on diskA peg1))
  (effects ((del (on diskA peg1))
            (add (on diskA peg3)))))

(MOVE-DISK-A-PEG-3-1
  (preconds (on diskA peg3))
  (effects ((del (on diskA peg3))
            (add (on diskA peg1)))))

(MOVE-DISK-A-PEG-2-3
  (preconds (on diskA peg2))
  (effects ((del (on diskA peg2))
            (add (on diskA peg3)))))

(MOVE-DISK-A-PEG-3-2
  (preconds (on diskA peg3))
  (effects ((del (on diskA peg3))
            (add (on diskA peg2)))))

(MOVE-DISK-B-PEG-1-2
  (preconds
    (and (on diskB peg1)
         (~ (on diskA peg1))
         (~ (on diskA peg2))))
  (effects ((del (on diskB peg1))
            (add (on diskB peg2)))))

(MOVE-DISK-B-PEG-2-1
  (preconds
    (and (on diskB peg2)
         (~ (on diskA peg2))
         (~ (on diskA peg1))))
  (effects ((del (on diskB peg2))
            (add (on diskB peg1)))))

(MOVE-DISK-B-PEG-1-3
  (preconds
    (and (on diskB peg1)
         (~ (on diskA peg1))
         (~ (on diskA peg3))))
  (effects ((del (on diskB peg1))
            (add (on diskB peg3)))))

(MOVE-DISK-B-PEG-3-1
  (preconds
    (and (on diskB peg3)
         (~ (on diskA peg3))
         (~ (on diskA peg1))))
  (effects ((del (on diskB peg3))
            (add (on diskB peg1)))))

(MOVE-DISK-B-PEG-2-3
  (preconds
    (and (on diskB peg2)
         (~ (on diskA peg2))
         (~ (on diskA peg3))))
  (effects ((del (on diskB peg2))
            (add (on diskB peg3)))))

(MOVE-DISK-B-PEG-3-2
  (preconds
    (and (on diskB peg3)
         (~ (on diskA peg3))
         (~ (on diskA peg2))))
  (effects ((del (on diskB peg3))
            (add (on diskB peg2)))))

(MOVE-DISK-C-PEG-1-2
  (preconds
    (and (on diskC peg1)
         (~ (on diskB peg1))
         (~ (on diskA peg1))
         (~ (on diskB peg2))
         (~ (on diskA peg2))))
  (effects ((del (on diskC peg1))
            (add (on diskC peg2)))))

(MOVE-DISK-C-PEG-2-1
  (preconds
    (and (on diskC peg2)
         (~ (on diskB peg2))
         (~ (on diskA peg2))
         (~ (on diskB peg1))
         (~ (on diskA peg1))))
  (effects ((del (on diskC peg2))
            (add (on diskC peg1)))))
```

```
(MOVE-DISK-C-PEG-1-3
  (preconds
    (and (on diskC peg1)
         (~ (on diskB peg1))
         (~ (on diskA peg1))
         (~ (on diskB peg3))
         (~ (on diskA peg3))))
  (effects ((del (on diskC peg1))
            (add (on diskC peg3)))))

(MOVE-DISK-C-PEG-3-1
  (preconds
    (and (on diskC peg3)
         (~ (on diskB peg3))
         (~ (on diskA peg3))
         (~ (on diskB peg1))
         (~ (on diskA peg1))))
  (effects ((del (on diskC peg3))
            (add (on diskC peg1)))))

(MOVE-DISK-C-PEG-2-3
  (preconds
    (and (on diskC peg2)
         (~ (on diskB peg2))
         (~ (on diskA peg2))
         (~ (on diskB peg3))
         (~ (on diskA peg3))))
  (effects ((del (on diskC peg2))
            (add (on diskC peg3)))))

(MOVE-DISK-C-PEG-3-2
  (preconds
    (and (on diskC peg3)
         (~ (on diskB peg3))
         (~ (on diskA peg3))
         (~ (on diskB peg2))
         (~ (on diskA peg2))))
  (effects ((del (on diskC peg3))
            (add (on diskC peg2)))))

Goal: (and (on diskA peg3)
           (on diskB peg3)
           (on diskC peg3))

Initial State: ((on diskC peg1)
                (on diskB peg1)
                (on diskA peg1))
```

## A.4 Experimental Results

The Tower of Hanoi experiments were run in Allegro Common Lisp on a SparcStation 1+ with 12 megabytes of memory. These experiments used the single-operator representation of the Tower of Hanoi, but since the problem spaces are equivalent, the numbers would be roughly the same for any of the representations.

The tables below compare PRODIGY both with and without using the abstractions produced by ALPINE. The entries in the table are defined as follows:

**Disks** The number of disks in the problem.

**Time** Total CPU time used in solving the problem. A 600 CPU second time bound was imposed on all problems.

**Nodes** Total number of nodes searched in solving the problem.

**Len** Length of the solution found. Zero if no solution exists.

**ACT** Time required to create the abstraction hierarchy. This time is also included in the total CPU time for ALPINE.

**AbNodes** Nodes searched at each level in the hierarchy. Ordered from more abstract to less abstract levels.

**AbLen** Solution length found at each level in the hierarchy. Ordered from more abstract to less abstract levels.

### Depth-First Iterative-Deepening Search

| | Prodigy | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Disks | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 0.1 | 4 | 1 | 0.2 | 4 | 1 | 0.1 | 4 | 1 |
| 2 | 1.8 | 122 | 3 | 0.5 | 12 | 3 | 0.2 | 4,8 | 1,2 |
| 3 | 47.0 | 2790 | 7 | 0.8 | 24 | 7 | 0.3 | 4,8,12 | 1,2,4 |
| 4 | 600.0 | — | — | 2.1 | 52 | 15 | 0.5 | 4,8,12,28 | 1,2,4,8 |
| 5 | 600.0 | — | — | 3.8 | 96 | 31 | 0.8 | 4,8,12,28,44 | 1,2,4,8,16 |
| 6 | 600.0 | — | — | 8.4 | 204 | 63 | 1.3 | 4,8,12,28,44,108 | 1,2,4,8,16,32 |
| 7 | 600.0 | — | — | 16.4 | 376 | 127 | 1.6 | 4,8,12,28,44,108,172 | 1,2,4,8,16,32,64 |

## Depth-First Search

| | Prodigy | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Disks | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 0.0 | 4 | 1 | 0.1 | 4 | 1 | 0.1 | 4 | 1 |
| 2 | 0.4 | 15 | 6 | 0.5 | 12 | 4 | 0.2 | 4,8 | 1,3 |
| 3 | 1.1 | 49 | 21 | 1.1 | 34 | 13 | 0.3 | 4,8,22 | 1,3,9 |
| 4 | 4.2 | 147 | 64 | 3.5 | 96 | 40 | 0.5 | 4,8,22,62 | 1,3,9,27 |
| 5 | 13.0 | 424 | 185 | 10.2 | 280 | 121 | 0.8 | 4,8,22,62,184 | 1,3,9,27,81 |
| 6 | 51.3 | 1202 | 524 | 36.1 | 828 | 364 | 1.3 | 4,8,22,62,184,548 | 1,3,9,27,81,243 |
| 7 | 233.1 | 3395 | 1477 | 168.2 | 2470 | 1093 | 1.6 | 4,8,22,62,184,548,1642 | 1,3,9,27,81,243,729 |

## Depth-First Search on a Variant of the Tower of Hanoi

| | Prodigy | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Disks | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 0.1 | 4 | 1 | 0.3 | 4 | 1 | 0.1 | 4 | 1 |
| 2 | 1.4 | 41 | 3 | 0.4 | 12 | 3 | 0.2 | 4,8 | 1,2 |
| 3 | 1.6 | 52 | 7 | 1.1 | 24 | 7 | 0.3 | 4,8,12 | 1,2,4 |
| 4 | 3.6 | 116 | 19 | 2.7 | 52 | 15 | 0.5 | 4,8,12,28 | 1,2,4,8 |
| 5 | 9.1 | 254 | 51 | 5.2 | 96 | 31 | 0.8 | 4,8,12,28,44 | 1,2,4,8,16 |
| 6 | 27.5 | 695 | 131 | 10.7 | 204 | 63 | 1.3 | 4,8,12,28,44,108 | 1,2,4,8,16,32 |
| 7 | 92.4 | 1935 | 323 | 22.0 | 376 | 127 | 1.6 | 4,8,12,28,44,108,172 | 1,2,4,8,16,32,64 |

This variant of the Tower of Hanoi disallows moving the same disk twice in a row. This is implemented using the following PRODIGY control rule:

```
(DO-NOT-MOVE-TWICE
 (lhs (and (current-node <node>)
   (current-op <node> MOVE-DISK)
   (candidate-bindings <node> (<disk> <peg1> <peg2>))
   (last-disk-moved <node> <last-disk>)
   (is-equal <last-disk> <disk>)))
 (rhs (reject bindings (<disk> <peg1> <peg2>))))
```

# Appendix B

# Extended STRIPS Domain

The robot planning domain described in this section is equivalent to the domain described by Minton [1988a], but there are three minor syntactic changes that were made to improve the abstractions produced by ALPINE. These changes are as follows:

- The original domain treated both boxes and keys simply as objects and made no explicit distinction between them. This domain uses a type hierarchy to distinguish between them, but since the operator language is restricted to refer to conditions on the leaves of the type hierarchy (Section 4.4.1), each operator for manipulating objects is divided into two operators, one for boxes and one for keys.

- The original domain used the static conditions `dr-to-room` and `connects` to express the relationships between the rooms and doors. The revised version simply uses `connects` uniformly. This is done to simplify the analysis of the relationships between variables in different operators.

- There were also some subtle precondition ordering problems that prevented the problem solver from finding solutions to some problems that have solutions. To avoid these problems, preconditions that require `holding` an object were moved after the preconditions that require getting the same object into a particular room, and preconditions that require `arm-empty` were moved after `door-open` preconditions.

## B.1 Problem Space Definition

```
(PICKUP-BOX
 (params (<box.o1>))
 (preconds
   (and (arm-empty)
        (next-to robot <box.o1>)
        (carriable <box.o1>)))
 (effects
   ((del (arm-empty))
    (del (next-to <box.o1> <box.*30>))
    (del (next-to <box.o1> <door.*30>))
    (del (next-to <box.*31> <box.o1>))
    (del (next-to robot <box.o1>))
    (add (holding <box.o1>)))))

(PICKUP-KEY
 (params (<key.o1>))
 (preconds
   (and (arm-empty)
        (next-to robot <key.o1>)
        (carriable <key.o1>)))
 (effects
   ((del (arm-empty))
    (del (next-to robot <key.o1>))
    (add (holding <key.o1>)))))

(PUTDOWN-BOX
 (params (<box.o2>))
 (preconds
   (and (holding <box.o2>)
        (is-box <box.o2>)))
 (effects
   ((del (holding <box.*35>))
    (add (next-to robot <box.o2>))
    (add (arm-empty)))))

(PUTDOWN-KEY
 (params (<key.o2>))
 (preconds
   (and (holding <key.o2>)
        (is-key <door.o2> <key.o2>)))
 (effects
   ((del (holding <key.*35>))
    (add (next-to robot <key.o2>))
    (add (arm-empty)))))

(PUTDOWN-BOX-NEXT-TO
 (params (<box.o3> <box.other>
          <room.o3-rm>))
 (preconds
   (and
     (is-object <box.other>)
     (inroom <box.other> <room.o3-rm>)
     (inroom <box.o3> <room.o3-rm>)
     (holding <box.o3>)
     (next-to robot <box.other>)))
 (effects
   ((del (holding <box.o3>))
    (add (next-to <box.o3> <box.other>))
    (add (next-to robot <box.o3>))
    (add (next-to <box.other> <box.o3>))
    (add (arm-empty)))))

(PUSH-BOX-TO-DR
 (params (<box.b1> <door.d1> <room.r1>))
 (preconds
   (and
     (is-door <door.d1>)
     (connects <door.d1> <room.r2>
                         <room.r1>)
     (inroom <box.b1> <room.r1>)
     (next-to robot <box.b1>)
     (pushable <box.b1>)))
 (effects
   ((del (next-to <box.b1> <box.*5>))
    (del (next-to <box.b1> <door.*5>))
    (del (next-to <box.*13> <box.b1>))
    (del (next-to robot <box.*3>))
    (add (next-to robot <box.b1>))
    (add (next-to <box.b1> <door.d1>))
    )))

(PUSH-BOX-THRU-DR
 (params (<box.b-x> <door.d-x>
          <room.r-x> <room.r-y>))
 (preconds
   (and
     (is-room <room.r-x>)
     (connects <door.d-x> <room.r-x>
                          <room.r-y>)
     (is-door <door.d-x>)
     (dr-open <door.d-x>)
     (next-to <box.b-x> <door.d-x>)
     (next-to robot <box.b-x>)
     (pushable <box.b-x>)
     (inroom <box.b-x> <room.r-y>)))
 (effects
```

```
   ((del (next-to robot <box.*1>))
    (del (next-to <box.b-x> <box.*12>))
    (del (next-to <box.b-x> <door.*12>))
    (del (next-to <box.*7> <box.b-x>))
    (del (inroom robot <room.*21>))
    (del (inroom <box.b-x> <room.*22>))
    (add (inroom robot <room.r-x>))
    (add (inroom <box.b-x> <room.r-x>))
    (add (next-to robot <box.b-x>)))))

(GO-THRU-DR
 (params (<door.ddx> <room.rrx>
          <room.rry>))
 (preconds
   (and
     (is-room <room.rrx>)
     (connects <door.ddx> <room.rrx>
                          <room.rry>)
     (is-door <door.ddx>)
     (dr-open <door.ddx>)
     (arm-empty)
     (next-to robot <door.ddx>)
     (inroom robot <room.rry>)))
 (effects
   ((del (next-to robot <door.*19>))
    (del (inroom robot <room.*20>))
    (add (inroom robot <room.rrx>)))))

(CARRY-BOX-THRU-DR
 (params (<box.b-zz> <door.d-zz>
          <room.r-zz> <room.r-ww>))
 (preconds
   (and
     (is-room <room.r-zz>)
     (connects <door.d-zz> <room.r-zz>
                           <room.r-ww>)
     (is-door <door.d-zz>)
     (dr-open <door.d-zz>)
     (is-object <box.b-zz>)
     (inroom <box.b-zz> <room.r-ww>)
     (carriable <box.b-zz>)
     (holding <box.b-zz>)
     (inroom robot <room.r-ww>)
     (next-to robot <door.d-zz>)))
 (effects
   ((del (next-to robot <door.*48>))
    (del (inroom robot <room.*41>))
    (del (inroom <box.b-zz> <room.*42>))
    (add (inroom robot <room.r-zz>))
    (add (inroom <box.b-zz> <room.r-zz>)
    ))))

(CARRY-KEY-THRU-DR
 (params (<key.b-zz> <door.d-zz>
          <room.r-zz> <room.r-ww>))
 (preconds
   (and
     (is-room <room.r-zz>)
     (connects <door.d-zz> <room.r-zz>
                           <room.r-ww>)
     (is-door <door.d-zz>)
     (dr-open <door.d-zz>)
     (is-object <key.b-zz>)
     (inroom <key.b-zz> <room.r-ww>)
     (carriable <key.b-zz>)
     (holding <key.b-zz>)
     (inroom robot <room.r-ww>)
     (next-to robot <door.d-zz>)))
 (effects
   ((del (next-to robot <door.*48>))
    (del (inroom robot <room.*41>))
    (del (inroom <key.b-zz> <room.*42>))
    (add (inroom robot <room.r-zz>))
    (add (inroom <key.b-zz> <room.r-zz>)
    ))))

(GOTO-DR
 (params (<door.dx> <room.rx>))
 (preconds
   (and
     (is-door <door.dx>)
     (connects <door.dx> <room.ry>
                         <room.rx>)
     (inroom robot <room.rx>)))
 (effects
   ((del (next-to robot <box.*18>))
    (del (next-to robot <door.*18>))
    (del (next-to robot <key.*18>))
    (add (next-to robot <door.dx>)))))

(PUSH-BOX
 (params (<box.ba> <box.bb> <room.ra>))
 (preconds
   (and
     (is-object <box.ba>)
     (is-object <box.bb>)
```

```
     (inroom <box.bb> <room.ra>)
     (inroom <box.ba> <room.ra>)
     (pushable <box.ba>)
     (next-to robot <box.ba>)))
 (effects
   ((del (next-to robot <box.*14>))
    (del (next-to <box.ba> <door.*5>))
    (del (next-to <box.ba> <box.*5>))
    (del (next-to <box.*6> <box.ba>))
    (add (next-to robot <box.ba>))
    (add (next-to robot <box.bb>))
    (add (next-to <box.ba> <box.bb>))
    (add (next-to <box.bb> <box.ba>)))))

(GOTO-BOX
 (params (<box.b> <room.rm>))
 (preconds
   (and (is-object <box.b>)
        (inroom <box.b> <room.rm>)
        (inroom robot <room.rm>)))
 (effects
   ((add (next-to robot <box.b>))
    (del (next-to robot <box.*109>))
    (del (next-to robot <door.*109>))
    (del (next-to robot <key.*109>)))))

(GOTO-KEY
 (params (<key.b> <room.rm>))
 (preconds
   (and (is-object <key.b>)
        (inroom <key.b> <room.rm>)
        (inroom robot <room.rm>)))
 (effects
   ((add (next-to robot <key.b>))
    (del (next-to robot <box.*109>))
    (del (next-to robot <door.*109>))
    (del (next-to robot <key.*109>)))))

(OPEN
 (params (<door>))
 (preconds
   (and (is-door <door>)
        (unlocked <door>)
        (next-to robot <door>)
        (dr-closed <door>)))
 (effects
   ((del (dr-closed <door>))
    (add (dr-open <door>)))))

(CLOSE
 (params (<door.door1>))
 (preconds
   (and
     (is-door <door.door1>)
     (next-to robot <door.door1>)
     (dr-open <door.door1>)))
 (effects
   ((del (dr-open <door.door1>))
    (add (dr-closed <door.door1>)))))

(LOCK
 (params (<door.door2> <key.k1>
          <room.rm-b>))
 (preconds
   (and
     (is-door <door.door2>)
     (is-key <door.door2> <key.k1>)
     (connects <door.door2> <room.rm-c>
                            <room.rm-b>)
     (inroom <key.k1> <room.rm-b>)
     (holding <key.k1>)
     (next-to robot <door.door2>)
     (dr-closed <door.door2>)
     (unlocked <door.door2>)))
 (effects
   ((del (unlocked <door.door2>))
    (add (locked <door.door2>)))))

(UNLOCK
 (params (<door.door3> <key.k2>
          <room.rm-a>))
 (preconds
   (and
     (is-door <door.door3>)
     (is-key <door.door3> <key.k2>)
     (connects <door.door3> <room.rm-d>
                            <room.rm-a>)
     (inroom <key.k2> <room.rm-a>)
     (holding <key.k2>)
     (inroom robot <room.rm-a>)
     (next-to robot <door.door3>)
     (locked <door.door3>)))
 (effects
   ((del (locked <door.door3>))
    (add (unlocked <door.door3>)))))
```

```
(setq *AXIOMS*
   '(((next-to <box.1-axiom1> <box.2-axiom1>) .
                              ((inroom <box.1-axiom1> <room.axiom1>)
                               (inroom <box.2-axiom1> <room.axiom1>)))
     ((next-to robot <box.axiom2>) . ((inroom <box.axiom2> <room.axiom2>)
                                      (inroom robot <room.axiom2>)))
     ((next-to robot <key.axiom3>) . ((inroom <key.axiom3> <room.axiom3>)
                                      (inroom robot <room.axiom3>)))
     ((next-to robot <door.axiom4>) .
                           ((connects <door.axiom4> <room.x4> <room.y4>)
                            (inroom robot <room.y4>)))
     ((dr-open <door.axiom5>) . ((unlocked <door.axiom5>)))
     ((locked <door.axiom6>) . ((dr-closed <door.axiom6>)))
     ((~ (dr-open <door.axiom9>)) . ((dr-closed <door.axiom9>)))
     ((~ (dr-closed <door.axiom10>)) . ((dr-open <door.axiom10>)
                                        (unlocked <door.axiom10>)))
     ((~ (locked <door.axiom11>)) . ((unlocked <door.axiom11>)))
     ((~ (unlocked <door.axiom12>)) . ((locked <door.axiom12>)
                                       (dr-closed <door.axiom12>)))
     ((~ (arm-empty)) . ((holding <box.o13>)(holding <key.o13>)))
     ((~ (holding <box.axiom14>)) . ((arm-empty)))
     ((~ (holding <key.axiom15>)) . ((arm-empty)))))

(setq *VARIABLE-TYPING* '(
   (isa 'object 'type)(isa 'box 'object)(isa 'key 'object)
   (isa 'door 'object)(isa 'robot 'type)(isa 'room 'type)
   (isa-instance 'robot 'robot)(isa-instance 'box1 'box)
   (isa-instance 'box2 'box)(isa-instance 'box3 'box)
   (isa-instance 'room1 'room)(isa-instance 'room2 'room)
   (isa-instance 'room3 'room)(isa-instance 'room4 'room)
   (isa-instance 'room5 'room)(isa-instance 'room6 'room)
   (isa-instance 'room7 'room)(isa-instance 'door12 'door)
   (isa-instance 'door23 'door)(isa-instance 'door34 'door)
   (isa-instance 'door25 'door)(isa-instance 'door56 'door)
   (isa-instance 'door26 'door)(isa-instance 'door36 'door)
   (isa-instance 'door67 'door)(isa-instance 'key12 'key)
   (isa-instance 'key23 'key)(isa-instance 'key34 'key)
   (isa-instance 'key25 'key)(isa-instance 'key56 'key)
   (isa-instance 'key26 'key)(isa-instance 'key36 'key)
   (isa-instance 'key67 'key)(isa-instance 'rm1 'room)
   (isa-instance 'rm2 'room)(isa-instance 'dr12 'door)
   (isa-instance 'key12 'key)(isa-instance 'rm3 'room)
   (isa-instance 'rm4 'room)(isa-instance 'dr23 'door)
   (isa-instance 'dr34 'door)(isa-instance 'key23 'key)
   (isa-instance 'key34 'key)(isa-instance 'A 'box)
   (isa-instance 'B 'box)(isa-instance 'C 'box)
   (isa-instance 'D 'box)(isa-instance 'E 'box)
   (isa-instance 'F 'box)(isa-instance 'G 'box))
```

```
(setq *PRIMARY* '(
   ((holding <box>) . (PICKUP-BOX))
   ((holding <key>) . (PICKUP-KEY))
   ((arm-empty) . (PUTDOWN-BOX PUTDOWN-KEY))
   ((next-to <box.1> <box.2>) . (PUTDOWN-BOX-NEXT-TO PUSH-BOX))
   ((next-to <box> <door>) . (PUSH-BOX-TO-DR))
   ((inroom <box> <room>) . (PUSH-BOX-THRU-DR CARRY-BOX-THRU-DR))
   ((inroom robot <room>) . (GO-THRU-DR))
   ((inroom <key> <room>) . (CARRY-KEY-THRU-DR))
   ((next-to robot <door>) . (GOTO-DR))
   ((next-to robot <box>) . (GOTO-BOX))
   ((next-to robot <key>) . (GOTO-KEY))
   ((dr-open <door>) . (OPEN))
   ((dr-closed <door>). (CLOSE))
   ((locked <door>) . (LOCK))
   ((unlocked <door>) . (UNLOCK))))

Example problem:

Goal: '(and (next-to a d) (inroom b room3) (inroom a room4))

Initial State:
'((arm-empty) (dr-to-rm door67 room7) (dr-to-rm door67 room6)
  (connects door67 room7 room6) (connects door67 room6 room7)
  (dr-to-rm door56 room6) (dr-to-rm door56 room5)
  (connects door56 room6 room5) (connects door56 room5 room6)
  (dr-to-rm door36 room6) (dr-to-rm door36 room3)
  (connects door36 room6 room3) (connects door36 room3 room6)
  (dr-to-rm door25 room5) (dr-to-rm door25 room2)
  (connects door25 room5 room2) (connects door25 room2 room5)
  (dr-to-rm door34 room4) (dr-to-rm door34 room3)
  (connects door34 room4 room3) (connects door34 room3 room4)
  (dr-to-rm door23 room3) (dr-to-rm door23 room2)
  (connects door23 room3 room2) (connects door23 room2 room3)
  (dr-to-rm door12 room2) (dr-to-rm door12 room1)
  (connects door12 room2 room1) (connects door12 room1 room2)
  (next-to c e) (dr-closed door67) (locked door67)
  (dr-closed door56) (locked door56) (dr-closed door36)
  (locked door36) (unlocked door25) (dr-closed door25)
  (unlocked door34) (dr-open door34) (unlocked door23)
  (dr-closed door23) (dr-closed door12) (locked door12)
  (is-room room7) (is-room room6) (is-room room5) (is-room room4)
  (is-room room3) (is-room room2) (is-room room1) (is-door door67)
  (is-door door56) (is-door door36) (is-door door25)
  (is-door door34) (is-door door23) (is-door door12) (carriable e)
  (carriable d) (carriable c) (carriable b) (pushable d) (pushable c)
  (pushable b) (pushable a) (is-object key67)
```

```
(is-object key56) (is-object key36)
(is-object key25) (is-object key34)
(is-object key23) (is-object key12) (is-object e)
(is-box e) (is-object d)(is-box d) (is-object c)(is-box c)
(is-object b)(is-box b) (is-object a)(is-box a)
(inroom e room4) (inroom d room2) (inroom c room4) (inroom b room7)
(inroom a room3) (inroom key67 room6) (inroom key56 room1)
(inroom key36 room3) (inroom key25 room5)
(inroom key34 room1) (inroom key23 room7)
(inroom key12 room5) (inroom robot room5) (carriable key67)
(is-key door67 key67) (carriable key56)
(is-key door56 key56) (carriable key36)
(is-key door36 key36) (carriable key25)
(is-key door25 key25) (carriable key34)
(is-key door34 key34) (carriable key23)
(is-key door23 key23) (carriable key12)
(is-key door12 key12))
```

## B.2 Experimental Results

The experiments in this domain were run in CMU Common Lisp on a IBM RT Model 130 with 16 megabytes of memory. The tables below compare PRODIGY without any control knowledge, PRODIGY with a set of hand-code control rules, and PRODIGY with the abstractions generated by ALPINE. The first 100 problems are the test problems used in Minton's experiments [Minton, 1988a].

The entries in the table are defined as follows:

**Prob Num** The problem number.

**Time** Total CPU time used in solving the problem. A 600 CPU second time bound was imposed on all problems.

**Nodes** Total number of nodes searched in solving the problem.

**Len** Length of the solution found. Zero if no solution exists.

**ACT** Time required to create the abstraction hierarchy. This time is also included in the total CPU time for ALPINE.

**AbNodes** Nodes searched at each level in the hierarchy. Ordered from more abstract to less abstract levels.

**AbLen** Solution length found at each level in the hierarchy. Ordered from more abstract to less abstract levels.

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 0.6 | 14 | 6 | 0.8 | 14 | 6 | 2.1 | 14 | 6 | 0.8 | 6,8 | 2,4 |
| 2 | 0.3 | 10 | 4 | 0.5 | 10 | 4 | 1.5 | 10 | 4 | 0.7 | 4,0,6 | 1,0,3 |
| 3 | 0.6 | 20 | 6 | 0.7 | 14 | 6 | 1.7 | 14 | 6 | 0.7 | 4,0,10 | 1,0,5 |
| 4 | 0.5 | 15 | 6 | 0.8 | 15 | 6 | 2.2 | 17 | 6 | 1.0 | 4,4,9 | 1,1,4 |
| 5 | 0.5 | 16 | 7 | 0.8 | 16 | 7 | 1.8 | 18 | 7 | 0.6 | 6,6,6 | 2,2,3 |
| 6 | 0.3 | 12 | 5 | 0.5 | 12 | 5 | 1.3 | 12 | 5 | 0.6 | 8,4 | 3,2 |
| 7 | 1.0 | 32 | 7 | 1.6 | 31 | 7 | 3.9 | 80 | 7 | 1.2 | 72,8 | 3,4 |
| 8 | 1.3 | 35 | 15 | 1.9 | 33 | 15 | 3.3 | 35 | 15 | 1.0 | 4,8,23 | 1,3,11 |
| 9 | 0.8 | 24 | 8 | 1.0 | 18 | 8 | 2.2 | 16 | 7 | 1.1 | 8,8 | 3,4 |
| 10 | 0.9 | 26 | 12 | 1.5 | 26 | 12 | 2.4 | 26 | 9 | 0.9 | 16,10 | 4,5 |
| 11 | 0.7 | 21 | 8 | 1.0 | 19 | 8 | 2.4 | 21 | 8 | 1.0 | 4,4,13 | 1,1,6 |
| 12 | 1.4 | 31 | 14 | 2.0 | 30 | 14 | 3.8 | 31 | 14 | 1.3 | 17,14 | 7,7 |
| 13 | 0.1 | 6 | 2 | 0.2 | 6 | 2 | 1.7 | 8 | 2 | 1.2 | 2,4,2 | 0,1,1 |
| 14 | 1.1 | 27 | 12 | 1.3 | 21 | 9 | 3.0 | 23 | 9 | 1.2 | 4,11,8 | 1,4,4 |
| 15 | 0.9 | 24 | 11 | 1.5 | 24 | 11 | 2.8 | 26 | 11 | 1.0 | 4,6,16 | 1,2,8 |
| 16 | 0.6 | 20 | 9 | 1.0 | 20 | 9 | 2.2 | 20 | 9 | 0.9 | 10,10 | 4,5 |
| 17 | 0.6 | 20 | 9 | 1.0 | 20 | 9 | 1.2 | 10 | 4 | 0.6 | 4,6 | 1,3 |
| 18 | 0.2 | 8 | 3 | 0.4 | 8 | 3 | 1.2 | 8 | 3 | 0.7 | 4,4 | 1,2 |
| 19 | 0.6 | 17 | 7 | 0.9 | 17 | 7 | 2.8 | 19 | 7 | 1.3 | 4,4,11 | 1,1,5 |
| 20 | 1.6 | 33 | 11 | 1.7 | 24 | 11 | 4.2 | 31 | 11 | 1.4 | 23,8 | 7,4 |
| 21 | 3.7 | 108 | 15 | 3.0 | 53 | 15 | 4.2 | 36 | 16 | 1.5 | 4,12,20 | 1,5,10 |
| 22 | 0.5 | 16 | 7 | 0.9 | 16 | 7 | 2.2 | 18 | 7 | 0.8 | 6,6,6 | 2,2,3 |
| 23 | 1.7 | 35 | 6 | 0.9 | 14 | 6 | 3.7 | 16 | 6 | 2.0 | 4,8,4 | 1,3,2 |
| 24 | 1.3 | 35 | 13 | 1.6 | 28 | 13 | 4.8 | 21 | 8 | 2.7 | 2,11,8 | 0,4,4 |
| 25 | 0.2 | 8 | 3 | 0.3 | 8 | 3 | 2.0 | 10 | 3 | 1.3 | 2,4,4 | 0,1,2 |
| 26 | 1.1 | 34 | 13 | 1.4 | 28 | 13 | 4.0 | 32 | 14 | 1.5 | 4,10,18 | 1,4,9 |
| 27 | 0.8 | 20 | 9 | 1.1 | 20 | 9 | 2.6 | 22 | 9 | 1.0 | 4,6,12 | 1,2,6 |
| 28 | 0.8 | 26 | 8 | 1.0 | 18 | 8 | 3.2 | 34 | 8 | 1.2 | 2,22,10 | 0,3,5 |
| 29 | 2.7 | 64 | 10 | 1.5 | 22 | 10 | 5.2 | 38 | 11 | 1.9 | 26,12 | 5,6 |
| 30 | 1.1 | 34 | 12 | 1.4 | 26 | 12 | 3.3 | 34 | 12 | 1.2 | 20,14 | 5,7 |
| 31 | 0.9 | 20 | 8 | 1.2 | 20 | 8 | 3.4 | 21 | 8 | 1.3 | 4,8,9 | 1,3,4 |
| 32 | 1.9 | 46 | 20 | 2.7 | 44 | 20 | 5.0 | 47 | 20 | 1.2 | 9,16,22 | 3,6,11 |
| 33 | 3.2 | 84 | 20 | 2.6 | 44 | 20 | 4.5 | 47 | 20 | 1.1 | 9,12,26 | 3,4,13 |
| 34 | 1.0 | 25 | 10 | 1.4 | 25 | 10 | 3.0 | 26 | 10 | 1.1 | 4,9,13 | 1,3,6 |
| 35 | 2.6 | 57 | 23 | 3.0 | 47 | 20 | 4.6 | 39 | 17 | 1.4 | 9,14,16 | 3,6,8 |
| 36 | 3.4 | 86 | 21 | 2.7 | 46 | 21 | 5.0 | 49 | 21 | 1.0 | 9,10,10,20 | 3,4,4,10 |
| 37 | 1.3 | 37 | 11 | 1.3 | 25 | 11 | 2.7 | 26 | 11 | 1.1 | 4,10,12 | 1,4,6 |
| 38 | 0.9 | 29 | 4 | 1.4 | 29 | 4 | 1.9 | 22 | 4 | 1.0 | 18,4 | 2,2 |
| 39 | 3.2 | 79 | 24 | 3.4 | 55 | 24 | 4.6 | 43 | 19 | 1.4 | 9,8,26 | 3,3,13 |
| 40 | 1.1 | 27 | 10 | 1.5 | 27 | 10 | 3.0 | 34 | 10 | 0.9 | 18,6,10 | 3,2,5 |
| 41 | 0.7 | 18 | 8 | 1.1 | 18 | 8 | 2.2 | 18 | 8 | 1.0 | 8,10 | 3,5 |
| 42 | 1.6 | 35 | 8 | 2.1 | 35 | 8 | 3.3 | 22 | 8 | 1.3 | 8,6,8 | 2,2,4 |
| 43 | 1.9 | 51 | 14 | 2.1 | 36 | 14 | 3.4 | 34 | 14 | 1.1 | 4,10,20 | 1,3,10 |
| 44 | 0.6 | 18 | 5 | 0.9 | 18 | 5 | 1.5 | 14 | 5 | 0.7 | 8,6 | 2,3 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 45 | 0.6 | 17 | 5 | 0.8 | 17 | 5 | 1.6 | 16 | 5 | 0.6 | 8,2,6 | 2,0,3 |
| 46 | 0.2 | 6 | 2 | 0.3 | 6 | 2 | 1.3 | 6 | 2 | 0.9 | 4,2 | 1,1 |
| 47 | 1.7 | 41 | 15 | 2.5 | 41 | 15 | 3.6 | 38 | 15 | 1.0 | 22,16 | 7,8 |
| 48 | 3.4 | 77 | 25 | 4.0 | 62 | 25 | 6.1 | 51 | 22 | 1.6 | 11,10,30 | 4,3,15 |
| 49 | 0.8 | 24 | 10 | 1.2 | 22 | 10 | 2.5 | 22 | 10 | 1.0 | 12,10 | 5,5 |
| 50 | 2.3 | 53 | 22 | 3.2 | 53 | 22 | 5.0 | 51 | 22 | 1.4 | 4,17,30 | 1,6,15 |
| 51 | 3.5 | 82 | 29 | 4.5 | 71 | 29 | 7.7 | 74 | 33 | 1.5 | 10,24,40 | 4,10,19 |
| 52 | 1.4 | 44 | 7 | 0.9 | 17 | 7 | 2.3 | 16 | 7 | 1.3 | 10,6 | 4,3 |
| 53 | 2.3 | 53 | 22 | 3.2 | 53 | 22 | 4.9 | 51 | 22 | 1.4 | 4,17,30 | 1,6,15 |
| 54 | 0.5 | 12 | 5 | 0.8 | 12 | 5 | 3.4 | 24 | 7 | 1.5 | 4,10,4,6 | 1,2,1,3 |
| 55 | 1.0 | 27 | 12 | 1.5 | 26 | 12 | 3.1 | 27 | 12 | 1.3 | 15,12 | 6,6 |
| 56 | 4.3 | 99 | 20 | 3.7 | 55 | 20 | 6.4 | 54 | 22 | 1.8 | 12,16,26 | 4,5,13 |
| 57 | 0.1 | 6 | 2 | 0.2 | 6 | 2 | 1.6 | 6 | 2 | 1.3 | 4,2 | 1,1 |
| 58 | 1.1 | 28 | 12 | 1.7 | 28 | 12 | 4.1 | 28 | 12 | 2.0 | 4,14,10 | 1,6,5 |
| 59 | 8.3 | 205 | 23 | 7.5 | 122 | 23 | 7.9 | 87 | 23 | 2.0 | 6,54,27 | 2,8,13 |
| 60 | 6.3 | 169 | 27 | 5.1 | 79 | 27 | 7.5 | 64 | 28 | 1.9 | 10,20,34 | 4,8,16 |
| 61 | 4.0 | 112 | 22 | 6.3 | 110 | 22 | 6.6 | 99 | 22 | 1.2 | 77,22 | 11,11 |
| 62 | 1.1 | 29 | 10 | 1.7 | 29 | 10 | 4.1 | 30 | 10 | 2.0 | 4,18,8 | 1,5,4 |
| 63 | 6.2 | 149 | 24 | 9.0 | 147 | 24 | 7.9 | 61 | 25 | 2.5 | 4,25,10,22 | 1,9,4,11 |
| 64 | 3.9 | 78 | 26 | 5.0 | 71 | 26 | 9.0 | 62 | 26 | 3.4 | 40,22 | 15,11 |
| 65 | 36.6 | 955 | 21 | 3.7 | 65 | 21 | 5.0 | 47 | 17 | 2.0 | 4,25,18 | 1,7,9 |
| 66 | 6.7 | 167 | 35 | 4.1 | 62 | 27 | 8.6 | 101 | 30 | 1.9 | 11,55,35 | 4,9,17 |
| 67 | 1.3 | 41 | 6 | 0.8 | 15 | 6 | 2.5 | 25 | 6 | 1.3 | 19,6 | 3,3 |
| 68 | 4.9 | 119 | 31 | 3.0 | 47 | 18 | 7.0 | 74 | 18 | 2.0 | 4,49,21 | 1,7,10 |
| 69 | 0.5 | 14 | 6 | 0.8 | 14 | 6 | 2.3 | 14 | 6 | 1.3 | 6,8 | 2,4 |
| 70 | 4.0 | 107 | 26 | 4.1 | 67 | 26 | 11.0 | 152 | 37 | 1.2 | 112,6,34 | 18,2,17 |
| 71 | 0.8 | 23 | 9 | 1.3 | 23 | 9 | 3.3 | 24 | 9 | 1.5 | 10,6,8 | 3,2,4 |
| 72 | 15.1 | 352 | 25 | 3.9 | 55 | 25 | 3.0 | 18 | 8 | 0.8 | 6,0,12 | 2,0,6 |
| 73 | 0.4 | 14 | 5 | 0.7 | 14 | 5 | 1.5 | 16 | 7 | 0.4 | 10,6 | 4,3 |
| 74 | 1.1 | 35 | 9 | 1.8 | 35 | 9 | 2.2 | 31 | 10 | 0.4 | 23,8 | 6,4 |
| 75 | 2.0 | 46 | 19 | 2.8 | 46 | 19 | 2.8 | 34 | 14 | 0.5 | 14,20 | 4,10 |
| 76 | 2.7 | 56 | 24 | 3.8 | 56 | 24 | 3.2 | 36 | 16 | 0.4 | 16,20 | 6,10 |
| 77 | 33.9 | 821 | 31 | 14.6 | 231 | 31 | 4.2 | 37 | 17 | 0.8 | 11,4,22 | 4,2,11 |
| 78 | 3.6 | 77 | 14 | 4.8 | 77 | 14 | 4.6 | 34 | 14 | 0.8 | 14,4,16 | 4,2,8 |
| 79 | 1.2 | 25 | 11 | 1.7 | 25 | 11 | 2.4 | 25 | 11 | 0.5 | 11,14 | 4,7 |
| 80 | 0.2 | 6 | 2 | 0.2 | 6 | 2 | 0.7 | 6 | 2 | 0.8 | 4,2 | 1,1 |
| 81 | 1.8 | 42 | 18 | 2.6 | 42 | 18 | 3.4 | 39 | 18 | 0.5 | 15,24 | 6,12 |
| 82 | 0.4 | 10 | 4 | 0.6 | 10 | 4 | 1.6 | 10 | 4 | 0.8 | 6,4 | 2,2 |
| 83 | 2.6 | 59 | 26 | 3.7 | 59 | 26 | 5.3 | 50 | 23 | 1.2 | 28,22 | 12,11 |
| 84 | 3.9 | 73 | 31 | 4.9 | 73 | 31 | 4.3 | 33 | 13 | 1.3 | 6,14,13 | 2,5,6 |
| 85 | 24.1 | 591 | 31 | 32.2 | 551 | 31 | 8.3 | 134 | 22 | 0.8 | 104,30 | 7,15 |
| 86 | 2.2 | 56 | 22 | 2.9 | 50 | 19 | 3.5 | 42 | 18 | 0.7 | 28,14 | 11,7 |
| 87 | 1.1 | 32 | 9 | 1.7 | 32 | 9 | 2.0 | 21 | 9 | 0.6 | 13,8 | 5,4 |
| 88 | 1.2 | 32 | 13 | 1.8 | 32 | 13 | 2.0 | 21 | 9 | 0.6 | 13,8 | 5,4 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 89 | 5.2 | 104 | 32 | 6.8 | 104 | 32 | 8.1 | 75 | 32 | 1.1 | 11,20,44 | 4,7,21 |
| 90 | 48.6 | 1171 | 21 | 3.6 | 60 | 21 | 21.2 | 360 | 21 | 0.7 | 344,16 | 13,8 |
| 91 | 36.1 | 900 | 6 | 17.0 | 291 | 6 | 2.4 | 16 | 6 | 1.1 | 4,4,8 | 1,1,4 |
| 92 | 2.5 | 69 | 8 | 3.8 | 69 | 8 | 2.2 | 19 | 8 | 0.7 | 11,8 | 4,4 |
| 93 | 0.5 | 16 | 6 | 1.2 | 21 | 8 | 2.3 | 16 | 6 | 1.2 | 10,6 | 3,3 |
| 94 | 1.2 | 33 | 14 | 2.0 | 33 | 14 | 3.8 | 35 | 16 | 1.2 | 21,14 | 9,7 |
| 95 | 11.1 | 285 | 50 | 8.0 | 124 | 47 | 13.5 | 223 | 48 | 1.4 | 4,167,52 | 1,21,26 |
| 96 | 2.4 | 54 | 23 | 3.4 | 54 | 23 | 3.6 | 32 | 14 | 1.2 | 16,16 | 6,8 |
| 97 | 4.1 | 109 | 13 | 2.7 | 41 | 19 | 3.6 | 30 | 13 | 1.4 | 4,10,16 | 1,4,8 |
| 98 | 0.1 | 6 | 2 | 0.2 | 6 | 2 | 1.6 | 6 | 2 | 1.2 | 4,2 | 1,1 |
| 99 | 58.9 | 1367 | 30 | 4.7 | 67 | 30 | 6.4 | 61 | 27 | 1.3 | 4,27,30 | 1,11,15 |
| 100 | 3.2 | 80 | 18 | 4.8 | 80 | 18 | 5.1 | 49 | 18 | 1.1 | 27,6,16 | 8,2,8 |
| 101 | 0.4 | 8 | 3 | 0.5 | 8 | 3 | 1.6 | 8 | 3 | 0.4 | 4,4 | 1,2 |
| 102 | 5.0 | 91 | 37 | 6.6 | 91 | 37 | 7.5 | 85 | 37 | 0.5 | 39,46 | 14,23 |
| 103 | 2.2 | 44 | 14 | 3.0 | 44 | 14 | 4.4 | 33 | 14 | 0.7 | 4,11,18 | 1,4,9 |
| 104 | 3.5 | 107 | 0 | 3.4 | 65 | 0 | 3.8 | 87 | 0 | 0.4 | | |
| 105 | 3.6 | 75 | 26 | 5.0 | 75 | 26 | 5.4 | 63 | 26 | 0.5 | 31,32 | 10,16 |
| 106 | 0.1 | 6 | 2 | 0.2 | 6 | 2 | 1.1 | 6 | 2 | 0.6 | 4,2 | 1,1 |
| 107 | 1.4 | 24 | 8 | 1.7 | 24 | 8 | 3.0 | 18 | 8 | 0.7 | 6,2,10 | 2,1,5 |
| 108 | 0.1 | 6 | 2 | 0.2 | 6 | 2 | 1.1 | 6 | 2 | 1.0 | 4,2 | 1,1 |
| 109 | 0.2 | 6 | 2 | 0.2 | 6 | 2 | 0.8 | 6 | 2 | 0.4 | 4,2 | 1,1 |
| 110 | 29.4 | 648 | 22 | 40.1 | 644 | 22 | 5.8 | 50 | 21 | 0.7 | 14,4,32 | 4,2,15 |
| 111 | 0.4 | 6 | 2 | 0.5 | 6 | 2 | 1.8 | 6 | 2 | 0.7 | 4,0,2 | 1,0,1 |
| 112 | 1.0 | 24 | 10 | 1.4 | 24 | 10 | 2.0 | 23 | 10 | 0.4 | 9,14 | 3,7 |
| 113 | 0.1 | 6 | 2 | 0.3 | 6 | 2 | 0.8 | 6 | 2 | 0.4 | 4,2 | 1,1 |
| 114 | 0.6 | 14 | 6 | 0.9 | 14 | 6 | 1.5 | 14 | 6 | 0.4 | 4,10 | 1,5 |
| 115 | 41.2 | 939 | 26 | 10.2 | 156 | 26 | 20.6 | 297 | 26 | 1.0 | 271,26 | 13,13 |
| 116 | 1.5 | 40 | 12 | 2.2 | 40 | 12 | 2.5 | 31 | 12 | 0.4 | 15,16 | 4,8 |
| 117 | 113.5 | 2253 | 27 | 140.1 | 2053 | 27 | 25.1 | 344 | 29 | 1.0 | 316,28 | 15,14 |
| 118 | 15.2 | 355 | 25 | 7.8 | 118 | 25 | 4.0 | 31 | 13 | 0.7 | 13,0,18 | 4,0,9 |
| 119 | 267.7 | 5391 | 38 | 346.2 | 4941 | 38 | 84.9 | 1219 | 38 | 0.9 | 1183,36 | 20,18 |
| 120 | 6.4 | 124 | 45 | 8.4 | 122 | 45 | 5.2 | 58 | 25 | 0.5 | 26,32 | 9,16 |
| 121 | 0.8 | 14 | 6 | 0.9 | 14 | 6 | 2.3 | 14 | 6 | 0.7 | 4,2,8 | 1,1,4 |
| 122 | 5.6 | 119 | 28 | 7.8 | 119 | 28 | 6.3 | 76 | 28 | 0.5 | 42,34 | 11,17 |
| 123 | 7.3 | 159 | 25 | 10.1 | 159 | 25 | 4.1 | 60 | 11 | 0.4 | 46,14 | 4,7 |
| 124 | 3.0 | 62 | 25 | 4.1 | 62 | 25 | 5.3 | 60 | 25 | 0.6 | 36,24 | 13,12 |
| 125 | 6.5 | 165 | 14 | 2.0 | 31 | 14 | 3.4 | 30 | 14 | 0.7 | 4,6,20 | 1,3,10 |
| 126 | 1.5 | 37 | 11 | 2.3 | 37 | 11 | 2.2 | 10 | 4 | 1.0 | 6,4 | 2,2 |
| 127 | 7.8 | 186 | 15 | 7.0 | 107 | 21 | 4.1 | 35 | 13 | 1.0 | 4,17,14 | 1,5,7 |
| 128 | 8.8 | 179 | 31 | 11.7 | 174 | 31 | 7.7 | 68 | 28 | 1.0 | 6,23,39 | 2,7,19 |
| 129 | 42.9 | 833 | 55 | 37.7 | 541 | 55 | 30.7 | 395 | 49 | 0.9 | 349,46 | 26,23 |
| 130 | 82.2 | 1739 | 0 | 108.7 | 1683 | 0 | 12.1 | 196 | 0 | 1.1 | | |
| 131 | 40.7 | 1091 | 0 | 65.5 | 1091 | 0 | 122.5 | 2920 | 0 | 1.0 | | |
| 132 | 28.9 | 806 | 0 | 19.2 | 328 | 0 | 17.8 | 405 | 0 | 1.0 | | |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 133 | 4.2 | 110 | 0 | 5.9 | 98 | 0 | 10.0 | 193 | 0 | 0.8 | | |
| 134 | 33.5 | 833 | 0 | 15.2 | 250 | 0 | 17.5 | 371 | 0 | 1.3 | | |
| 135 | 19.8 | 395 | 50 | 11.4 | 137 | 55 | 16.4 | 152 | 46 | 1.5 | 114,38 | 27,19 |
| 136 | 1.2 | 27 | 10 | 1.8 | 27 | 10 | 2.9 | 26 | 10 | 0.6 | 10,6,10 | 3,2,5 |
| 137 | 6.3 | 114 | 48 | 8.6 | 113 | 48 | 11.6 | 113 | 48 | 1.3 | 69,44 | 26,22 |
| 138 | 62.2 | 1539 | 0 | 60.6 | 963 | 0 | 71.7 | 1525 | 0 | 1.2 | | |
| 139 | 124.9 | 2616 | 54 | 29.7 | 443 | 54 | 18.2 | 221 | 56 | 1.3 | 13,146,62 | 4,21,31 |
| 140 | 197.8 | 3874 | 19 | 133.6 | 1968 | 19 | 5.1 | 43 | 19 | 1.0 | 9,10,24 | 3,4,12 |
| 141 | 0.7 | 18 | 5 | 1.1 | 18 | 5 | 1.8 | 14 | 5 | 0.8 | 8,6 | 2,3 |
| 142 | 6.2 | 116 | 47 | 8.5 | 116 | 47 | 10.2 | 109 | 48 | 1.0 | 53,56 | 20,28 |
| 143 | 454.9 | 8883 | 0 | 444.3 | 6481 | 0 | 71.5 | 1154 | 0 | 1.0 | | |
| 144 | 3.6 | 107 | 0 | 3.6 | 65 | 0 | 4.5 | 98 | 0 | 0.6 | | |
| 145 | 9.1 | 172 | 48 | 9.6 | 132 | 48 | 8.2 | 74 | 34 | 1.1 | 6,24,44 | 2,10,22 |
| 146 | 0.5 | 12 | 5 | 0.7 | 12 | 5 | 1.8 | 12 | 5 | 0.9 | 8,4 | 3,2 |
| 147 | 2.5 | 46 | 16 | 3.3 | 46 | 16 | 4.6 | 46 | 16 | 0.9 | 46 | 16 |
| 148 | 0.8 | 21 | 8 | 1.2 | 21 | 8 | 2.3 | 16 | 7 | 0.9 | 10,6 | 4,3 |
| 149 | 0.7 | 18 | 6 | 1.1 | 18 | 6 | 1.7 | 14 | 6 | 0.6 | 8,6 | 3,3 |
| 150 | 30.4 | 664 | 30 | 40.8 | 660 | 30 | 12.6 | 138 | 29 | 1.5 | 108,30 | 14,15 |
| 151 | 2.9 | 42 | 14 | 3.7 | 42 | 14 | 5.2 | 20 | 8 | 1.8 | 6,6,8 | 2,2,4 |
| 152 | 160.1 | 3130 | 50 | 11.0 | 161 | 50 | 16.0 | 139 | 59 | 1.3 | 14,43,26,56 | 4,15,12,28 |
| 153 | 11.2 | 205 | 67 | 104.8 | 1415 | 51 | 12.7 | 156 | 29 | 1.3 | 14,111,31 | 3,11,15 |
| 154 | 9.1 | 188 | 36 | 16.4 | 239 | 45 | 13.4 | 135 | 42 | 2.0 | 14,69,52 | 4,13,25 |
| 155 | 336.6 | 6605 | 56 | 9.4 | 127 | 56 | 12.6 | 119 | 52 | 1.7 | 4,57,58 | 1,22,29 |
| 156 | 70.9 | 1453 | 71 | 19.4 | 267 | 71 | 37.5 | 470 | 69 | 1.9 | 402,68 | 35,34 |
| 157 | 40.7 | 853 | 31 | 53.2 | 836 | 27 | 21.4 | 422 | 27 | 1.0 | 388,34 | 10,17 |
| 158 | 174.7 | 3428 | 39 | 223.4 | 3226 | 39 | 72.0 | 985 | 39 | 1.9 | 957,28 | 25,14 |
| 159 | 19.2 | 436 | 32 | 13.3 | 199 | 32 | 6.9 | 50 | 20 | 1.4 | 19,8,23 | 6,3,11 |
| 160 | 294.7 | 5927 | 60 | 30.0 | 447 | 57 | 16.1 | 140 | 50 | 2.3 | 20,84,36 | 5,27,18 |
| 161 | 14.0 | 259 | 59 | 7.0 | 95 | 31 | 21.9 | 269 | 59 | 1.0 | 269 | 59 |
| 162 | 11.1 | 202 | 61 | 12.4 | 162 | 61 | 9.6 | 94 | 39 | 1.3 | 4,38,52 | 1,12,26 |
| 163 | 16.2 | 365 | 41 | 22.4 | 347 | 41 | 16.0 | 221 | 45 | 1.6 | 4,169,48 | 1,20,24 |
| 164 | 9.1 | 178 | 45 | 12.9 | 177 | 45 | 13.1 | 126 | 45 | 1.6 | 86,40 | 25,20 |
| 165 | 9.5 | 205 | 44 | 13.5 | 205 | 44 | 24.7 | 485 | 53 | 1.3 | 423,62 | 22,31 |
| 166 | 11.4 | 200 | 68 | 9.2 | 124 | 39 | 10.6 | 100 | 42 | 1.5 | 6,49,45 | 2,18,22 |
| 167 | 61.4 | 1264 | 54 | 78.2 | 1130 | 54 | 20.4 | 272 | 51 | 1.5 | 4,214,54 | 1,23,27 |
| 168 | 97.4 | 2042 | 27 | 4.7 | 70 | 27 | 75.7 | 1177 | 23 | 1.4 | 1161,16 | 15,8 |
| 169 | 1.5 | 33 | 15 | 2.3 | 33 | 15 | 4.0 | 62 | 11 | 0.9 | 50,12 | 5,6 |
| 170 | 2.5 | 58 | 15 | 3.7 | 58 | 15 | 4.4 | 34 | 15 | 1.3 | 2,14,18 | 0,6,9 |
| 171 | 4.3 | 83 | 13 | 9.1 | 133 | 29 | 11.8 | 118 | 35 | 2.4 | 4,78,36 | 1,16,18 |
| 172 | 33.2 | 685 | 70 | 12.9 | 177 | 63 | 15.1 | 173 | 59 | 1.5 | 9,84,80 | 3,16,40 |
| 173 | 194.6 | 4069 | 76 | 17.1 | 224 | 69 | 19.2 | 194 | 45 | 2.5 | 15,141,38 | 5,21,19 |
| 174 | 13.1 | 382 | 0 | 7.4 | 124 | 0 | 14.1 | 358 | 0 | 1.1 | | |
| 175 | 5.4 | 153 | 0 | 9.0 | 148 | 0 | 6.4 | 134 | 0 | 1.2 | | |
| 176 | 600.0 | 11452 | — | 600.0 | 8153 | — | 600.0 | 11058 | — | 1.9 | | |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 177 | 150.8 | 3028 | 83 | 50.1 | 753 | 73 | 124.3 | 1925 | 84 | 2.2 | 1857,68 | 50,34 |
| 178 | 20.0 | 491 | 0 | 18.0 | 297 | 0 | 12.8 | 261 | 0 | 1.4 | | |
| 179 | 62.4 | 1241 | 59 | 64.4 | 877 | 59 | 13.7 | 129 | 56 | 1.8 | 16,38,75 | 5,14,37 |
| 180 | 134.7 | 2894 | 14 | 137.6 | 2041 | 9 | 6.4 | 42 | 18 | 1.6 | 8,12,22 | 3,5,10 |
| 181 | 306.8 | 5697 | 0 | 72.2 | 1009 | 0 | 47.6 | 925 | 0 | 1.7 | | |
| 182 | 600.0 | 11506 | — | 600.0 | 8087 | — | 12.0 | 134 | 0 | 3.2 | | |
| 183 | 252.2 | 6010 | 0 | 77.7 | 1202 | 0 | 36.2 | 584 | 0 | 2.9 | | |
| 184 | 271.9 | 5283 | 63 | 80.2 | 1167 | 59 | 95.2 | 1448 | 64 | 2.1 | 1392,56 | 36,28 |
| 185 | 39.0 | 749 | 44 | 8.2 | 107 | 44 | 10.3 | 84 | 35 | 2.0 | 11,33,40 | 4,11,20 |
| 186 | 30.0 | 854 | 0 | 15.2 | 264 | 0 | 7.3 | 121 | 0 | 1.5 | | |
| 187 | 7.4 | 132 | 35 | 8.5 | 115 | 32 | 8.7 | 63 | 26 | 1.7 | 14,18,31 | 4,8,14 |
| 188 | 600.0 | 11466 | — | 600.0 | 8305 | — | 600.0 | 8954 | — | 2.7 | | |
| 189 | 5.9 | 133 | 33 | 8.7 | 133 | 33 | 6.5 | 49 | 22 | 1.6 | 4,17,28 | 1,7,14 |
| 190 | 36.1 | 780 | 60 | 46.2 | 696 | 60 | 31.0 | 437 | 55 | 1.9 | 6,369,62 | 2,22,31 |
| 191 | 173.8 | 3688 | 0 | 137.3 | 1994 | 0 | 20.8 | 369 | 0 | 1.8 | | |
| 192 | 1.9 | 45 | 15 | 2.7 | 45 | 15 | 4.2 | 45 | 15 | 1.2 | 45 | 15 |
| 193 | 30.6 | 714 | 16 | 30.4 | 453 | 31 | 6.7 | 39 | 16 | 1.8 | 6,13,8,12 | 2,5,4,5 |
| 194 | 27.9 | 640 | 31 | 16.9 | 268 | 31 | 10.2 | 72 | 31 | 2.1 | 11,27,34 | 4,11,16 |
| 195 | 10.8 | 207 | 50 | 14.4 | 206 | 50 | 10.4 | 69 | 31 | 1.9 | 12,19,38 | 5,8,18 |
| 196 | 39.7 | 899 | 65 | 26.1 | 421 | 65 | 13.7 | 126 | 47 | 2.0 | 12,54,60 | 5,12,30 |
| 197 | 20.4 | 407 | 63 | 22.6 | 342 | 59 | 21.0 | 197 | 72 | 1.7 | 19,97,81 | 7,26,39 |
| 198 | 4.8 | 106 | 27 | 7.9 | 126 | 29 | 20.2 | 396 | 30 | 1.4 | 356,14,26 | 11,6,13 |
| 199 | 256.6 | 5266 | 0 | 94.8 | 1448 | 0 | 35.4 | 633 | 0 | 2.0 | | |
| 200 | 35.8 | 804 | 33 | 12.5 | 194 | 33 | 10.6 | 114 | 33 | 1.5 | 11,67,36 | 3,12,18 |
| 201 | 203.4 | 4003 | 90 | 471.5 | 6691 | 66 | 34.5 | 399 | 70 | 2.8 | 8,311,20,60 | 3,28,10,29 |
| 202 | 50.2 | 1154 | 53 | 23.1 | 365 | 56 | 18.5 | 183 | 70 | 2.6 | 2,121,60 | 0,40,30 |
| 203 | 320.0 | 6544 | 0 | 169.1 | 2518 | 0 | 600.0 | 8735 | — | 1.6 | | |
| 204 | 600.0 | 11947 | — | 20.1 | 290 | 66 | 19.2 | 142 | 62 | 2.4 | 12,62,20,48 | 3,26,9,24 |
| 205 | 27.5 | 792 | 0 | 17.6 | 303 | 0 | 5.6 | 71 | 0 | 1.7 | | |
| 206 | 445.0 | 8166 | 77 | 35.6 | 537 | 77 | 23.6 | 212 | 87 | 3.3 | 12,112,88 | 5,38,44 |
| 207 | 6.7 | 114 | 36 | 8.2 | 113 | 35 | 12.1 | 122 | 36 | 1.7 | 122 | 36 |
| 208 | 600.0 | 11453 | — | 15.6 | 219 | 78 | 23.7 | 321 | 70 | 1.9 | 9,232,80 | 3,27,40 |
| 209 | 82.2 | 2077 | 0 | 22.4 | 371 | 0 | 223.0 | 5239 | 0 | 1.9 | | |
| 210 | 600.0 | 11239 | — | 600.0 | 8338 | — | 600.0 | 9394 | — | 1.7 | | |
| 211 | 114.4 | 2770 | 0 | 97.1 | 1551 | 0 | 11.9 | 217 | 0 | 2.1 | | |
| 212 | 34.0 | 745 | 38 | 50.3 | 745 | 38 | 20.1 | 248 | 41 | 1.7 | 2,198,14,34 | 0,18,6,17 |
| 213 | 600.0 | 12678 | — | 600.0 | 8807 | — | 600.0 | 10878 | — | 3.0 | | |
| 214 | 600.0 | 11275 | — | 600.0 | 7831 | — | 600.0 | 9596 | — | 2.9 | | |
| 215 | 115.5 | 2962 | 0 | 121.6 | 1857 | 0 | 105.6 | 2165 | 0 | 2.5 | | |
| 216 | 600.0 | 11112 | — | 600.0 | 8432 | — | 600.0 | 9496 | — | 2.2 | | |
| 217 | 9.2 | 187 | 37 | 12.5 | 187 | 37 | 11.1 | 142 | 33 | 1.9 | 106,36 | 15,18 |
| 218 | 600.0 | 12121 | — | 600.0 | 8131 | — | 600.0 | 10323 | — | 2.5 | | |
| 219 | 71.4 | 1544 | 0 | 91.0 | 1396 | 0 | 62.2 | 1367 | 0 | 2.3 | | |
| 220 | 26.7 | 546 | 61 | 27.1 | 415 | 61 | 16.3 | 158 | 56 | 2.3 | 16,77,65 | 5,19,32 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 221 | 124.9 | 2544 | 44 | 83.2 | 1188 | 51 | 16.7 | 177 | 48 | 2.3 | 6,131,40 | 2,26,20 |
| 222 | 600.0 | 11902 | — | 600.0 | 8690 | — | 13.8 | 89 | 38 | 2.6 | 14,29,14,32 | 5,11,7,15 |
| 223 | 600.0 | 12743 | — | 94.9 | 1486 | 0 | 31.0 | 628 | 0 | 2.3 | | |
| 224 | 8.6 | 153 | 59 | 10.9 | 153 | 59 | 16.1 | 128 | 55 | 2.1 | 11,41,22,54 | 4,14,10,27 |
| 225 | 145.3 | 2837 | 36 | 8.6 | 130 | 36 | 106.4 | 1513 | 27 | 2.5 | 1491,22 | 16,11 |
| 226 | 600.0 | 10917 | — | 600.0 | 8117 | — | 600.0 | 8993 | — | 2.8 | | |
| 227 | 8.5 | 150 | 49 | 7.9 | 102 | 41 | 10.7 | 112 | 30 | 2.6 | 4,72,12,24 | 1,12,5,12 |
| 228 | 12.8 | 234 | 51 | 13.5 | 176 | 39 | 12.2 | 99 | 42 | 2.6 | 8,44,47 | 3,16,23 |
| 229 | 20.2 | 413 | 62 | 14.2 | 202 | 52 | 20.5 | 175 | 59 | 4.5 | 6,119,50 | 2,32,25 |
| 230 | 202.9 | 4163 | 99 | 23.7 | 294 | 92 | 35.8 | 367 | 94 | 4.3 | 15,280,72 | 5,53,36 |
| 231 | 22.2 | 561 | 0 | 18.0 | 297 | 0 | 30.3 | 614 | 0 | 2.4 | | |
| 232 | 37.4 | 955 | 0 | 33.7 | 540 | 0 | 48.9 | 948 | 0 | 2.6 | | |
| 233 | 600.0 | 11519 | — | 600.0 | 8068 | — | 15.6 | 200 | 0 | 4.3 | | |
| 234 | 600.0 | 11114 | — | 600.0 | 8110 | — | 600.0 | 9573 | — | 2.7 | | |
| 235 | 55.4 | 1339 | 0 | 43.7 | 721 | 0 | 27.3 | 568 | 0 | 2.3 | | |
| 236 | 84.2 | 1580 | 77 | 68.8 | 918 | 77 | 22.6 | 241 | 76 | 2.7 | 25,117,99 | 8,19,49 |
| 237 | 143.2 | 2893 | 14 | 137.1 | 2043 | 10 | 8.5 | 44 | 19 | 2.4 | 8,12,6,18 | 3,5,3,8 |
| 238 | 600.0 | 11046 | — | 600.0 | 8066 | — | 133.1 | 2474 | 0 | 4.5 | | |
| 239 | 246.8 | 6090 | 0 | 81.6 | 1274 | 0 | 37.1 | 584 | 0 | 3.8 | | |
| 240 | 281.2 | 5512 | 85 | 83.7 | 1222 | 81 | 159.8 | 2215 | 86 | 3.0 | 2145,70 | 51,35 |
| 241 | 59.0 | 1140 | 62 | 11.9 | 142 | 59 | 14.8 | 116 | 50 | 2.9 | 18,41,57 | 7,15,28 |
| 242 | 73.5 | 1968 | 0 | 30.2 | 522 | 0 | 17.3 | 314 | 0 | 2.6 | | |
| 243 | 600.0 | 11084 | — | 600.0 | 7781 | — | 600.0 | 10668 | — | 2.6 | | |
| 244 | 6.3 | 133 | 33 | 8.9 | 133 | 33 | 8.2 | 54 | 23 | 2.2 | 4,24,8,18 | 1,10,3,9 |
| 245 | 600.0 | 11596 | — | 199.0 | 2868 | 0 | 31.9 | 615 | 0 | 2.4 | | |
| 246 | 75.2 | 1626 | 0 | 21.4 | 365 | 0 | 600.0 | 10265 | — | 2.6 | | |
| 247 | 600.0 | 11554 | — | 600.0 | 8623 | — | 14.9 | 153 | 38 | 2.5 | 8,99,14,32 | 3,13,7,15 |
| 248 | 45.2 | 922 | 46 | 22.5 | 321 | 46 | 19.3 | 132 | 58 | 4.5 | 13,75,44 | 5,31,22 |
| 249 | 14.2 | 248 | 65 | 18.8 | 247 | 65 | 16.6 | 103 | 46 | 2.8 | 12,35,16,40 | 5,14,7,20 |
| 250 | 40.3 | 901 | 66 | 28.0 | 423 | 66 | 17.5 | 144 | 55 | 2.8 | 12,58,22,52 | 5,14,10,26 |

# Appendix C

# Machine-Shop Planning and Scheduling

The version of the machine-shop domain used in these experiments is almost identical to the original PRODIGY version presented in [Minton, 1988a]. There are only two minor syntactic differences from the original problem-space definition. First, the `polish` operator had a disjunctive precondition in the original domain and in the version used here this operator was separated into two operators. ALPINE can handle the full PRODIGY language, but it does so in a conservative manner and it forces disjunctive conditions into the same abstraction level. This particular operator was separated into two operators so that ALPINE could produce a finer-grained hierarchy. Second, the fact that an operation could not be performed on an object once it was joined was made explicit by adding negated joined conditions to each operator. This condition was implicit in the fact that the original objects are deleted when they are joined. This change allows the system to separate the joined literals from some of the other literals in a few additional situations.

## C.1 Problem Space Definition

```
(POLISH-1
  (params (<obj-pc> <time-pc> <time-prev-pc>))
  (preconds
    (and
      (is-object <obj-pc>)
      (~ (joined <obj-pc> <obj-pc2> <or-pc>))
      (~ (joined <obj-pc2> <obj-pc> <or-pc>))
      (clampable <obj-pc> POLISHER)
      (last-scheduled <obj-pc> <time-prev-pc>)
      (later <time-pc> <time-prev-pc>)
```

```
      (idle POLISHER <time-pc>)))
  (effects (
    (del (surface-condition <obj-pc> <surface-*7-pc>))
    (add (surface-condition <obj-pc> POLISHED))
    (del (last-scheduled <obj-pc> <time-prev-pc>))
    (add (last-scheduled <obj-pc> <time-pc>))
    (add (scheduled <obj-pc> POLISHER <time-pc>)))))

(POLISH-2
  (params (<obj-pr> <time-pr> <time-prev-pr>))
  (preconds
    (and
      (is-object <obj-pr>)
      (~ (joined <obj-pr> <obj-pr2> <or-pr>))
      (~ (joined <obj-pr2> <obj-pr> <or-pr>))
      (shape <obj-pr> RECTANGULAR)
      (last-scheduled <obj-pr> <time-prev-pr>)
      (later <time-pr> <time-prev-pr>)
      (idle POLISHER <time-pr>)))
  (effects (
    (del (surface-condition <obj-pr> <surface-*7-pr>))
    (add (surface-condition <obj-pr> POLISHED))
    (del (last-scheduled <obj-pr> <time-prev-pr>))
    (add (last-scheduled <obj-pr> <time-pr>))
    (add (scheduled <obj-pr> POLISHER <time-pr>)))))

(GRIND
  (params (<obj-g> <time-g> <time-prev-g>))
  (preconds
    (and
      (is-object <obj-g>)
      (~ (joined <obj-g> <obj-g2> <or-g>))
      (~ (joined <obj-g2> <obj-g> <or-g>))
      (last-scheduled <obj-g> <time-prev-g>)
      (later <time-g> <time-prev-g>)
      (idle GRINDER <time-g>)))
  (effects (
     (del (surface-condition <obj-g> <surface-*1-g>))
     (add (surface-condition <obj-g> SMOOTH))
     (del (painted <obj-g> <color-*2-g>))
     (del (last-scheduled <obj-g> <time-prev-g>))
     (add (last-scheduled <obj-g> <time-g>))
     (add (scheduled <obj-g> GRINDER <time-g>)))))

(ROLL
  (params (<obj-r> <time-r> <time-prev-r>))
  (preconds
    (and
```

```
      (is-object <obj-r>)
      (~ (joined <obj-r> <obj-r2> <or-r>))
      (~ (joined <obj-r2> <obj-r> <or-r>))
      (last-scheduled <obj-r> <time-prev-r>)
      (later <time-r> <time-prev-r>)
      (idle ROLLER <time-r>)
      (shape <obj-r> <shape-old-r>)))
  (effects (
     (del (shape <obj-r> <shape-old-r>))
     (del (temperature <obj-r> <temp-old-r>))
     (del (has-hole <obj-r> <width-*3-r> <orientation-*4-r>>))
     (del (surface-condition <obj-r> <surface-*1-r>))
     (del (painted <obj-r> <color-*2-r>))
     (del (last-scheduled <obj-r> <time-prev-r>))
     (add (temperature <obj-r> HOT))
     (add (shape <obj-r> CYLINDRICAL))
     (add (last-scheduled <obj-r> <time-r>))
     (add (scheduled <obj-r> ROLLER <time-r>)))))

(LATHE
  (params (<obj-l> <time-l> <shape-l> <time-prev-l>))
  (preconds
    (and
      (is-object <obj-l>)
      (~ (joined <obj-l> <obj-l2> <or-l>))
      (~ (joined <obj-l2> <obj-l> <or-l>))
      (last-scheduled <obj-l> <time-prev-l>)
      (later <time-l> <time-prev-l>)
      (idle LATHE <time-l>)
      (shape <obj-l> <shape-l>)))
  (effects (
     (del (shape <obj-l> <shape-l>))
     (del (surface-condition <obj-l> <surface-*3-l>))
     (del (painted <obj-l> <color-*4-l>))
     (del (last-scheduled <obj-l> <time-prev-l>))
     (add (surface-condition <obj-l> ROUGH))
     (add (shape <obj-l> CYLINDRICAL))
     (add (last-scheduled <obj-l> <time-l>))
     (add (scheduled <obj-l> LATHE <time-l>)))))

(PUNCH
  (params (<obj-u> <time-u> <width-hole-u> <orientation-u> <time-prev-u>))
  (preconds
    (and
      (is-object <obj-u>)
      (~ (joined <obj-u> <obj-u2> <or-u>))
      (~ (joined <obj-u2> <obj-u> <or-u>))
      (is-punchable <obj-u> <width-hole-u> <orientation-u>)
```

```
      (clampable <obj-u> PUNCH)
      (last-scheduled <obj-u> <time-prev-u>)
      (later <time-u> <time-prev-u>)
      (idle PUNCH <time-u>)))
  (effects (
    (del (surface-condition <obj-u> <surface-*33-u>))
    (del (last-scheduled <obj-u> <time-prev-u>))
    (add (surface-condition <obj-u> ROUGH))
    (add (has-hole <obj-u> <width-hole-u> <orientation-u>))
    (add (last-scheduled <obj-u> <time-u>))
    (add (scheduled <obj-u> PUNCH <time-u>)))))

(DRILL-PRESS
  (params (<obj-d> <time-d> <width-hole-d> <orientation-d> <time-prev-d>))
  (preconds
    (and
      (is-object <obj-d>)
      (~ (joined <obj-d> <obj-d2> <or-d>))
      (~ (joined <obj-d2> <obj-d> <or-d>))
      (is-drillable <obj-d> <orientation-d>)
      (last-scheduled <obj-d> <time-prev-d>)
      (later <time-d> <time-prev-d>)
      (idle DRILL-PRESS <time-d>)
      (have-bit <width-hole-d>)))
  (effects (
    (del (last-scheduled <obj-d> <time-prev-d>))
    (add (has-hole <obj-d> <width-hole-d> <orientation-d>))
    (add (last-scheduled <obj-d> <time-d>))
    (add (scheduled <obj-d> DRILL-PRESS <time-d>)))))

(BOLT
  (params (<obj-1-b> <obj-2-b> <time-b> <obj-new-b> <time-prev1-b>
           <time-prev2-b> <orientation-b> <width-b> <bolt-b>))
  (preconds
      (and
        (is-object <obj-1-b>)
        (is-object <obj-2-b>)
        (~ (joined <obj-1-b> <obj-1-b2> <or-b>))
        (~ (joined <obj-1-b2> <obj-1-b> <or-b>))
        (~ (joined <obj-2-b> <obj-2-b2> <or-b>))
        (~ (joined <obj-2-b2> <obj-2-b> <or-b>))
        (can-be-bolted <obj-1-b> <obj-2-b> <orientation-b>)
        (is-bolt <bolt-b>)
        (is-width <width-b> <bolt-b>)
        (has-hole <obj-1-b> <width-b> <orientation-b>)
        (has-hole <obj-2-b> <width-b> <orientation-b>)
        (last-scheduled <obj-1-b> <time-prev1-b>)
        (last-scheduled <obj-2-b> <time-prev2-b>)
```

```
        (later <time-b> <time-prev1-b>)
        (later <time-b> <time-prev2-b>)
        (idle BOLTING-MACHINE <time-b>)
        (composite-object <obj-new-b> <orientation-b> <obj-1-b> <obj-2-b>)))
  (effects (
     (del (last-scheduled <obj-1-b> <time-prev1-b>))
     (del (last-scheduled <obj-2-b> <time-prev2-b>))
     (add (last-scheduled <obj-new-b> <time-b>))
     (add (is-object <obj-new-b>))
     (del (is-object <obj-1-b>))
     (del (is-object <obj-2-b>))
     (add (joined <obj-1-b> <obj-2-b> <orientation-b>))
     (add (scheduled <obj-new-b> BOLTING-MACHINE <time-b>)))))

(WELD
  (params (<obj-1-w> <obj-2-w> <time-w> <obj-new-w> <time-prev1-w>
           <time-prev2-w> <orientation-w>))
  (preconds
      (and
        (is-object <obj-1-w>)
        (is-object <obj-2-w>)
        (~ (joined <obj-1-w> <obj-1-w2> <or-w>))
        (~ (joined <obj-1-w2> <obj-1-w> <or-w>))
        (~ (joined <obj-2-w> <obj-2-w2> <or-w>))
        (~ (joined <obj-2-w2> <obj-2-w> <or-w>))
        (can-be-welded <obj-1-w> <obj-2-w> <orientation-w>)
        (last-scheduled <obj-1-w> <time-prev1-w>)
        (last-scheduled <obj-2-w> <time-prev2-w>)
        (later <time-w> <time-prev1-w>)
        (later <time-w> <time-prev2-w>)
        (idle WELDER <time-w>)
        (composite-object <obj-new-w> <orientation-w> <obj-1-w> <obj-2-w>)))
  (effects (
     (del (last-scheduled <obj-1-w> <time-prev1-w>))
     (del (last-scheduled <obj-2-w> <time-prev2-w>))
     (add (last-scheduled <obj-new-w> <time-w>))
     (del (temperature <obj-new-w> <temp-old*>))
     (add (temperature <obj-new-w> HOT))
     (add (is-object <obj-new-w>))
     (del (is-object <obj-1-w>))
     (del (is-object <obj-2-w>))
     (add (joined <obj-1-w> <obj-2-w> <orientation-w>))
     (add (scheduled <obj-new-w> WELDER <time-w>)))))

(SPRAY-PAINT
  (params (<obj-s> <time-s> <color-s> <time-prev-s>))
  (preconds
    (and
```

```
      (sprayable <color-s>)
      (is-object <obj-s>)
      (~ (joined <obj-s> <obj-s2> <or-s>))
      (~ (joined <obj-s2> <obj-s> <or-s>))
      (shape <obj-s> <shape-s-s>)
      (regular-shape <shape-s-s>)
      (clampable <obj-s> SPRAY-PAINTER)
      (last-scheduled <obj-s> <time-prev-s>)
      (later <time-s> <time-prev-s>)
      (idle SPRAY-PAINTER <time-s>)))
  (effects (
    (add (painted <obj-s> <color-s>))
    (del (surface-condition <obj-s> <surface-*2-s>))
    (del (last-scheduled <obj-s> <time-prev-s>))
    (add (last-scheduled <obj-s> <time-s>))
    (add (scheduled <obj-s> SPRAY-PAINTER <time-s>)))))

(IMMERSION-PAINT
  (params (<obj-i> <time-i> <color-i> <time-prev-i>))
  (preconds
    (and
      (is-object <obj-i>)
      (~ (joined <obj-i> <obj-i2> <or-i>))
      (~ (joined <obj-i2> <obj-i> <or-i>))
      (have-paint-for-immersion <color-i>)
      (last-scheduled <obj-i> <time-prev-i>)
      (later <time-i> <time-prev-i>)
      (idle IMMERSION-PAINTER <time-i>)
      ))
  (effects (
    (add (painted <obj-i> <color-i>))
    (del (last-scheduled <obj-i> <time-prev-i>))
    (add (last-scheduled <obj-i> <time-i>))
    (add (scheduled <obj-i> IMMERSION-PAINTER <time-i>)))))

(IS-CLAMPABLE
  (params (<obj-1> <machine>))
  (preconds
    (and
      (has-clamp <machine>)
      (temperature <obj-1> COLD)))
  (effects ((add (clampable <obj-1> <machine>)))))

(INFER-IDLE
  (params (<machine-1>  <time-t>))
  (preconds
     (forall (<obj-2> <machine-2>)
             (scheduled <obj-2> <machine-2> <time-t>)
```

```
            (not-equal <machine-2> <machine-1>)))
  (effects (
    (add (idle <machine-1> <time-t>)))))

(setq *AXIOMS* nil)

(setq *VARIABLE-TYPING* nil)

(setq *PRIMARY*
      '(((surface-condition <obj> POLISHED) . (POLISH-1 POLISH-2))
        ((surface-condition <obj> SMOOTH) . (GRIND))
        ((shape <obj> CYLINDRICAL) . (ROLL LATHE))
        ((shape <obj> RECTANGULAR) . nil)
        ((has-hole <obj> <width> <orientation>) . (PUNCH DRILL-PRESS))
        ((joined <obj1> <obj2> <orientation>) . (BOLT WELD))
        ((painted <obj> <color>) . (SPRAY-PAINT IMMERSION-PAINT))
        ((clampable <obj> <machine>) . (IS-CLAMPABLE))
        ((idle <machine> <time>) . (INFER-IDLE))
        ((last-scheduled <obj> <time>) . nil)
        ((scheduled <obj> <machine> <time>) . nil)
        ((is-object <obj>) . nil)
        ((temperature <obj> <temp>) . nil)
        ((~ (joined <obj1> <obj2> <orientation>)) . (t))
        ))

Example Problem:

Goal: '(and (has-hole d (4 mm) orientation-4) (shape d cylindrical)
            (surface-condition e smooth) (painted d (water-res white)))

Initial State:
'((last-time 10) (is-bolt (b1 (1.199999 cm))) (is-bolt (b2 (1 cm)))
  (is-bolt (b3 (4 mm))) (is-bolt (b4 (1.4 cm)))
  (is-bolt (b5 (1.4 cm))) (last-scheduled e 0) (last-scheduled d 0)
  (last-scheduled c 0) (last-scheduled b 0) (last-scheduled a 0)
  (has-hole e (8 mm) orientation-4) (surface-condition e rough)
  (temperature e cold) (shape e irregular) (is-object e)
  (painted d (regular red)) (temperature d cold)
  (shape d undetermined) (is-object d) (painted c (regular white))
  (temperature c cold) (shape c cylindrical) (is-object c)
  (has-hole b (8 mm) orientation-4) (painted b (water-res white))
  (surface-condition b smooth) (temperature b cold)
  (shape b undetermined) (is-object b) (painted a (regular white))
  (temperature a cold) (shape a undetermined) (is-object a))
```

## C.2 Experimental Results

The experiments in this domain were run in CMU Common Lisp on a IBM RT Model 130 with 16 megabytes of memory. The first set of six tables below compares PRODIGY without any control knowledge, PRODIGY with a set of hand-code control rules, and PRODIGY with the abstractions generated by ALPINE. The second set of six tables below compares PRODIGY with the control rules produced by EBL [Minton, 1988a], PRODIGY with the control rules produced by STATIC [Etzioni, 1990], and PRODIGY with both the hand-code control rules and the abstractions produced by ALPINE. The first 100 problems in each set of tables are the test problems used in Minton's experiments [Minton, 1988a].

The entries in the table are defined as follows:

**Prob Num** The problem number.

**Time** Total CPU time used in solving the problem. A 600 CPU second time bound was imposed on all problems.

**Nodes** Total number of nodes searched in solving the problem.

**Len** Length of the solution found. Zero if no solution exists.

**ACT** Time required to create the abstraction hierarchy. This time is also included in the total CPU time for ALPINE.

**AbNodes** Nodes searched at each level in the hierarchy. Ordered from more abstract to less abstract levels.

**AbLen** Solution length found at each level in the hierarchy. Ordered from more abstract to less abstract levels.

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 0.7 | 19 | 8 | 2.2 | 19 | 8 | 2.7 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 2 | 1.1 | 34 | 0 | 0.1 | 1 | 0 | 1.7 | 4 | 0 | 1.0 | | |
| 3 | 0.2 | 6 | 2 | 0.3 | 6 | 2 | 1.7 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 4 | 0.4 | 10 | 4 | 0.8 | 10 | 4 | 2.4 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 5 | 0.4 | 15 | 6 | 1.7 | 15 | 6 | 2.3 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 6 | 0.5 | 21 | 0 | 0.0 | 1 | 0 | 1.3 | 6 | 0 | 0.7 | | |
| 7 | 0.2 | 8 | 3 | 0.5 | 8 | 3 | 2.0 | 10 | 3 | 0.7 | 2,0,4,2,2 | 0,0,1,1,1 |
| 8 | 1.1 | 34 | 0 | 0.1 | 1 | 0 | 1.6 | 4 | 0 | 0.9 | | |
| 9 | 0.3 | 10 | 4 | 0.9 | 10 | 4 | 1.8 | 10 | 4 | 0.7 | 6,0,4 | 2,0,2 |
| 10 | 0.6 | 19 | 8 | 3.0 | 19 | 8 | 2.5 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 11 | 0.1 | 2 | 0 | 0.1 | 1 | 0 | 0.7 | 2 | 0 | 0.5 | | |
| 12 | 0.5 | 15 | 6 | 0.4 | 6 | 2 | 2.4 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 13 | 0.2 | 8 | 3 | 0.5 | 8 | 3 | 1.9 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 14 | 0.4 | 12 | 5 | 1.2 | 12 | 5 | 2.1 | 12 | 5 | 0.7 | 6,2,4 | 2,1,2 |
| 15 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 1.6 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 16 | 22.6 | 718 | 0 | 119.3 | 424 | 0 | 3.0 | 28 | 0 | 0.8 | | |
| 17 | 0.6 | 20 | 9 | 2.3 | 16 | 7 | 3.4 | 22 | 9 | 1.0 | 4,8,2,8 | 1,3,1,4 |
| 18 | 0.9 | 36 | 0 | 0.1 | 1 | 0 | 1.7 | 15 | 0 | 0.8 | | |
| 19 | 5.6 | 178 | 0 | 0.1 | 1 | 0 | 2.7 | 28 | 0 | 1.0 | | |
| 20 | 0.8 | 36 | 0 | 0.1 | 1 | 0 | 1.2 | 2 | 0 | 0.8 | | |
| 21 | 1.4 | 44 | 0 | 0.1 | 1 | 0 | 1.9 | 6 | 0 | 1.1 | | |
| 22 | 0.2 | 8 | 3 | 0.5 | 8 | 3 | 2.0 | 10 | 3 | 0.8 | 2,4,2,2 | 0,1,1,1 |
| 23 | 0.1 | 6 | 2 | 0.3 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 24 | 0.7 | 16 | 6 | 2.4 | 16 | 7 | 4.1 | 25 | 6 | 1.1 | 4,4,8,3,6 | 1,1,1,0,3 |
| 25 | 16.1 | 642 | 0 | 0.1 | 1 | 0 | 2.3 | 15 | 0 | 1.2 | | |
| 26 | 0.7 | 20 | 8 | 3.1 | 20 | 8 | 2.9 | 23 | 8 | 0.6 | 6,10,7 | 1,4,3 |
| 27 | 0.4 | 10 | 4 | 0.9 | 10 | 4 | 2.5 | 10 | 4 | 0.9 | 6,0,4 | 2,0,2 |
| 28 | 112.9 | 3154 | 0 | 0.1 | 1 | 0 | 3.8 | 40 | 0 | 1.3 | | |
| 29 | 0.4 | 12 | 5 | 1.3 | 12 | 5 | 2.7 | 12 | 5 | 0.9 | 6,2,4 | 2,1,2 |
| 30 | 72.7 | 2450 | 9 | 4.2 | 21 | 9 | 8.4 | 60 | 9 | 1.4 | 12,0,26,14,8 | 2,0,2,1,4 |
| 31 | 600.0 | 9224 | — | 0.2 | 1 | 0 | 34.8 | 389 | 0 | 1.9 | | |
| 32 | 600.0 | 9061 | — | 0.1 | 1 | 0 | 69.2 | 837 | 0 | 1.9 | | |
| 33 | 9.8 | 342 | 0 | 0.1 | 1 | 0 | 2.2 | 13 | 0 | 1.1 | | |
| 34 | 161.0 | 3667 | 10 | 1.6 | 12 | 5 | 6.1 | 54 | 5 | 1.2 | 48,2,4 | 2,1,2 |
| 35 | 1.6 | 54 | 5 | 0.9 | 12 | 5 | 3.5 | 24 | 5 | 0.8 | 6,0,9,5,4 | 1,0,1,1,2 |
| 36 | 0.9 | 16 | 7 | 3.1 | 16 | 7 | 4.3 | 18 | 7 | 1.0 | 4,0,6,2,6 | 1,0,2,1,3 |
| 37 | 2.0 | 48 | 8 | 4.9 | 18 | 8 | 5.8 | 22 | 8 | 2.1 | 2,10,4,6 | 0,3,2,3 |
| 38 | 4.5 | 102 | 8 | 3.6 | 18 | 8 | 5.0 | 22 | 8 | 1.6 | 14,0,8 | 4,0,4 |
| 39 | 129.7 | 2657 | 10 | 5.5 | 22 | 10 | 4.5 | 22 | 10 | 1.2 | 10,4,8 | 4,2,4 |
| 40 | 3.6 | 156 | 0 | 0.0 | 1 | 0 | 1.8 | 15 | 0 | 0.8 | | |
| 41 | 600.0 | 11043 | — | 0.1 | 1 | 0 | 3.7 | 10 | 0 | 2.0 | | |
| 42 | 1.4 | 22 | 9 | 2.8 | 16 | 7 | 4.4 | 21 | 6 | 1.2 | 4,8,3,6 | 1,2,0,3 |
| 43 | 600.0 | 8019 | — | 600.0 | 772 | — | 330.3 | 3891 | 0 | 2.1 | | |
| 44 | 13.8 | 474 | 0 | 0.1 | 1 | 0 | 2.9 | 16 | 0 | 0.9 | | |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 45 | 0.2 | 6 | 2 | 0.5 | 6 | 2 | 1.7 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 46 | 600.0 | 9432 | — | 0.2 | 1 | 0 | 41.6 | 466 | 0 | 2.1 | | |
| 47 | 0.6 | 14 | 6 | 2.1 | 14 | 6 | 2.6 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 48 | 0.2 | 6 | 2 | 0.5 | 6 | 2 | 2.1 | 6 | 2 | 0.9 | 4,0,2 | 1,0,1 |
| 49 | 600.0 | 8472 | — | 0.1 | 1 | 0 | 107.8 | 1188 | 0 | 2.1 | | |
| 50 | 103.1 | 3891 | 0 | 0.1 | 1 | 0 | 1.9 | 6 | 0 | 1.1 | | |
| 51 | 351.5 | 6702 | 13 | 12.0 | 31 | 14 | 15.0 | 115 | 13 | 2.3 | 8,69,26,12 | 2,4,1,6 |
| 52 | 0.2 | 8 | 3 | 0.6 | 8 | 3 | 1.9 | 10 | 3 | 0.8 | 2,4,2,2 | 0,1,1,1 |
| 53 | 600.0 | 8957 | — | 140.5 | 168 | 21 | 8.4 | 44 | 14 | 2.1 | 6,20,3,15 | 2,5,0,7 |
| 54 | 3.8 | 86 | 13 | 11.6 | 30 | 13 | 18.4 | 192 | 13 | 2.0 | 6,0,100,74,12 | 1,0,4,3,5 |
| 55 | 14.3 | 414 | 6 | 2.2 | 14 | 6 | 3.7 | 20 | 6 | 1.1 | 14,0,6 | 3,0,3 |
| 56 | 600.0 | 8945 | — | 0.1 | 1 | 0 | 16.0 | 159 | 0 | 2.2 | | |
| 57 | 1.4 | 15 | 6 | 1.9 | 15 | 6 | 3.6 | 17 | 6 | 0.9 | 6,0,4,0,7 | 2,0,1,0,3 |
| 58 | 600.0 | 13447 | — | 15.8 | 31 | 14 | 15.1 | 120 | 14 | 2.2 | 14,16,49,27,14 | 2,1,3,2,6 |
| 59 | 0.4 | 10 | 4 | 1.0 | 10 | 4 | 2.3 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 60 | 36.3 | 1421 | 0 | 0.1 | 1 | 0 | 3.1 | 7 | 0 | 1.8 | | |
| 61 | 600.0 | 12675 | — | 0.1 | 1 | 0 | 6.0 | 67 | 0 | 1.3 | | |
| 62 | 0.5 | 12 | 5 | 1.4 | 12 | 5 | 2.3 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 63 | 0.9 | 14 | 6 | 2.0 | 14 | 6 | 3.4 | 16 | 6 | 0.9 | 4,0,6,0,6 | 1,0,2,0,3 |
| 64 | 4.8 | 107 | 7 | 2.5 | 16 | 7 | 3.7 | 20 | 7 | 1.1 | 12,2,6 | 3,1,3 |
| 65 | 0.9 | 37 | 0 | 0.1 | 1 | 0 | 1.2 | 2 | 0 | 0.8 | | |
| 66 | 176.0 | 3524 | 9 | 5.1 | 20 | 9 | 5.8 | 30 | 9 | 1.8 | 20,2,8 | 4,1,4 |
| 67 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 2.2 | 8 | 2 | 0.8 | 4,0,2,0,2 | 1,0,0,0,1 |
| 68 | 600.0 | 10374 | — | 0.1 | 1 | 0 | 246.1 | 2713 | 0 | 2.3 | | |
| 69 | 600.0 | 9138 | — | 12.7 | 29 | 13 | 8.4 | 53 | 12 | 1.9 | 38,4,11 | 5,2,5 |
| 70 | 0.9 | 10 | 4 | 1.0 | 10 | 4 | 2.7 | 12 | 4 | 0.8 | 4,0,4,0,4 | 1,0,1,0,2 |
| 71 | 6.5 | 99 | 17 | 15.0 | 32 | 15 | 12.8 | 64 | 15 | 2.5 | 8,14,18,12,12 | 2,2,2,3,6 |
| 72 | 0.8 | 16 | 7 | 3.6 | 16 | 7 | 4.2 | 18 | 7 | 1.2 | 2,0,8,2,6 | 0,0,3,1,3 |
| 73 | 3.5 | 34 | 14 | 194.5 | 198 | 23 | 6.7 | 29 | 10 | 1.6 | 6,0,11,0,12 | 2,0,3,0,5 |
| 74 | 0.8 | 16 | 7 | 1.7 | 16 | 7 | 3.6 | 18 | 7 | 1.0 | 4,0,6,2,6 | 1,0,2,1,3 |
| 75 | 0.5 | 12 | 5 | 1.4 | 12 | 5 | 2.8 | 12 | 5 | 0.9 | 6,2,4 | 2,1,2 |
| 76 | 125.5 | 2702 | 10 | 4.1 | 22 | 10 | 5.6 | 40 | 10 | 1.0 | 6,8,18,8 | 1,1,4,4 |
| 77 | 1.1 | 21 | 9 | 5.0 | 21 | 9 | 4.4 | 21 | 9 | 1.1 | 11,2,8 | 4,1,4 |
| 78 | 600.0 | 12547 | — | 0.1 | 1 | 0 | 8.2 | 104 | 0 | 1.8 | | |
| 79 | 7.3 | 153 | 14 | 10.9 | 31 | 14 | 6.4 | 29 | 11 | 1.9 | 16,2,11 | 5,1,5 |
| 80 | 600.0 | 15201 | — | 10.8 | 32 | 15 | 11.7 | 100 | 15 | 1.7 | 6,8,45,29,12 | 1,1,4,3,6 |
| 81 | 600.0 | 12475 | — | 0.1 | 1 | 0 | 5.6 | 44 | 0 | 1.8 | | |
| 82 | 121.3 | 3749 | 14 | 8.5 | 29 | 13 | 15.1 | 97 | 12 | 2.1 | 8,20,34,25,10 | 2,2,1,2,5 |
| 83 | 1.6 | 15 | 6 | 2.8 | 15 | 6 | 4.9 | 19 | 6 | 1.4 | 6,4,2,0,7 | 2,1,0,0,3 |
| 84 | 600.0 | 12795 | — | 0.1 | 1 | 0 | 71.2 | 779 | 0 | 2.2 | | |
| 85 | 600.0 | 7636 | — | 0.1 | 1 | 0 | 122.3 | 1413 | 0 | 2.3 | | |
| 86 | 1.4 | 19 | 8 | 4.6 | 19 | 8 | 4.7 | 21 | 8 | 1.3 | 4,0,8,0,9 | 1,0,3,0,4 |
| 87 | 1.4 | 38 | 0 | 0.1 | 1 | 0 | 3.2 | 7 | 0 | 1.7 | | |
| 88 | 3.2 | 27 | 12 | 10.7 | 24 | 10 | 7.1 | 26 | 10 | 2.2 | 6,0,8,0,12 | 2,0,3,0,5 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 89 | 9.1 | 184 | 9 | 2.9 | 16 | 7 | 4.6 | 23 | 6 | 1.3 | 4,10,3,6 | 1,2,0,3 |
| 90 | 600.0 | 11870 | — | 0.2 | 1 | 0 | 5.1 | 9 | 0 | 2.9 | | |
| 91 | 6.3 | 45 | 19 | 21.8 | 39 | 17 | 11.9 | 57 | 15 | 2.4 | 8,4,20,8,17 | 3,1,3,1,7 |
| 92 | 600.0 | 12545 | — | 0.2 | 1 | 0 | 346.1 | 4960 | 0 | 3.8 | | |
| 93 | 600.0 | 10993 | — | 0.2 | 1 | 0 | 189.6 | 1971 | 0 | 3.0 | | |
| 94 | 1.1 | 19 | 8 | 0.6 | 6 | 2 | 3.1 | 19 | 8 | 0.7 | 4,8,7 | 1,4,3 |
| 95 | 600.0 | 7299 | — | 22.4 | 36 | 17 | 26.1 | 223 | 17 | 2.8 | 6,198,2,17 | 2,6,1,8 |
| 96 | 7.3 | 169 | 18 | 28.1 | 43 | 18 | 21.6 | 190 | 16 | 1.4 | 6,26,142,16 | 1,2,6,7 |
| 97 | 600.0 | 13117 | — | 59.4 | 48 | 23 | 600.0 | 7796 | — | 3.6 | | |
| 98 | 4.7 | 35 | 15 | 18.1 | 34 | 16 | 11.6 | 49 | 15 | 2.6 | 4,8,16,5,16 | 1,3,3,1,7 |
| 99 | 3.3 | 37 | 17 | 14.5 | 33 | 15 | 8.1 | 35 | 15 | 2.1 | 4,12,6,13 | 1,5,3,6 |
| 100 | 1.9 | 16 | 7 | 3.8 | 16 | 7 | 5.2 | 18 | 7 | 1.5 | 4,6,2,6 | 1,2,1,3 |
| 101 | 1.1 | 19 | 8 | 5.2 | 19 | 8 | 3.8 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 102 | 0.1 | 2 | 0 | 0.1 | 1 | 0 | 0.8 | 2 | 0 | 0.5 | | |
| 103 | 0.3 | 8 | 3 | 0.6 | 8 | 3 | 1.6 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 104 | 0.5 | 6 | 2 | 0.4 | 6 | 2 | 1.6 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 105 | 0.2 | 8 | 3 | 0.5 | 8 | 3 | 1.5 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 106 | 0.7 | 15 | 6 | 2.5 | 15 | 6 | 3.4 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 107 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.5 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 108 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.5 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 109 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.6 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 110 | 0.9 | 19 | 8 | 4.3 | 19 | 8 | 3.1 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 111 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 1.2 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 112 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.6 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 113 | 0.9 | 19 | 8 | 3.3 | 19 | 8 | 3.1 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 114 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 1.5 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 115 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 2.1 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 116 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 1.2 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 117 | 1.0 | 19 | 8 | 0.6 | 6 | 2 | 2.8 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 118 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.5 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 119 | 0.1 | 2 | 0 | 0.1 | 1 | 0 | 0.8 | 2 | 0 | 0.5 | | |
| 120 | 0.3 | 8 | 3 | 0.5 | 8 | 3 | 1.6 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 121 | 0.9 | 19 | 8 | 3.9 | 19 | 8 | 2.8 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 122 | 0.9 | 19 | 8 | 4.1 | 19 | 8 | 3.1 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 123 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.5 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 124 | 0.2 | 6 | 2 | 0.4 | 6 | 2 | 1.2 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 125 | 0.4 | 6 | 2 | 0.4 | 6 | 2 | 1.6 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 126 | 2.8 | 82 | 0 | 0.1 | 1 | 0 | 2.1 | 15 | 0 | 0.8 | | |
| 127 | 0.7 | 10 | 4 | 1.1 | 10 | 4 | 3.5 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 128 | 0.6 | 10 | 4 | 1.1 | 10 | 4 | 2.7 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 129 | 0.5 | 12 | 5 | 1.0 | 12 | 5 | 2.6 | 14 | 5 | 0.7 | 4,4,2,4 | 1,1,1,2 |
| 130 | 1.1 | 23 | 10 | 5.3 | 23 | 10 | 5.1 | 23 | 10 | 0.9 | 10,4,9 | 4,2,4 |
| 131 | 0.7 | 15 | 6 | 1.9 | 15 | 6 | 2.7 | 15 | 6 | 0.7 | 6,4,5 | 2,2,2 |
| 132 | 128.4 | 2837 | 9 | 1.2 | 10 | 4 | 3.9 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 133 | 1.1 | 16 | 6 | 1.5 | 12 | 5 | 3.4 | 15 | 4 | 0.8 | 4,0,4,3,4 | 1,0,1,0,2 |
| 134 | 0.5 | 10 | 4 | 1.1 | 10 | 4 | 2.1 | 10 | 4 | 0.7 | 6,0,4 | 2,0,2 |
| 135 | 0.8 | 10 | 4 | 1.0 | 10 | 4 | 2.8 | 12 | 4 | 0.7 | 4,0,4,0,4 | 1,0,1,0,2 |
| 136 | 0.6 | 10 | 4 | 1.1 | 10 | 4 | 2.7 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 137 | 1.0 | 12 | 5 | 1.3 | 12 | 5 | 3.0 | 14 | 5 | 0.8 | 4,0,4,2,4 | 1,0,1,1,2 |
| 138 | 0.8 | 10 | 4 | 1.0 | 10 | 4 | 2.7 | 12 | 4 | 0.7 | 4,0,4,0,4 | 1,0,1,0,2 |
| 139 | 0.6 | 14 | 6 | 1.4 | 14 | 6 | 2.9 | 16 | 6 | 0.8 | 4,4,4,4 | 1,1,2,2 |
| 140 | 0.8 | 15 | 6 | 2.7 | 15 | 6 | 4.5 | 15 | 6 | 1.0 | 8,0,7 | 3,0,3 |
| 141 | 0.6 | 14 | 6 | 1.3 | 14 | 6 | 2.9 | 16 | 6 | 0.8 | 4,4,4,4 | 1,1,2,2 |
| 142 | 1.1 | 12 | 5 | 1.4 | 12 | 5 | 3.0 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 143 | 1.2 | 11 | 4 | 1.2 | 11 | 4 | 2.6 | 11 | 4 | 0.8 | 6,0,5 | 2,0,2 |
| 144 | 1.0 | 10 | 4 | 1.0 | 10 | 4 | 2.8 | 12 | 4 | 0.7 | 4,0,4,0,4 | 1,0,1,0,2 |
| 145 | 600.0 | 12243 | — | 600.0 | 1949 | — | 3.6 | 34 | 0 | 0.8 | | |
| 146 | 0.4 | 10 | 4 | 1.2 | 10 | 4 | 2.7 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 147 | 0.9 | 10 | 4 | 1.2 | 10 | 4 | 3.2 | 12 | 4 | 0.8 | 4,0,4,0,4 | 1,0,1,0,2 |
| 148 | 0.5 | 10 | 4 | 0.9 | 10 | 4 | 2.6 | 12 | 4 | 0.7 | 4,4,0,4 | 1,1,0,2 |
| 149 | 0.8 | 10 | 4 | 1.0 | 10 | 4 | 2.8 | 12 | 4 | 0.7 | 4,0,4,0,4 | 1,0,1,0,2 |
| 150 | 0.7 | 12 | 5 | 1.4 | 12 | 5 | 2.3 | 12 | 5 | 0.6 | 6,2,4 | 2,1,2 |
| 151 | 1.1 | 21 | 9 | 5.9 | 21 | 9 | 3.7 | 17 | 7 | 1.1 | 8,2,7 | 3,1,3 |
| 152 | 4.1 | 74 | 6 | 2.2 | 14 | 6 | 5.0 | 16 | 6 | 1.3 | 4,6,0,6 | 1,2,0,3 |
| 153 | 2.1 | 17 | 7 | 2.9 | 17 | 7 | 4.5 | 19 | 7 | 1.1 | 6,0,4,2,7 | 2,0,1,1,3 |
| 154 | 4.2 | 75 | 6 | 2.4 | 15 | 6 | 5.2 | 17 | 6 | 1.2 | 6,4,0,7 | 2,1,0,3 |
| 155 | 0.9 | 18 | 8 | 2.0 | 14 | 6 | 3.8 | 20 | 8 | 1.0 | 4,8,0,8 | 1,3,0,4 |
| 156 | 1.2 | 14 | 6 | 2.4 | 14 | 6 | 4.1 | 18 | 6 | 1.1 | 4,4,4,0,6 | 1,1,1,0,3 |
| 157 | 1.0 | 18 | 8 | 2.9 | 18 | 8 | 3.8 | 20 | 8 | 1.0 | 4,6,4,6 | 1,2,2,3 |
| 158 | 60.2 | 1722 | 0 | 0.1 | 1 | 0 | 3.3 | 28 | 0 | 1.1 | | |
| 159 | 1.0 | 16 | 7 | 2.3 | 16 | 7 | 4.4 | 18 | 7 | 1.0 | 6,4,2,6 | 2,1,1,3 |
| 160 | 0.8 | 14 | 6 | 2.3 | 14 | 6 | 3.6 | 16 | 6 | 1.0 | 4,6,0,6 | 1,2,0,3 |
| 161 | 1.5 | 15 | 6 | 2.5 | 15 | 6 | 4.2 | 17 | 6 | 1.1 | 6,0,4,0,7 | 2,0,1,0,3 |
| 162 | 1.1 | 16 | 7 | 3.3 | 16 | 7 | 4.0 | 16 | 7 | 1.2 | 8,2,6 | 3,1,3 |
| 163 | 1.9 | 22 | 10 | 1.8 | 14 | 6 | 3.9 | 18 | 6 | 1.0 | 4,4,4,0,6 | 1,1,1,0,3 |
| 164 | 1.8 | 22 | 9 | 2.9 | 16 | 7 | 4.2 | 21 | 6 | 1.0 | 4,0,8,3,6 | 1,0,2,0,3 |
| 165 | 600.0 | 13813 | — | 0.1 | 1 | 0 | 8.1 | 90 | 0 | 1.3 | | |
| 166 | 2.1 | 29 | 13 | 2.7 | 16 | 7 | 5.3 | 16 | 7 | 1.3 | 8,2,6 | 3,1,3 |
| 167 | 0.7 | 14 | 6 | 2.1 | 14 | 6 | 3.4 | 16 | 6 | 1.0 | 4,6,0,6 | 1,2,0,3 |
| 168 | 600.0 | 11187 | — | 600.0 | 919 | — | 6.4 | 36 | 13 | 1.1 | 22,2,12 | 6,1,6 |
| 169 | 1.2 | 16 | 6 | 3.0 | 16 | 7 | 4.2 | 21 | 6 | 1.0 | 4,0,8,3,6 | 1,0,2,0,3 |
| 170 | 3.7 | 83 | 9 | 2.1 | 16 | 7 | 5.6 | 34 | 7 | 1.1 | 6,8,9,5,6 | 1,1,1,1,3 |
| 171 | 1.8 | 28 | 13 | 10.3 | 28 | 13 | 5.7 | 26 | 12 | 1.2 | 12,4,10 | 5,2,5 |
| 172 | 130.6 | 5092 | 0 | 0.1 | 1 | 0 | 1.9 | 4 | 0 | 1.1 | | |
| 173 | 1.1 | 18 | 8 | 3.8 | 18 | 8 | 3.6 | 18 | 7 | 1.0 | 4,6,2,6 | 1,2,1,3 |
| 174 | 600.0 | 10297 | — | 6.5 | 22 | 10 | 5.7 | 24 | 10 | 1.4 | 4,6,4,10 | 1,2,2,5 |
| 175 | 1.8 | 15 | 6 | 2.7 | 15 | 6 | 4.4 | 17 | 6 | 1.1 | 6,0,4,0,7 | 2,0,1,0,3 |
| 176 | 1.7 | 25 | 11 | 5.3 | 20 | 9 | 5.4 | 22 | 9 | 1.3 | 4,8,2,8 | 1,3,1,4 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 177 | 2.4 | 18 | 8 | 4.6 | 18 | 8 | 5.5 | 20 | 8 | 1.4 | 4,8,0,8 | 1,3,0,4 |
| 178 | 2.1 | 20 | 9 | 5.0 | 20 | 9 | 6.3 | 24 | 9 | 1.5 | 4,4,6,2,8 | 1,1,2,1,4 |
| 179 | 600.0 | 13373 | — | 0.1 | 1 | 0 | 34.9 | 443 | 0 | 1.5 | | |
| 180 | 2.7 | 24 | 10 | 5.8 | 21 | 9 | 6.6 | 26 | 8 | 1.3 | 6,0,8,3,9 | 2,0,2,0,4 |
| 181 | 164.7 | 5150 | 9 | 5.8 | 20 | 9 | 7.0 | 39 | 9 | 1.4 | 6,8,12,5,8 | 1,1,2,1,4 |
| 182 | 1.2 | 19 | 8 | 5.1 | 19 | 8 | 4.6 | 21 | 8 | 1.4 | 4,8,0,9 | 1,3,0,4 |
| 183 | 12.0 | 227 | 15 | 7.2 | 23 | 10 | 6.2 | 23 | 10 | 1.6 | 10,4,9 | 4,2,4 |
| 184 | 3.2 | 39 | 18 | 11.2 | 26 | 12 | 8.6 | 24 | 11 | 1.7 | 12,2,10 | 5,1,5 |
| 185 | 4.0 | 76 | 10 | 7.8 | 22 | 10 | 7.2 | 41 | 10 | 1.6 | 6,8,12,7,8 | 1,1,2,2,4 |
| 186 | 1.7 | 25 | 11 | 6.2 | 25 | 11 | 5.1 | 27 | 11 | 1.4 | 6,6,6,9 | 2,2,3,4 |
| 187 | 48.2 | 1363 | 15 | 11.9 | 33 | 15 | 20.7 | 176 | 15 | 1.7 | 6,98,60,12 | 1,5,3,6 |
| 188 | 1.2 | 20 | 9 | 4.5 | 20 | 9 | 4.8 | 22 | 9 | 1.5 | 4,8,2,8 | 1,3,1,4 |
| 189 | 2.9 | 21 | 9 | 4.6 | 21 | 9 | 5.4 | 23 | 9 | 1.3 | 6,0,6,2,9 | 2,0,2,1,4 |
| 190 | 1.4 | 18 | 8 | 3.8 | 18 | 8 | 5.1 | 22 | 8 | 1.4 | 4,4,6,0,8 | 1,1,2,0,4 |
| 191 | 350.4 | 7170 | 16 | 16.6 | 33 | 14 | 7.8 | 37 | 16 | 1.6 | 17,4,16 | 7,2,7 |
| 192 | 600.0 | 13802 | — | 0.1 | 1 | 0 | 4.2 | 40 | 0 | 1.4 | | |
| 193 | 2.6 | 23 | 10 | 6.6 | 23 | 10 | 5.9 | 25 | 10 | 1.5 | 6,0,6,4,9 | 2,0,2,2,4 |
| 194 | 600.0 | 12760 | — | 11.9 | 29 | 13 | 6.5 | 26 | 12 | 1.5 | 14,0,12 | 6,0,6 |
| 195 | 1.6 | 20 | 9 | 5.3 | 20 | 9 | 4.9 | 18 | 8 | 1.6 | 10,0,8 | 4,0,4 |
| 196 | 2.5 | 31 | 14 | 10.9 | 26 | 12 | 7.0 | 31 | 14 | 1.5 | 16,0,15 | 7,0,7 |
| 197 | 3.3 | 30 | 13 | 5.0 | 20 | 9 | 6.2 | 22 | 9 | 1.8 | 4,8,2,8 | 1,3,1,4 |
| 198 | 1.7 | 18 | 8 | 3.9 | 18 | 8 | 5.1 | 22 | 8 | 1.4 | 4,4,6,0,8 | 1,1,2,0,4 |
| 199 | 2.9 | 24 | 11 | 3.9 | 21 | 9 | 6.6 | 23 | 9 | 1.3 | 6,0,6,2,9 | 2,0,2,1,4 |
| 200 | 2.1 | 36 | 15 | 20.5 | 36 | 15 | 7.6 | 36 | 15 | 1.6 | 17,6,13 | 6,3,6 |
| 201 | 13.0 | 308 | 19 | 17.3 | 33 | 15 | 17.2 | 123 | 15 | 1.8 | 8,0,60,43,12 | 2,0,4,3,6 |
| 202 | 600.0 | 14277 | — | 0.1 | 1 | 0 | 13.6 | 172 | 0 | 1.8 | | |
| 203 | 61.8 | 1173 | 23 | 600.0 | 405 | — | 6.4 | 23 | 10 | 1.7 | 12,0,11 | 5,0,5 |
| 204 | 600.0 | 10071 | — | 7.0 | 25 | 11 | 7.1 | 32 | 11 | 1.8 | 6,13,2,11 | 2,3,1,5 |
| 205 | 3.9 | 28 | 12 | 8.9 | 28 | 12 | 8.0 | 32 | 12 | 1.9 | 6,4,6,4,12 | 2,1,2,2,5 |
| 206 | 600.0 | 10337 | — | 600.0 | 494 | — | 600.0 | 6928 | — | 2.1 | | |
| 207 | 3.2 | 27 | 12 | 9.4 | 23 | 10 | 7.9 | 27 | 10 | 1.9 | 6,4,6,0,11 | 2,1,2,0,5 |
| 208 | 2.6 | 37 | 17 | 27.1 | 37 | 17 | 9.2 | 37 | 17 | 2.1 | 17,6,14 | 7,3,7 |
| 209 | 2.8 | 27 | 11 | 9.9 | 27 | 12 | 8.1 | 38 | 11 | 1.8 | 6,4,12,5,11 | 2,1,2,1,5 |
| 210 | 600.0 | 9205 | — | 600.0 | 649 | — | 70.2 | 785 | 0 | 2.2 | | |
| 211 | 2.6 | 27 | 12 | 8.7 | 27 | 12 | 6.5 | 31 | 12 | 1.8 | 4,4,8,4,11 | 1,1,3,2,5 |
| 212 | 3.3 | 27 | 12 | 15.5 | 27 | 12 | 7.7 | 29 | 12 | 1.7 | 4,0,10,4,11 | 1,0,4,2,5 |
| 213 | 2.1 | 29 | 13 | 10.2 | 24 | 11 | 6.5 | 31 | 13 | 1.8 | 6,10,2,13 | 2,4,1,6 |
| 214 | 12.6 | 369 | 17 | 16.6 | 32 | 15 | 16.0 | 129 | 15 | 1.9 | 6,8,60,43,12 | 1,1,4,3,6 |
| 215 | 600.0 | 13867 | — | 0.1 | 1 | 0 | 4.9 | 39 | 0 | 1.7 | | |
| 216 | 4.1 | 83 | 19 | 111.7 | 83 | 19 | 8.8 | 41 | 17 | 2.0 | 18,6,17 | 7,3,7 |
| 217 | 2.4 | 26 | 11 | 7.3 | 26 | 11 | 5.5 | 28 | 11 | 1.4 | 6,8,2,12 | 2,3,1,5 |
| 218 | 600.0 | 9850 | — | 0.1 | 1 | 0 | 35.6 | 401 | 0 | 2.0 | | |
| 219 | 600.0 | 12472 | — | 6.8 | 24 | 11 | 17.4 | 143 | 11 | 1.9 | 6,98,29,10 | 1,4,1,5 |
| 220 | 600.0 | 11590 | — | 25.6 | 36 | 16 | 64.5 | 614 | 16 | 2.0 | 6,456,136,16 | 1,6,2,7 |

| Prob | Prodigy | | | Prodigy + HCR | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 221 | 600.0 | 13515 | — | 0.1 | 1 | 0 | 7.3 | 67 | 0 | 1.9 | | |
| 222 | 600.0 | 10365 | — | 600.0 | 755 | — | 106.1 | 1203 | 0 | 2.0 | | |
| 223 | 600.0 | 9819 | — | 21.5 | 36 | 17 | 70.3 | 658 | 17 | 2.1 | 6,448,190,14 | 1,6,3,7 |
| 224 | 4.8 | 34 | 16 | 15.6 | 30 | 14 | 7.8 | 36 | 16 | 1.6 | 4,0,14,4,14 | 1,0,6,2,7 |
| 225 | 600.0 | 10396 | — | 600.0 | 509 | — | 12.8 | 65 | 16 | 1.9 | 4,42,3,16 | 1,7,0,8 |
| 226 | 33.1 | 652 | 19 | 600.0 | 293 | — | 11.3 | 37 | 17 | 2.4 | 16,6,15 | 7,3,7 |
| 227 | 600.0 | 9710 | — | 0.2 | 1 | 0 | 49.4 | 523 | 0 | 2.3 | | |
| 228 | 600.0 | 10605 | — | 0.1 | 1 | 0 | 225.0 | 2847 | 0 | 2.1 | | |
| 229 | 600.0 | 13842 | — | 0.2 | 1 | 0 | 6.4 | 43 | 0 | 2.3 | | |
| 230 | 600.0 | 8883 | — | 0.2 | 1 | 0 | 600.0 | 6735 | — | 2.4 | | |
| 231 | 600.0 | 16681 | — | 0.1 | 1 | 0 | 9.3 | 83 | 0 | 2.1 | | |
| 232 | 4.2 | 39 | 17 | 13.9 | 29 | 13 | 9.2 | 45 | 12 | 2.2 | 4,12,12,3,14 | 1,3,2,0,6 |
| 233 | 600.0 | 12603 | — | 0.2 | 1 | 0 | 51.1 | 547 | 0 | 2.3 | | |
| 234 | 600.0 | 9767 | — | 11.6 | 29 | 13 | 20.0 | 164 | 13 | 2.3 | 4,145,2,13 | 1,5,1,6 |
| 235 | 4.8 | 29 | 13 | 16.6 | 29 | 13 | 8.2 | 33 | 13 | 2.2 | 6,4,8,2,13 | 2,1,3,1,6 |
| 236 | 26.7 | 487 | 16 | 15.2 | 30 | 14 | 46.2 | 412 | 14 | 2.2 | 6,299,95,12 | 1,5,2,6 |
| 237 | 6.3 | 87 | 17 | 16.1 | 35 | 15 | 8.8 | 39 | 15 | 2.4 | 4,14,6,15 | 1,5,3,6 |
| 238 | 600.0 | 11868 | — | 30.2 | 45 | 20 | 10.8 | 47 | 20 | 2.4 | 4,16,8,19 | 1,7,4,8 |
| 239 | 600.0 | 9083 | — | 600.0 | 718 | — | 197.9 | 2167 | 0 | 2.3 | | |
| 240 | 600.0 | 9374 | — | 30.4 | 41 | 19 | 72.1 | 700 | 19 | 2.3 | 12,616,56,16 | 2,6,3,8 |
| 241 | 2.5 | 32 | 15 | 15.6 | 32 | 15 | 7.4 | 34 | 15 | 2.1 | 6,10,6,12 | 2,4,3,6 |
| 242 | 600.0 | 9392 | — | 600.0 | 336 | — | 600.0 | 6802 | — | 2.4 | | |
| 243 | 600.0 | 8774 | — | 600.0 | 424 | — | 272.5 | 3307 | 0 | 2.3 | | |
| 244 | 600.0 | 12316 | — | 507.8 | 331 | 29 | 8.5 | 45 | 17 | 1.7 | 4,4,16,2,19 | 1,1,6,1,8 |
| 245 | 600.0 | 11754 | — | 0.2 | 1 | 0 | 141.0 | 1790 | 0 | 2.1 | | |
| 246 | 600.0 | 13099 | — | 0.2 | 1 | 0 | 8.3 | 67 | 0 | 2.2 | | |
| 247 | 600.0 | 14037 | — | 0.2 | 1 | 0 | 104.7 | 1124 | 0 | 2.9 | | |
| 248 | 600.0 | 10440 | — | 0.1 | 1 | 0 | 70.0 | 763 | 0 | 2.5 | | |
| 249 | 600.0 | 12685 | — | 0.1 | 1 | 0 | 28.5 | 411 | 0 | 2.1 | | |
| 250 | 319.1 | 5393 | 20 | 30.1 | 42 | 20 | 424.7 | 4099 | 20 | 2.5 | 6,3071,1006,16 | 1,7,4,8 |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 1.7 | 19 | 8 | 3.0 | 19 | 8 | 2.9 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 2 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 1.0 | 1 | 0 | 1.0 | | |
| 3 | 0.4 | 6 | 2 | 0.5 | 6 | 2 | 1.5 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 4 | 1.0 | 10 | 4 | 1.2 | 10 | 4 | 2.0 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 5 | 1.3 | 15 | 6 | 2.2 | 15 | 6 | 2.0 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 6 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 0.9 | 1 | 0 | 0.7 | | |
| 7 | 0.4 | 8 | 3 | 0.6 | 8 | 3 | 1.8 | 10 | 3 | 0.7 | 2,4,2,2 | 0,1,1,1 |
| 8 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.0 | 1 | 0 | 0.9 | | |
| 9 | 0.7 | 10 | 4 | 1.0 | 10 | 4 | 2.1 | 10 | 4 | 0.7 | 6,0,4 | 2,0,2 |
| 10 | 2.1 | 19 | 8 | 2.7 | 19 | 8 | 2.4 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 11 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 0.6 | 1 | 0 | 0.5 | | |
| 12 | 1.3 | 15 | 6 | 4.0 | 19 | 8 | 1.4 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 13 | 0.4 | 8 | 3 | 0.7 | 8 | 3 | 1.2 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 14 | 1.0 | 12 | 5 | 1.4 | 12 | 5 | 2.9 | 12 | 5 | 0.7 | 6,2,4 | 2,1,2 |
| 15 | 0.4 | 6 | 2 | 0.5 | 6 | 2 | 1.3 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 16 | 135.8 | 718 | 0 | 101.4 | 458 | 0 | 3.7 | 18 | 0 | 0.8 | | |
| 17 | 1.8 | 16 | 7 | 1.2 | 10 | 4 | 3.4 | 16 | 7 | 1.0 | 8,2,6 | 3,1,3 |
| 18 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 0.9 | 1 | 0 | 0.8 | | |
| 19 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 1.5 | 8 | 0 | 1.0 | | |
| 20 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 1.0 | 1 | 0 | 0.8 | | |
| 21 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.3 | 1 | 0 | 1.1 | | |
| 22 | 0.4 | 8 | 3 | 0.6 | 8 | 3 | 1.8 | 10 | 3 | 0.8 | 2,4,2,2 | 0,1,1,1 |
| 23 | 0.3 | 6 | 2 | 0.4 | 6 | 2 | 1.0 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 24 | 2.1 | 16 | 7 | 3.0 | 16 | 7 | 3.8 | 18 | 7 | 1.1 | 4,6,2,6 | 1,2,1,3 |
| 25 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 1.6 | 8 | 0 | 1.2 | | |
| 26 | 2.7 | 20 | 8 | 4.0 | 20 | 8 | 2.9 | 23 | 8 | 0.6 | 6,10,7 | 1,4,3 |
| 27 | 0.8 | 10 | 4 | 1.4 | 10 | 4 | 2.1 | 10 | 4 | 0.9 | 6,0,4 | 2,0,2 |
| 28 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.9 | 8 | 0 | 1.3 | | |
| 29 | 1.0 | 12 | 5 | 2.0 | 12 | 5 | 2.5 | 12 | 5 | 0.9 | 6,2,4 | 2,1,2 |
| 30 | 3.8 | 21 | 9 | 5.3 | 21 | 9 | 4.7 | 23 | 9 | 1.4 | 6,6,2,9 | 2,2,1,4 |
| 31 | 0.2 | 1 | 0 | 0.2 | 1 | 0 | 2.7 | 8 | 0 | 1.9 | | |
| 32 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.1 | 1 | 0 | 1.9 | | |
| 33 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.2 | 1 | 0 | 1.1 | | |
| 34 | 600.0 | 2578 | — | 7.6 | 24 | 10 | 3.4 | 12 | 5 | 1.2 | 6,2,4 | 2,1,2 |
| 35 | 1.1 | 12 | 5 | 1.4 | 12 | 5 | 2.2 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 36 | 2.2 | 14 | 6 | 3.4 | 16 | 7 | 3.9 | 18 | 7 | 1.0 | 4,6,2,6 | 1,2,1,3 |
| 37 | 2.5 | 16 | 7 | 5.8 | 18 | 8 | 7.2 | 20 | 8 | 2.1 | 2,8,4,6 | 0,3,2,3 |
| 38 | 23.8 | 102 | 8 | 5.0 | 18 | 8 | 5.9 | 18 | 8 | 1.6 | 10,0,8 | 4,0,4 |
| 39 | 3.9 | 22 | 10 | 7.5 | 22 | 10 | 6.0 | 22 | 10 | 1.2 | 10,4,8 | 4,2,4 |
| 40 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 1.1 | 8 | 0 | 0.8 | | |
| 41 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.2 | 1 | 0 | 2.0 | | |
| 42 | 2.7 | 14 | 6 | 4.0 | 16 | 7 | 4.0 | 18 | 7 | 1.2 | 4,6,2,6 | 1,2,1,3 |
| 43 | 600.0 | 1024 | — | 600.0 | 741 | — | 88.5 | 187 | 0 | 2.1 | | |
| 44 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.0 | 1 | 0 | 0.9 | | |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 45 | 0.4 | 6 | 2 | 0.6 | 6 | 2 | 1.3 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 46 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.4 | 1 | 0 | 2.1 | | |
| 47 | 1.3 | 12 | 5 | 2.3 | 14 | 6 | 2.4 | 12 | 5 | 0.8 | 6,2,4 | 2,1,2 |
| 48 | 0.4 | 6 | 2 | 0.6 | 6 | 2 | 1.7 | 6 | 2 | 0.9 | 4,0,2 | 1,0,1 |
| 49 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.3 | 1 | 0 | 2.1 | | |
| 50 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.2 | 1 | 0 | 1.1 | | |
| 51 | 9.7 | 29 | 13 | 17.0 | 31 | 14 | 9.6 | 33 | 14 | 2.3 | 6,10,4,13 | 2,4,2,6 |
| 52 | 0.6 | 8 | 3 | 0.6 | 8 | 3 | 2.3 | 10 | 3 | 0.8 | 2,4,2,2 | 0,1,1,1 |
| 53 | 10.9 | 30 | 14 | 13.1 | 26 | 12 | 8.4 | 32 | 12 | 2.1 | 6,13,0,13 | 2,4,0,6 |
| 54 | 9.1 | 30 | 13 | 15.5 | 30 | 13 | 10.0 | 32 | 13 | 2.0 | 4,10,6,12 | 1,4,3,5 |
| 55 | 51.0 | 414 | 6 | 3.0 | 14 | 6 | 3.5 | 14 | 6 | 1.1 | 8,0,6 | 3,0,3 |
| 56 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.8 | 8 | 0 | 2.2 | | |
| 57 | 2.5 | 15 | 6 | 3.6 | 15 | 6 | 2.9 | 17 | 6 | 0.9 | 6,4,0,7 | 2,1,0,3 |
| 58 | 11.4 | 31 | 14 | 17.1 | 31 | 14 | 12.3 | 33 | 14 | 2.2 | 6,10,4,13 | 2,4,2,6 |
| 59 | 0.9 | 10 | 4 | 1.3 | 10 | 4 | 2.2 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 60 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.1 | 4 | 0 | 1.8 | | |
| 61 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.4 | 21 | 0 | 1.3 | | |
| 62 | 1.0 | 10 | 4 | 1.7 | 12 | 5 | 2.2 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 63 | 2.0 | 14 | 6 | 2.9 | 14 | 6 | 3.4 | 16 | 6 | 0.9 | 4,6,0,6 | 1,2,0,3 |
| 64 | 7.7 | 51 | 7 | 3.2 | 14 | 6 | 3.7 | 16 | 7 | 1.1 | 8,2,6 | 3,1,3 |
| 65 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 0.9 | 1 | 0 | 0.8 | | |
| 66 | 600.0 | 2619 | — | 7.2 | 20 | 9 | 5.6 | 20 | 9 | 1.8 | 10,2,8 | 4,1,4 |
| 67 | 0.6 | 6 | 2 | 0.7 | 6 | 2 | 1.6 | 8 | 2 | 0.8 | 4,2,0,2 | 1,0,0,1 |
| 68 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 3.2 | 8 | 0 | 2.3 | | |
| 69 | 600.0 | 1390 | — | 15.7 | 29 | 13 | 10.1 | 27 | 12 | 1.9 | 12,4,11 | 5,2,5 |
| 70 | 1.3 | 10 | 4 | 1.7 | 10 | 4 | 2.2 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 71 | 11.7 | 30 | 14 | 18.6 | 32 | 15 | 12.7 | 35 | 15 | 2.5 | 6,10,6,13 | 2,4,3,6 |
| 72 | 2.0 | 14 | 6 | 3.5 | 16 | 7 | 5.9 | 18 | 7 | 1.2 | 2,8,2,6 | 0,3,1,3 |
| 73 | 8.3 | 24 | 10 | 8.6 | 24 | 10 | 10.5 | 29 | 10 | 1.6 | 6,11,0,12 | 2,3,0,5 |
| 74 | 1.9 | 16 | 7 | 2.6 | 16 | 7 | 3.2 | 18 | 7 | 1.0 | 4,6,2,6 | 1,2,1,3 |
| 75 | 1.2 | 12 | 5 | 2.2 | 12 | 5 | 2.4 | 12 | 5 | 0.9 | 6,2,4 | 2,1,2 |
| 76 | 4.2 | 22 | 10 | 9.9 | 49 | 10 | 3.7 | 24 | 10 | 1.0 | 4,4,8,8 | 1,1,4,4 |
| 77 | 3.8 | 21 | 9 | 6.5 | 21 | 9 | 4.5 | 21 | 9 | 1.1 | 11,2,8 | 4,1,4 |
| 78 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.4 | 8 | 0 | 1.8 | | |
| 79 | 35.1 | 117 | 11 | 10.0 | 25 | 11 | 9.0 | 25 | 11 | 1.9 | 12,2,11 | 5,1,5 |
| 80 | 6.5 | 26 | 12 | 10.0 | 26 | 12 | 11.2 | 34 | 15 | 1.7 | 4,12,6,12 | 1,5,3,6 |
| 81 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.3 | 8 | 0 | 1.8 | | |
| 82 | 10.6 | 33 | 15 | 11.9 | 27 | 12 | 7.4 | 31 | 13 | 2.1 | 6,8,6,11 | 2,3,3,5 |
| 83 | 3.2 | 15 | 6 | 3.7 | 14 | 6 | 3.7 | 17 | 6 | 1.4 | 6,4,0,7 | 2,1,0,3 |
| 84 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 3.5 | 8 | 0 | 2.2 | | |
| 85 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 3.0 | 8 | 0 | 2.3 | | |
| 86 | 3.5 | 19 | 8 | 4.3 | 19 | 8 | 5.6 | 21 | 8 | 1.3 | 4,8,0,9 | 1,3,0,4 |
| 87 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.2 | 4 | 0 | 1.7 | | |
| 88 | 8.4 | 23 | 10 | 10.4 | 24 | 10 | 9.4 | 26 | 10 | 2.2 | 6,8,0,12 | 2,3,0,5 |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 89 | 10.2 | 62 | 6 | 4.2 | 16 | 7 | 4.1 | 18 | 7 | 1.3 | 4,6,2,6 | 1,2,1,3 |
| 90 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 3.6 | 4 | 0 | 2.9 | | |
| 91 | 19.1 | 37 | 16 | 26.8 | 37 | 17 | 12.6 | 41 | 17 | 2.4 | 8,10,6,17 | 3,4,3,7 |
| 92 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 4.7 | 8 | 0 | 3.8 | | |
| 93 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 4.8 | 21 | 0 | 3.0 | | |
| 94 | 2.4 | 19 | 8 | 3.0 | 15 | 6 | 1.5 | 6 | 2 | 0.7 | 4,0,2 | 1,0,1 |
| 95 | 600.0 | 540 | — | 37.1 | 43 | 20 | 21.5 | 39 | 17 | 2.8 | 6,14,2,17 | 2,6,1,8 |
| 96 | 24.8 | 43 | 18 | 57.9 | 70 | 18 | 6.9 | 38 | 16 | 1.4 | 4,6,12,16 | 1,2,6,7 |
| 97 | 42.9 | 51 | 24 | 58.1 | 47 | 22 | 41.3 | 52 | 23 | 3.6 | 6,18,6,22 | 2,8,3,10 |
| 98 | 16.9 | 34 | 16 | 25.3 | 34 | 16 | 19.3 | 36 | 16 | 2.6 | 4,14,4,14 | 1,6,2,7 |
| 99 | 12.8 | 33 | 15 | 17.4 | 31 | 13 | 9.0 | 35 | 15 | 2.1 | 4,12,6,13 | 1,5,3,6 |
| 100 | 3.5 | 14 | 6 | 5.3 | 16 | 7 | 5.6 | 18 | 7 | 1.5 | 4,6,2,6 | 1,2,1,3 |
| 101 | 3.5 | 19 | 8 | 4.1 | 19 | 8 | 3.4 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 102 | 0.0 | 1 | 0 | 0.1 | 1 | 0 | 0.7 | 1 | 0 | 0.5 | | |
| 103 | 0.6 | 8 | 3 | 0.8 | 8 | 3 | 1.5 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 104 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 105 | 0.5 | 8 | 3 | 0.6 | 8 | 3 | 1.4 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 106 | 2.2 | 15 | 6 | 3.2 | 15 | 6 | 2.8 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 107 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 108 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 109 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 110 | 3.3 | 19 | 8 | 2.6 | 15 | 6 | 2.8 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 111 | 0.3 | 6 | 2 | 0.5 | 6 | 2 | 1.1 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 112 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 113 | 3.1 | 19 | 8 | 4.5 | 19 | 8 | 2.8 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 114 | 0.4 | 6 | 2 | 0.5 | 6 | 2 | 1.2 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 115 | 0.3 | 6 | 2 | 0.5 | 6 | 2 | 1.8 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 116 | 0.3 | 6 | 2 | 0.5 | 6 | 2 | 1.1 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 117 | 2.5 | 19 | 8 | 4.7 | 19 | 8 | 1.6 | 6 | 2 | 0.6 | 4,0,2 | 1,0,1 |
| 118 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 119 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.6 | 1 | 0 | 0.5 | | |
| 120 | 0.5 | 8 | 3 | 0.7 | 8 | 3 | 1.4 | 8 | 3 | 0.5 | 4,2,2 | 1,1,1 |
| 121 | 3.3 | 19 | 8 | 5.1 | 19 | 8 | 2.5 | 15 | 6 | 0.6 | 4,4,7 | 1,2,3 |
| 122 | 3.3 | 19 | 8 | 5.2 | 19 | 8 | 2.8 | 19 | 8 | 0.6 | 4,8,7 | 1,4,3 |
| 123 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.3 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 124 | 0.4 | 6 | 2 | 0.5 | 6 | 2 | 1.2 | 6 | 2 | 0.4 | 4,0,2 | 1,0,1 |
| 125 | 0.7 | 6 | 2 | 0.8 | 6 | 2 | 1.2 | 6 | 2 | 0.5 | 4,0,2 | 1,0,1 |
| 126 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 1.4 | 8 | 0 | 0.8 | | |
| 127 | 1.5 | 10 | 4 | 1.8 | 10 | 4 | 2.7 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 128 | 1.2 | 10 | 4 | 1.7 | 10 | 4 | 2.4 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 129 | 1.1 | 12 | 5 | 1.6 | 12 | 5 | 2.2 | 12 | 5 | 0.7 | 6,2,4 | 2,1,2 |
| 130 | 4.1 | 23 | 10 | 6.8 | 23 | 10 | 4.9 | 23 | 10 | 0.9 | 10,4,9 | 4,2,4 |
| 131 | 1.8 | 15 | 6 | 3.1 | 15 | 6 | 2.6 | 15 | 6 | 0.7 | 6,4,5 | 2,2,2 |
| 132 | 600.0 | 2635 | — | 5.7 | 20 | 9 | 3.1 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 133 | 2.4 | 10 | 4 | 2.2 | 12 | 5 | 2.7 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 134 | 1.0 | 10 | 4 | 1.3 | 10 | 4 | 2.2 | 10 | 4 | 0.7 | 6,0,4 | 2,0,2 |
| 135 | 1.4 | 10 | 4 | 1.7 | 10 | 4 | 2.2 | 12 | 4 | 0.7 | 4,4,0,4 | 1,1,0,2 |
| 136 | 1.3 | 10 | 4 | 1.7 | 10 | 4 | 2.3 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 137 | 1.8 | 12 | 5 | 2.4 | 12 | 5 | 2.4 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 138 | 1.4 | 10 | 4 | 1.6 | 10 | 4 | 2.2 | 12 | 4 | 0.7 | 4,4,0,4 | 1,1,0,2 |
| 139 | 1.6 | 14 | 6 | 2.5 | 14 | 6 | 2.5 | 14 | 6 | 0.8 | 6,4,4 | 2,2,2 |
| 140 | 3.0 | 15 | 6 | 7.0 | 22 | 10 | 3.6 | 15 | 6 | 1.0 | 8,0,7 | 3,0,3 |
| 141 | 1.6 | 14 | 6 | 2.5 | 14 | 6 | 2.5 | 14 | 6 | 0.8 | 6,4,4 | 2,2,2 |
| 142 | 2.0 | 12 | 5 | 2.6 | 12 | 5 | 2.5 | 14 | 5 | 0.8 | 4,4,2,4 | 1,1,1,2 |
| 143 | 2.0 | 11 | 4 | 2.4 | 11 | 4 | 2.0 | 11 | 4 | 0.8 | 6,0,5 | 2,0,2 |
| 144 | 1.6 | 10 | 4 | 2.0 | 10 | 4 | 2.2 | 12 | 4 | 0.7 | 4,4,0,4 | 1,1,0,2 |
| 145 | 600.0 | 2504 | — | 600.0 | 1658 | — | 4.4 | 21 | 0 | 0.8 | | |
| 146 | 1.0 | 10 | 4 | 1.4 | 10 | 4 | 2.3 | 10 | 4 | 0.8 | 6,0,4 | 2,0,2 |
| 147 | 1.6 | 10 | 4 | 2.0 | 10 | 4 | 2.3 | 12 | 4 | 0.8 | 4,4,0,4 | 1,1,0,2 |
| 148 | 1.1 | 10 | 4 | 1.4 | 10 | 4 | 2.1 | 10 | 4 | 0.7 | 6,0,4 | 2,0,2 |
| 149 | 1.4 | 10 | 4 | 1.7 | 10 | 4 | 2.2 | 12 | 4 | 0.7 | 4,4,0,4 | 1,1,0,2 |
| 150 | 1.4 | 12 | 5 | 2.0 | 12 | 5 | 2.2 | 12 | 5 | 0.6 | 6,2,4 | 2,1,2 |
| 151 | 2.7 | 17 | 7 | 6.0 | 21 | 9 | 4.6 | 17 | 7 | 1.1 | 8,2,7 | 3,1,3 |
| 152 | 12.9 | 74 | 6 | 4.2 | 14 | 6 | 3.5 | 16 | 6 | 1.3 | 4,6,0,6 | 1,2,0,3 |
| 153 | 4.0 | 17 | 7 | 6.1 | 17 | 7 | 3.5 | 19 | 7 | 1.1 | 6,4,2,7 | 2,1,1,3 |
| 154 | 12.6 | 75 | 6 | 4.6 | 15 | 6 | 3.9 | 17 | 6 | 1.2 | 6,4,0,7 | 2,1,0,3 |
| 155 | 1.9 | 14 | 6 | 2.8 | 14 | 6 | 3.0 | 14 | 6 | 1.0 | 8,0,6 | 3,0,3 |
| 156 | 2.6 | 14 | 6 | 3.5 | 14 | 6 | 3.9 | 16 | 6 | 1.1 | 4,6,0,6 | 1,2,0,3 |
| 157 | 3.1 | 18 | 8 | 4.8 | 18 | 8 | 3.9 | 18 | 8 | 1.0 | 8,4,6 | 3,2,3 |
| 158 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.7 | 8 | 0 | 1.1 | | |
| 159 | 2.4 | 16 | 7 | 3.6 | 16 | 7 | 3.9 | 16 | 7 | 1.0 | 8,2,6 | 3,1,3 |
| 160 | 2.2 | 14 | 6 | 3.1 | 14 | 6 | 4.2 | 14 | 6 | 1.0 | 8,0,6 | 3,0,3 |
| 161 | 3.9 | 15 | 6 | 3.6 | 14 | 6 | 3.2 | 17 | 6 | 1.1 | 6,4,0,7 | 2,1,0,3 |
| 162 | 4.6 | 16 | 7 | 4.6 | 16 | 7 | 4.3 | 16 | 7 | 1.2 | 8,2,6 | 3,1,3 |
| 163 | 2.5 | 14 | 6 | 3.3 | 14 | 6 | 3.2 | 16 | 6 | 1.0 | 4,6,0,6 | 1,2,0,3 |
| 164 | 3.4 | 14 | 6 | 3.8 | 16 | 7 | 3.9 | 18 | 7 | 1.0 | 4,6,2,6 | 1,2,1,3 |
| 165 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.4 | 1 | 0 | 1.3 | | |
| 166 | 7.0 | 29 | 13 | 12.7 | 29 | 13 | 4.1 | 16 | 7 | 1.3 | 8,2,6 | 3,1,3 |
| 167 | 2.1 | 14 | 6 | 2.9 | 14 | 6 | 3.4 | 14 | 6 | 1.0 | 8,0,6 | 3,0,3 |
| 168 | 600.0 | 1291 | — | 8.9 | 24 | 11 | 10.5 | 34 | 11 | 1.1 | 22,2,10 | 5,1,5 |
| 169 | 3.0 | 14 | 6 | 3.8 | 16 | 7 | 3.8 | 18 | 7 | 1.0 | 4,6,2,6 | 1,2,1,3 |
| 170 | 3.0 | 16 | 7 | 4.0 | 16 | 7 | 3.5 | 18 | 7 | 1.1 | 4,6,2,6 | 1,2,1,3 |
| 171 | 7.7 | 26 | 12 | 10.0 | 26 | 11 | 6.7 | 26 | 12 | 1.2 | 12,4,10 | 5,2,5 |
| 172 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.3 | 1 | 0 | 1.1 | | |
| 173 | 2.6 | 16 | 7 | 4.6 | 18 | 8 | 3.9 | 16 | 7 | 1.0 | 8,2,6 | 3,1,3 |
| 174 | 6.0 | 22 | 10 | 11.4 | 24 | 11 | 4.8 | 24 | 10 | 1.4 | 4,6,4,10 | 1,2,2,5 |
| 175 | 3.4 | 15 | 6 | 4.2 | 15 | 6 | 3.3 | 17 | 6 | 1.1 | 6,4,0,7 | 2,1,0,3 |
| 176 | 5.9 | 25 | 11 | 5.7 | 19 | 8 | 6.4 | 20 | 9 | 1.3 | 10,2,8 | 4,1,4 |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 177 | 5.2 | 18 | 8 | 7.0 | 18 | 8 | 5.7 | 20 | 8 | 1.4 | 4,8,0,8 | 1,3,0,4 |
| 178 | 5.0 | 20 | 9 | 7.2 | 20 | 9 | 6.4 | 22 | 9 | 1.5 | 4,8,2,8 | 1,3,1,4 |
| 179 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.2 | 8 | 0 | 1.5 | | |
| 180 | 5.8 | 19 | 8 | 6.9 | 20 | 9 | 5.1 | 23 | 9 | 1.3 | 6,6,2,9 | 2,2,1,4 |
| 181 | 5.6 | 20 | 9 | 7.1 | 20 | 9 | 6.2 | 22 | 9 | 1.4 | 4,8,2,8 | 1,3,1,4 |
| 182 | 4.0 | 19 | 8 | 5.5 | 19 | 8 | 5.7 | 19 | 8 | 1.4 | 10,0,9 | 4,0,4 |
| 183 | 122.4 | 228 | 15 | 19.1 | 33 | 15 | 7.3 | 23 | 10 | 1.6 | 10,4,9 | 4,2,4 |
| 184 | 16.4 | 37 | 17 | 45.6 | 48 | 21 | 8.6 | 24 | 11 | 1.7 | 12,2,10 | 5,1,5 |
| 185 | 5.2 | 21 | 9 | 8.9 | 22 | 10 | 8.0 | 24 | 10 | 1.6 | 4,8,4,8 | 1,3,2,4 |
| 186 | 6.0 | 25 | 11 | 9.9 | 25 | 11 | 5.7 | 25 | 11 | 1.4 | 10,6,9 | 4,3,4 |
| 187 | 11.0 | 33 | 15 | 32.1 | 63 | 15 | 8.9 | 35 | 15 | 1.7 | 4,13,6,12 | 1,5,3,6 |
| 188 | 3.7 | 20 | 9 | 5.6 | 20 | 9 | 6.2 | 20 | 9 | 1.5 | 10,2,8 | 4,1,4 |
| 189 | 6.3 | 21 | 9 | 8.7 | 21 | 9 | 4.6 | 23 | 9 | 1.3 | 6,6,2,9 | 2,2,1,4 |
| 190 | 3.8 | 18 | 8 | 5.1 | 18 | 8 | 5.8 | 20 | 8 | 1.4 | 4,8,0,8 | 1,3,0,4 |
| 191 | 12.0 | 33 | 14 | 19.5 | 33 | 14 | 9.2 | 33 | 14 | 1.6 | 15,4,14 | 6,2,6 |
| 192 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 1.8 | 8 | 0 | 1.4 | | |
| 193 | 5.5 | 21 | 9 | 8.9 | 23 | 10 | 5.3 | 25 | 10 | 1.5 | 6,6,4,9 | 2,2,2,4 |
| 194 | 600.0 | 1174 | — | 15.3 | 30 | 14 | 9.1 | 27 | 12 | 1.5 | 14,0,13 | 6,0,6 |
| 195 | 4.0 | 18 | 8 | 6.5 | 20 | 9 | 6.0 | 18 | 8 | 1.6 | 10,0,8 | 4,0,4 |
| 196 | 7.9 | 26 | 12 | 13.0 | 28 | 13 | 8.7 | 26 | 12 | 1.5 | 14,0,12 | 6,0,6 |
| 197 | 11.1 | 30 | 13 | 19.1 | 32 | 14 | 5.3 | 22 | 9 | 1.8 | 4,8,2,8 | 1,3,1,4 |
| 198 | 4.2 | 18 | 8 | 5.7 | 18 | 8 | 5.4 | 20 | 8 | 1.4 | 4,8,0,8 | 1,3,0,4 |
| 199 | 5.9 | 21 | 9 | 7.0 | 21 | 9 | 4.6 | 23 | 9 | 1.3 | 6,6,2,9 | 2,2,1,4 |
| 200 | 12.2 | 36 | 15 | 19.7 | 36 | 15 | 12.9 | 36 | 15 | 1.6 | 17,6,13 | 6,3,6 |
| 201 | 9.8 | 25 | 11 | 14.5 | 26 | 12 | 11.6 | 35 | 15 | 1.8 | 6,10,6,13 | 2,4,3,6 |
| 202 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 3.3 | 21 | 0 | 1.8 | | |
| 203 | 6.7 | 23 | 10 | 9.8 | 23 | 10 | 9.2 | 23 | 10 | 1.7 | 12,0,11 | 5,0,5 |
| 204 | 600.0 | 1903 | — | 11.9 | 25 | 11 | 6.8 | 27 | 11 | 1.8 | 6,8,2,11 | 2,3,1,5 |
| 205 | 10.2 | 28 | 12 | 15.5 | 28 | 12 | 6.4 | 30 | 12 | 1.9 | 6,8,4,12 | 2,3,2,5 |
| 206 | 600.0 | 592 | — | 600.0 | 659 | — | 600.0 | 607 | — | 2.1 | | |
| 207 | 8.1 | 23 | 10 | 10.5 | 23 | 10 | 8.7 | 25 | 10 | 1.9 | 6,8,0,11 | 2,3,0,5 |
| 208 | 17.3 | 37 | 17 | 27.7 | 37 | 17 | 18.5 | 37 | 17 | 2.1 | 17,6,14 | 7,3,7 |
| 209 | 9.4 | 27 | 12 | 13.0 | 26 | 12 | 8.1 | 29 | 12 | 1.8 | 6,8,4,11 | 2,3,2,5 |
| 210 | 600.0 | 743 | — | 600.0 | 595 | — | 46.5 | 68 | 0 | 2.2 | | |
| 211 | 8.3 | 27 | 12 | 12.5 | 27 | 12 | 8.5 | 29 | 12 | 1.8 | 4,10,4,11 | 1,4,2,5 |
| 212 | 8.4 | 25 | 10 | 14.2 | 27 | 12 | 12.8 | 29 | 12 | 1.7 | 4,10,4,11 | 1,4,2,5 |
| 213 | 6.7 | 24 | 11 | 10.2 | 24 | 11 | 8.4 | 24 | 11 | 1.8 | 12,2,10 | 5,1,5 |
| 214 | 8.1 | 24 | 11 | 13.2 | 26 | 12 | 14.7 | 34 | 15 | 1.9 | 4,12,6,12 | 1,5,3,6 |
| 215 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 2.0 | 1 | 0 | 1.7 | | |
| 216 | 41.8 | 81 | 17 | 32.1 | 41 | 19 | 18.3 | 41 | 17 | 2.0 | 18,6,17 | 7,3,7 |
| 217 | 8.0 | 26 | 11 | 11.7 | 26 | 11 | 8.1 | 26 | 11 | 1.4 | 12,2,12 | 5,1,5 |
| 218 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 2.6 | 8 | 0 | 2.0 | | |
| 219 | 600.0 | 1758 | — | 11.9 | 24 | 11 | 7.2 | 26 | 11 | 1.9 | 4,10,2,10 | 1,4,1,5 |
| 220 | 16.4 | 36 | 16 | 24.9 | 36 | 16 | 19.0 | 38 | 16 | 2.0 | 4,14,4,16 | 1,6,2,7 |

| Prob | Prodigy + EBL | | | Prodigy + Static | | | Prodigy + Alpine + HCR | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 221 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 3.6 | 21 | 0 | 1.9 | | |
| 222 | 600.0 | 436 | — | 600.0 | 516 | — | 39.4 | 74 | 0 | 2.0 | | |
| 223 | 112.8 | 168 | 16 | 35.2 | 66 | 17 | 15.0 | 38 | 17 | 2.1 | 4,14,6,14 | 1,6,3,7 |
| 224 | 10.5 | 28 | 13 | 12.5 | 25 | 11 | 14.6 | 32 | 14 | 1.6 | 4,12,4,12 | 1,5,2,6 |
| 225 | 600.0 | 651 | — | 25.1 | 35 | 16 | 18.8 | 42 | 15 | 1.9 | 4,22,2,14 | 1,6,1,7 |
| 226 | 600.0 | 709 | — | 39.0 | 37 | 17 | 15.0 | 33 | 15 | 2.4 | 14,6,13 | 6,3,6 |
| 227 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 4.1 | 21 | 0 | 2.3 | | |
| 228 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 2.7 | 8 | 0 | 2.1 | | |
| 229 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 3.5 | 8 | 0 | 2.3 | | |
| 230 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 3.1 | 8 | 0 | 2.4 | | |
| 231 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 3.7 | 21 | 0 | 2.1 | | |
| 232 | 13.2 | 29 | 13 | 18.3 | 29 | 13 | 14.4 | 31 | 13 | 2.2 | 4,12,2,13 | 1,5,1,6 |
| 233 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 4.8 | 21 | 0 | 2.3 | | |
| 234 | 600.0 | 698 | — | 34.9 | 40 | 19 | 10.1 | 31 | 13 | 2.3 | 4,12,2,13 | 1,5,1,6 |
| 235 | 13.7 | 29 | 13 | 17.7 | 28 | 13 | 12.5 | 31 | 13 | 2.2 | 6,10,2,13 | 2,4,1,6 |
| 236 | 118.5 | 208 | 16 | 21.4 | 30 | 14 | 14.0 | 32 | 14 | 2.2 | 4,12,4,12 | 1,5,2,6 |
| 237 | 50.6 | 89 | 15 | 29.1 | 35 | 15 | 11.0 | 37 | 15 | 2.4 | 4,12,6,15 | 1,5,3,6 |
| 238 | 600.0 | 517 | — | 46.8 | 45 | 20 | 17.0 | 47 | 20 | 2.4 | 4,16,8,19 | 1,7,4,8 |
| 239 | 600.0 | 877 | — | 600.0 | 712 | — | 69.6 | 115 | 0 | 2.3 | | |
| 240 | 600.0 | 713 | — | 37.9 | 40 | 19 | 17.3 | 43 | 19 | 2.3 | 6,14,6,17 | 2,6,3,8 |
| 241 | 12.1 | 32 | 15 | 20.3 | 32 | 15 | 13.0 | 32 | 15 | 2.1 | 14,6,12 | 6,3,6 |
| 242 | 600.0 | 307 | — | 600.0 | 439 | — | 600.0 | 374 | — | 2.4 | | |
| 243 | 600.0 | 751 | — | 600.0 | 561 | — | 526.1 | 415 | 0 | 2.3 | | |
| 244 | 11.6 | 28 | 12 | 15.6 | 28 | 12 | 19.2 | 43 | 17 | 1.7 | 4,18,2,19 | 1,7,1,8 |
| 245 | 0.2 | 1 | 0 | 0.2 | 1 | 0 | 2.3 | 1 | 0 | 2.1 | | |
| 246 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 4.2 | 21 | 0 | 2.2 | | |
| 247 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 2.6 | 1 | 0 | 2.9 | | |
| 248 | 0.1 | 1 | 0 | 0.2 | 1 | 0 | 2.9 | 8 | 0 | 2.5 | | |
| 249 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 7.5 | 69 | 0 | 2.1 | | |
| 250 | 22.9 | 40 | 19 | 70.5 | 72 | 20 | 20.5 | 44 | 20 | 2.5 | 4,16,8,16 | 1,7,4,8 |

# Appendix D

# STRIPS Robot Planning Domain

This is the original STRIPS robot planning domain [Fikes and Nilsson, 1971] as it is encoded in PRODIGY. The only differences are that the variable arguments are typed and the deletes are changed into conditional deletes. The numbers after the preconditions are the criticalities that ABSTRIPS assigned and are used in the comparison with ALPINE.

## D.1 Problem Space Definition

```
(GOTO-BOX
 (params (<box> <room>))
 (preconds (and
            (is-type <box> box) ;(6)
            (in-room <box> <room>) ;(5)
            (in-room robot <room>) ;(3)
            ))
 (effects ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
           (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
           (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
           (add (next-to robot <box>)))))

(GOTO-DOOR
 (params (<door> <room.x>))
 (preconds (and
            (is-type <door> door) ;(6)
            (connects <door> <room.x> <room.y>) ;(6)
            (in-room robot <room.x>))) ;(3)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
```

```
     (add (next-to robot <door>)))))

(GOTO-LOC
 (params (<loc.x> <loc.y> <room.x>))
 (preconds (and
            (loc-in-room <loc.x> <loc.y> <room.x>) ;(6)
            (in-room robot <room.x>))) ;(3)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (add (at robot <loc.x> <loc.y>)))))

(PUSH-BOX
 (params (<box.x> <box.y>))
 (preconds (and
            (is-type <box.y> box) ;(6)
            (pushable <box.x>) ;(6)
            (in-room <box.y> <room.x>) ;(5)
            (in-room <box.x> <room.x>) ;(5)
            (in-room robot <room.x>) ;(3)
            (next-to robot <box.x>))) ;(2)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (if (at <box.x> <loc.3> <loc.4>)(del (at <box.x> <loc.3> <loc.4>)))
     (if (next-to <box.x> <box.2>)(del (next-to <box.x> <box.2>)))
     (if (next-to <box.x> <door.2>)(del (next-to <box.x> <door.2>)))
     (if (next-to <box.3> <box.x>)(del (next-to <box.3> <box.x>)))
     (add (next-to <box.y> <box.x>))
     (add (next-to <box.x> <box.y>))
     (add (next-to robot <box.x>)))))

(PUSH-TO-DOOR
 (params (<box> <door> <room.x>))
 (preconds (and
            (connects <door> <room.x> <room.y>) ;(6)
            (pushable <box>) ;(6)
            (is-type <door> door) ;(6)
            (in-room <box> <room.x>) ;(5)
            (in-room robot <room.x>) ;(3)
            (next-to robot <box>))) ;(2)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (if (at <box> <loc.3> <loc.4>)(del (at <box> <loc.3> <loc.4>)))
```

```
     (if (next-to <box> <box.2>)(del (next-to <box> <box.2>)))
     (if (next-to <box> <door.2>)(del (next-to <box> <door.2>)))
     (if (next-to <box.3> <box>)(del (next-to <box.3> <box>)))
     (add (next-to <box> <door>))
     (add (next-to robot <box>)))))

(PUSH-TO-LOC
 (params (<box> <loc.x> <loc.y>))
 (preconds (and
            (pushable <box>) ;(6)
            (loc-in-room <loc.x> <loc.y> <room.x>) ;(6)
            (in-room <box> <room.x>) ;(5)
            (in-room robot <room.x>) ;(3)
            (next-to robot <box>))) ;(2)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (if (at <box> <loc.3> <loc.4>)(del (at <box> <loc.3> <loc.4>)))
     (if (next-to <box> <box.2>)(del (next-to <box> <box.2>)))
     (if (next-to <box> <door.2>)(del (next-to <box> <door.2>)))
     (if (next-to <box.3> <box>)(del (next-to <box.3> <box>)))
     (add (at <box> <loc.x> <loc.y>))
     (add (next-to robot <box>)))))

(GO-THRU-DOOR
 (params (<door> <room.y> <room.x>))
 (preconds (and
            (connects <door> <room.y> <room.x>) ;(6)
            (is-type <door> door) ;(6)
            (is-type <room.x> room) ;(6)
            (in-room robot <room.y>) ;(3)
            (status <door> open))) ;(1)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (if (in-room robot <room.y>)(del (in-room robot <room.y>)))
     (add (in-room robot <room.x>)))))

(PUSH-THRU-DOOR
 (params (<box> <door> <room.y> <room.x>))
 (preconds (and
            (connects <door> <room.y> <room.x>) ;(6)
            (pushable <box>) ;(6)
            (is-type <door> door) ;(6)
            (is-type <room.x> room) ;(6)
            (in-room <box> <room.y>) ;(5)
```

```
            (next-to <box> <door>) ;(4)
            (in-room robot <room.y>) ;(3)
            (next-to robot <box>) ;(2)
            (status <door> open))) ;(1)
 (effects
    ((if (at robot <loc.1> <loc.2>)(del (at robot <loc.1> <loc.2>)))
     (if (next-to robot <box.1>)(del (next-to robot <box.1>)))
     (if (next-to robot <door.1>)(del (next-to robot <door.1>)))
     (if (at <box> <loc.3> <loc.4>)(del (at <box> <loc.3> <loc.4>)))
     (if (next-to <box> <box.2>)(del (next-to <box> <box.2>)))
     (if (next-to <box> <door.2>)(del (next-to <box> <door.2>)))
     (if (next-to <box.3> <box>)(del (next-to <box.3> <box>)))
     (if (in-room robot <room.y>)(del (in-room robot <room.y>)))
     (if (in-room <box> <room.y>)(del (in-room <box> <room.y>)))
     (add (in-room robot <room.x>))
     (add (in-room <box> <room.x>))
     (add (next-to robot <box>)))))

(OPEN-DOOR
 (params (<door>))
 (preconds (and
            (is-type <door> door) ;(6)
            (next-to robot <door>) ;(2)
            (status <door> closed))) ;(1)
 (effects
    ((if (status <door> closed)(del (status <door> closed)))
     (add (status <door> open)))))

(CLOSE-DOOR
 (params (<door>))
 (preconds (and
            (is-type <door> door) ;(6)
            (next-to robot <door>) ;(2)
            (status <door> open))) ;(1)
 (effects
    ((if (status <door> open)(del (status <door> open)))
     (add (status <door> closed)))))

(setq *AXIOMS*
      '(((next-to <box.1-1> <box.2-1>) . ((inroom <box.1-1> <room.1-1>)
                                          (inroom <box.2-1> <room.1-1>)))
        ((next-to robot <box.1-2>) . ((in-room <box.1-2> <room.1-2>)
                                      (in-room robot <room.1-2>)))
        ((next-to robot <door.1-3>) .
                     ((connects <door.1-3> <room.x-3> <room.y-3>)
                              (in-room robot <room.x-3>)))
        ((~ (status <door.1-4> closed)) . ((status <door.1-4> open)))
        ((~ (status <door.1-5> open)) . ((status <door.1-5> closed)))
```

```
       ))

(setq *VARIABLE-TYPING* '(
   (isa 'robot 'type)(isa 'box 'object)(isa 'door 'object)
   (isa 'object 'type)(isa 'room 'type)(isa 'loc 'type)
   (isa 'status 'type)
   (isa-instance 'open 'status)(isa-instance 'closed 'status)
   (isa-instance 'robot 'robot)(isa-instance 'a 'box)
   (isa-instance 'b 'box)(isa-instance 'c 'box)
   (isa-instance 'd 'box)(isa-instance 'e 'box)
   (isa-instance 'f 'box)(isa-instance 'room1 'room)
   (isa-instance 'room2 'room)(isa-instance 'room3 'room)
   (isa-instance 'room4 'room)(isa-instance 'room5 'room)
   (isa-instance 'room6 'room)(isa-instance 'room7 'room)
   (isa-instance 'door12 'door)(isa-instance 'door23 'door)
   (isa-instance 'door34 'door)(isa-instance 'door25 'door)
   (isa-instance 'door56 'door)(isa-instance 'door26 'door)
   (isa-instance 'door36 'door)(isa-instance 'door67 'door)))

(setq *PRIMARY* '(
      ((next-to robot <box>) . (GOTO-BOX))
      ((next-to robot <door>) . (GOTO-DOOR))
      ((at robot <loc.x> <loc.y>) . (GOTO-LOC))
      ((next-to <box.1> <box.2>) . (PUSH-BOX))
      ((next-to <box> <door>) . (PUSH-TO-DOOR))
      ((at box <loc.x> <loc.y>) . (PUSH-TO-LOC))
      ((in-room robot <room>) . (GO-THRU-DOOR))
      ((in-room <box> <room>) . (PUSH-THRU-DOOR))
      ((status <door> open) . (OPEN-DOOR))
      ((status <door> closed) . (CLOSE-DOOR))))

Example Problem:

Goal: '(and (in-room a room1) (status door56 closed)
            (status door12 closed) (in-room robot room3)
            (in-room b room6))

Initial State:
'((connects door67 room7 room6)(connects door67 room6 room7)
  (connects door56 room6 room5)(connects door56 room5 room6)
  (connects door36 room6 room3)(connects door36 room3 room6)
  (connects door26 room6 room2)(connects door26 room2 room6)
  (connects door25 room5 room2)(connects door25 room2 room5)
  (connects door34 room4 room3)(connects door34 room3 room4)
  (connects door23 room3 room2)(connects door23 room2 room3)
  (connects door12 room2 room1)(connects door12 room1 room2)
  (next-to d door36) (status door67 closed)
  (status door56 open) (status door36 open) (status door26 open)
```

```
(status door25 closed) (status door34 open) (status door23 open)
(status door12 open) (is-type room7 room) (is-type room6 room)
(is-type room5 room) (is-type room4 room) (is-type room3 room)
(is-type room2 room) (is-type room1 room) (is-type door67 door)
(is-type door56 door) (is-type door36 door)
(is-type door26 door) (is-type door25 door)
(is-type door34 door) (is-type door23 door)
(is-type door12 door) (pushable e) (pushable d) (pushable c)
(pushable b) (pushable a) (is-type e box) (is-type d box)
(is-type c box) (is-type b box) (is-type a box)
(in-room e room5) (in-room d room6) (in-room c room4)
(in-room b room7) (in-room a room2) (in-room robot room2))
```

## D.2 Experimental Results

The experiments in this domain were run in CMU Common Lisp on a IBM RT Model 130 with 16 megabytes of memory. The tables below compares PRODIGY without any control knowledge, PRODIGY with the abstractions generated by ABSTRIPS, and PRODIGY with the abstractions generated by ALPINE.

The entries in the table are defined as follows:

**Prob Num** The problem number.

**Time** Total CPU time used in solving the problem. A 600 CPU second time bound was imposed on all problems.

**Nodes** Total number of nodes searched in solving the problem.

**Len** Length of the solution found. Zero if no solution exists.

**ACT** Time required to create the abstraction hierarchy. This time is also included in the total CPU time for ALPINE.

**AbNodes** Nodes searched at each level in the hierarchy. Ordered from more abstract to less abstract levels.

**AbLen** Solution length found at each level in the hierarchy. Ordered from more abstract to less abstract levels.

| Prob | Prodigy | | | Prodigy + Abstrips | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 1 | 3.7 | 37 | 16 | 4.4 | 33 | 13 | 8.3 | 37 | 16 | 0.6 | 19,4,14 | 7,2,7 |
| 2 | 6.7 | 69 | 28 | 5.8 | 36 | 16 | 12.3 | 59 | 25 | 0.4 | 31,8,20 | 12,3,10 |
| 3 | 1.3 | 21 | 8 | 3.2 | 30 | 12 | 3.0 | 23 | 8 | 0.4 | 4,12,7 | 1,4,3 |
| 4 | 2.6 | 30 | 12 | 1.9 | 16 | 6 | 2.3 | 15 | 5 | 0.4 | 4,4,7 | 1,1,3 |
| 5 | 4.1 | 46 | 20 | 4.3 | 35 | 15 | 9.9 | 48 | 20 | 0.7 | 22,16,10 | 8,7,5 |
| 6 | 1.6 | 28 | 12 | 2.8 | 23 | 9 | 3.5 | 30 | 12 | 0.4 | 4,18,8 | 1,7,4 |
| 7 | 3.8 | 51 | 22 | 1.7 | 16 | 7 | 2.3 | 18 | 7 | 0.4 | 4,8,6 | 1,3,3 |
| 8 | 0.7 | 15 | 6 | 1.1 | 12 | 4 | 1.6 | 15 | 6 | 0.3 | 11,4 | 4,2 |
| 9 | 6.5 | 72 | 24 | 8.1 | 56 | 23 | 13.1 | 72 | 24 | 0.4 | 50,0,22 | 13,0,11 |
| 10 | 4.7 | 50 | 21 | 3.7 | 26 | 11 | 5.6 | 29 | 13 | 0.4 | 13,4,12 | 5,2,6 |
| 11 | 4.7 | 52 | 23 | 9.9 | 75 | 30 | 10.3 | 51 | 23 | 0.6 | 20,13,18 | 9,5,9 |
| 12 | 4.7 | 52 | 23 | 6.8 | 46 | 21 | 8.8 | 53 | 24 | 0.4 | 15,6,32 | 5,3,16 |
| 13 | 1.4 | 32 | 10 | 2.9 | 34 | 13 | 2.5 | 32 | 10 | 0.3 | 26,6 | 7,3 |
| 14 | 4.3 | 56 | 24 | 5.7 | 48 | 22 | 9.0 | 66 | 26 | 0.4 | 13,32,21 | 4,12,10 |
| 15 | 2.9 | 39 | 8 | 4.9 | 34 | 12 | 4.3 | 24 | 10 | 0.4 | 6,12,6 | 2,5,3 |
| 16 | 4.1 | 55 | 21 | 3.9 | 37 | 15 | 7.5 | 42 | 18 | 0.4 | 14,10,18 | 6,3,9 |
| 17 | 2.9 | 37 | 14 | 2.1 | 16 | 7 | 2.9 | 21 | 7 | 0.5 | 4,10,7 | 1,3,3 |
| 18 | 2.0 | 39 | 16 | 1.8 | 24 | 10 | 3.5 | 39 | 16 | 0.3 | 29,10 | 11,5 |
| 19 | 2.9 | 43 | 11 | 5.4 | 47 | 18 | 4.2 | 29 | 11 | 0.4 | 8,10,11 | 2,4,5 |
| 20 | 4.0 | 52 | 21 | 4.8 | 40 | 17 | 8.9 | 54 | 21 | 0.4 | 21,23,10 | 7,9,5 |
| 21 | 1.4 | 20 | 8 | 2.1 | 18 | 8 | 2.9 | 21 | 8 | 0.4 | 4,11,6 | 1,4,3 |
| 22 | 4.4 | 62 | 19 | 6.8 | 49 | 20 | 8.2 | 44 | 19 | 0.4 | 19,9,16 | 7,4,8 |
| 23 | 2.7 | 29 | 11 | 3.2 | 29 | 11 | 6.8 | 30 | 11 | 0.5 | 20,6,4 | 7,2,2 |
| 24 | 2.7 | 34 | 14 | 4.3 | 31 | 14 | 6.2 | 36 | 14 | 0.4 | 15,15,6 | 5,6,3 |
| 25 | 3.6 | 42 | 16 | 5.7 | 46 | 16 | 8.1 | 41 | 16 | 0.6 | 19,10,12 | 7,3,6 |
| 26 | 23.7 | 494 | 16 | 41.3 | 502 | 12 | 8.0 | 46 | 18 | 0.6 | 15,23,8 | 5,9,4 |
| 27 | 4.6 | 63 | 25 | 6.6 | 69 | 24 | 5.2 | 41 | 16 | 0.6 | 4,27,10 | 1,10,5 |
| 28 | 12.4 | 287 | 12 | 6.6 | 75 | 22 | 3.4 | 22 | 9 | 0.6 | 4,10,8 | 1,4,4 |
| 29 | 5.9 | 65 | 27 | 7.4 | 53 | 22 | 9.3 | 52 | 22 | 0.8 | 13,23,16 | 5,9,8 |
| 30 | 9.1 | 183 | 27 | 34.2 | 388 | 21 | 10.0 | 83 | 17 | 0.7 | 16,61,6 | 7,7,3 |
| 31 | 14.3 | 151 | 41 | 11.6 | 87 | 31 | 18.8 | 122 | 19 | 0.7 | 99,15,8 | 9,6,4 |
| 32 | 10.0 | 99 | 42 | 183.9 | 2318 | 27 | 23.3 | 97 | 42 | 0.8 | 55,24,18 | 23,10,9 |
| 33 | 9.4 | 97 | 43 | 13.9 | 98 | 41 | 14.8 | 81 | 37 | 0.7 | 18,24,39 | 8,10,19 |
| 34 | 3.2 | 41 | 16 | 3.6 | 28 | 12 | 4.5 | 30 | 12 | 0.7 | 6,16,8 | 2,6,4 |
| 35 | 8.3 | 101 | 22 | 10.8 | 79 | 33 | 15.4 | 103 | 22 | 0.7 | 69,26,8 | 9,9,4 |
| 36 | 6.9 | 76 | 31 | 8.0 | 54 | 25 | 15.9 | 78 | 31 | 0.8 | 34,34,10 | 13,13,5 |
| 37 | 1.4 | 20 | 8 | 15.6 | 259 | 19 | 2.8 | 18 | 7 | 0.6 | 4,6,8 | 1,2,4 |
| 38 | 45.1 | 927 | 24 | 21.3 | 415 | 33 | 20.9 | 327 | 21 | 0.7 | 16,301,10 | 7,9,5 |
| 39 | 6.0 | 73 | 26 | 16.6 | 164 | 28 | 13.2 | 75 | 27 | 0.7 | 33,30,12 | 10,11,6 |
| 40 | 7.9 | 99 | 33 | 8.9 | 70 | 29 | 16.4 | 101 | 33 | 0.6 | 53,32,16 | 13,12,8 |
| 41 | 4.8 | 61 | 25 | 7.9 | 63 | 26 | 5.8 | 41 | 16 | 0.6 | 8,21,12 | 3,7,6 |
| 42 | 3.9 | 51 | 20 | 4.8 | 38 | 18 | 9.1 | 50 | 20 | 0.7 | 19,23,8 | 7,9,4 |
| 43 | 6.1 | 65 | 21 | 16.7 | 142 | 18 | 13.2 | 48 | 21 | 0.9 | 30,8,10 | 13,3,5 |
| 44 | 5.8 | 65 | 26 | 8.2 | 77 | 17 | 8.9 | 42 | 18 | 0.7 | 16,12,14 | 7,4,7 |

| Prob | Prodigy | | | Prodigy + Abstrips | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 45 | 4.1 | 46 | 19 | 7.5 | 48 | 20 | 9.0 | 50 | 20 | 0.6 | 15,18,17 | 5,7,8 |
| 46 | 4.3 | 62 | 20 | 6.8 | 52 | 21 | 7.9 | 58 | 18 | 0.5 | 30,18,10 | 5,8,5 |
| 47 | 6.7 | 81 | 21 | 11.5 | 80 | 30 | 9.4 | 51 | 22 | 0.6 | 15,16,20 | 5,7,10 |
| 48 | 2.0 | 29 | 12 | 3.2 | 30 | 13 | 4.0 | 29 | 12 | 0.6 | 4,15,10 | 1,6,5 |
| 49 | 16.8 | 408 | 18 | 19.0 | 392 | 13 | 19.0 | 319 | 18 | 0.5 | 14,297,8 | 5,9,4 |
| 50 | 18.3 | 414 | 36 | 18.3 | 284 | 31 | 9.1 | 132 | 18 | 0.6 | 6,112,14 | 2,9,7 |
| 51 | 4.1 | 58 | 16 | 12.3 | 175 | 19 | 8.1 | 72 | 14 | 0.8 | 10,46,16 | 3,4,7 |
| 52 | 6.9 | 89 | 31 | 10.3 | 83 | 33 | 10.6 | 69 | 30 | 0.8 | 13,23,33 | 5,10,15 |
| 53 | 11.1 | 109 | 42 | 13.0 | 97 | 36 | 21.7 | 93 | 37 | 1.2 | 54,15,24 | 19,6,12 |
| 54 | 4.0 | 49 | 21 | 6.8 | 47 | 20 | 7.6 | 41 | 17 | 0.9 | 12,21,8 | 5,8,4 |
| 55 | 10.4 | 113 | 46 | 150.5 | 1801 | 53 | 23.2 | 111 | 46 | 1.1 | 58,21,32 | 21,9,16 |
| 56 | 22.0 | 480 | 29 | 24.6 | 491 | 23 | 11.6 | 189 | 19 | 0.7 | 4,174,11 | 1,13,5 |
| 57 | 15.5 | 173 | 48 | 9.4 | 74 | 31 | 31.9 | 228 | 32 | 1.0 | 183,33,12 | 14,12,6 |
| 58 | 9.7 | 112 | 50 | 28.0 | 362 | 57 | 19.0 | 91 | 40 | 0.9 | 40,29,22 | 17,12,11 |
| 59 | 13.4 | 165 | 41 | 18.7 | 129 | 51 | 22.8 | 159 | 38 | 1.0 | 110,29,20 | 16,12,10 |
| 60 | 3.8 | 44 | 18 | 15.3 | 201 | 30 | 9.0 | 44 | 18 | 0.9 | 20,14,10 | 7,6,5 |
| 61 | 8.2 | 85 | 35 | 14.6 | 106 | 42 | 17.9 | 82 | 35 | 1.1 | 37,21,24 | 15,8,12 |
| 62 | 11.9 | 151 | 33 | 38.0 | 444 | 32 | 17.5 | 113 | 31 | 1.1 | 75,20,18 | 13,9,9 |
| 63 | 59.3 | 1141 | 42 | 548.4 | 5229 | 51 | 19.3 | 94 | 40 | 0.9 | 37,41,16 | 16,16,8 |
| 64 | 23.7 | 488 | 31 | 13.7 | 208 | 30 | 10.8 | 69 | 28 | 0.8 | 15,31,23 | 5,12,11 |
| 65 | 5.0 | 60 | 24 | 21.8 | 332 | 28 | 6.1 | 39 | 16 | 0.8 | 8,21,10 | 3,8,5 |
| 66 | 7.9 | 88 | 35 | 135.7 | 1533 | 28 | 14.7 | 79 | 32 | 0.8 | 39,18,22 | 13,8,11 |
| 67 | 16.1 | 192 | 46 | 489.0 | 7625 | 47 | 27.2 | 156 | 46 | 1.0 | 95,45,16 | 21,17,8 |
| 68 | 6.2 | 64 | 25 | 9.7 | 67 | 27 | 14.7 | 67 | 26 | 0.7 | 36,23,8 | 14,8,4 |
| 69 | 8.5 | 98 | 26 | 225.0 | 4273 | 24 | 10.0 | 56 | 23 | 0.8 | 15,20,21 | 5,8,10 |
| 70 | 5.6 | 58 | 24 | 4.6 | 32 | 15 | 6.4 | 43 | 18 | 0.8 | 6,22,15 | 2,9,7 |
| 71 | 7.5 | 92 | 40 | 12.4 | 96 | 41 | 14.7 | 95 | 40 | 0.8 | 14,46,35 | 6,18,16 |
| 72 | 7.4 | 64 | 27 | 69.8 | 734 | 30 | 13.4 | 50 | 23 | 0.8 | 26,2,22 | 11,1,11 |
| 73 | 3.2 | 46 | 20 | 3.6 | 30 | 14 | 5.5 | 39 | 16 | 0.7 | 4,23,12 | 1,9,6 |
| 74 | 7.5 | 93 | 39 | 13.4 | 112 | 45 | 10.4 | 73 | 31 | 0.8 | 10,35,28 | 4,14,13 |
| 75 | 4.6 | 52 | 22 | 6.8 | 48 | 21 | 8.8 | 55 | 22 | 0.8 | 8,28,19 | 3,10,9 |
| 76 | 5.6 | 57 | 24 | 80.6 | 1422 | 26 | 10.8 | 43 | 19 | 1.0 | 17,18,8 | 7,8,4 |
| 77 | 61.9 | 1168 | 44 | 136.4 | 1299 | 34 | 29.1 | 106 | 44 | 1.1 | 67,13,26 | 26,5,13 |
| 78 | 24.6 | 250 | 55 | 600.0 | 9947 | — | 33.2 | 183 | 40 | 1.5 | 141,18,24 | 21,7,12 |
| 79 | 12.5 | 145 | 43 | 20.7 | 137 | 56 | 25.1 | 147 | 43 | 1.0 | 100,27,20 | 21,12,10 |
| 80 | 13.0 | 219 | 35 | 415.5 | 3593 | 43 | 17.8 | 91 | 37 | 1.0 | 39,30,22 | 15,11,11 |
| 81 | 13.1 | 131 | 32 | 9.3 | 72 | 28 | 15.5 | 92 | 23 | 1.3 | 59,21,12 | 9,8,6 |
| 82 | 7.6 | 92 | 25 | 8.6 | 69 | 27 | 13.2 | 91 | 24 | 1.1 | 59,16,16 | 9,7,8 |
| 83 | 7.6 | 66 | 27 | 16.7 | 129 | 44 | 13.1 | 58 | 26 | 1.2 | 10,12,18,18 | 4,5,8,9 |
| 84 | 21.4 | 226 | 33 | 17.9 | 111 | 44 | 21.1 | 112 | 23 | 1.4 | 96,6,10 | 15,3,5 |
| 85 | 5.6 | 75 | 29 | 600.0 | 12506 | — | 12.5 | 98 | 38 | 1.0 | 8,62,28 | 3,22,13 |
| 86 | 42.6 | 883 | 52 | 15.8 | 117 | 50 | 29.5 | 203 | 45 | 1.1 | 66,125,12 | 26,13,6 |
| 87 | 6.1 | 89 | 31 | 10.1 | 81 | 35 | 10.7 | 72 | 31 | 0.9 | 12,38,22 | 4,16,11 |
| 88 | 14.5 | 259 | 25 | 83.0 | 1631 | 19 | 10.2 | 114 | 19 | 0.9 | 6,93,15 | 2,10,7 |

| Prob | Prodigy | | | Prodigy + Abstrips | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 89 | 6.0 | 75 | 30 | 98.8 | 2007 | 29 | 9.5 | 61 | 25 | 1.0 | 8,34,19 | 3,13,9 |
| 90 | 72.6 | 1534 | 31 | 52.1 | 596 | 33 | 13.4 | 135 | 28 | 1.0 | 11,97,27 | 4,11,13 |
| 91 | 8.2 | 86 | 37 | 270.2 | 2968 | 26 | 18.1 | 101 | 40 | 1.2 | 37,42,22 | 14,15,11 |
| 92 | 5.5 | 65 | 26 | 323.5 | 5696 | 25 | 12.0 | 69 | 27 | 1.5 | 8,10,31,20 | 3,4,10,10 |
| 93 | 2.6 | 38 | 17 | 6.3 | 65 | 21 | 7.2 | 44 | 19 | 1.0 | 8,26,10 | 3,11,5 |
| 94 | 12.2 | 127 | 44 | 16.3 | 109 | 41 | 22.8 | 107 | 37 | 1.1 | 69,22,16 | 20,9,8 |
| 95 | 18.3 | 184 | 46 | 114.7 | 1275 | 40 | 28.1 | 162 | 37 | 1.4 | 125,13,24 | 19,6,12 |
| 96 | 120.2 | 2460 | 40 | 158.3 | 3291 | 40 | 17.7 | 77 | 33 | 1.1 | 40,23,14 | 16,10,7 |
| 97 | 8.4 | 89 | 38 | 600.0 | 9209 | — | 18.8 | 89 | 38 | 1.1 | 44,23,22 | 17,10,11 |
| 98 | 8.4 | 87 | 37 | 14.7 | 106 | 35 | 19.9 | 86 | 37 | 1.4 | 45,19,22 | 19,7,11 |
| 99 | 101.5 | 2020 | 55 | 26.8 | 335 | 43 | 28.7 | 273 | 41 | 1.1 | 84,165,24 | 17,12,12 |
| 100 | 36.0 | 786 | 38 | 27.7 | 447 | 30 | 24.5 | 392 | 32 | 1.0 | 17,344,31 | 6,11,15 |
| 101 | 8.7 | 73 | 31 | 54.4 | 582 | 35 | 20.5 | 73 | 30 | 1.8 | 22,18,21,12 | 8,8,8,6 |
| 102 | 13.5 | 134 | 58 | 14.4 | 92 | 40 | 31.7 | 220 | 53 | 1.5 | 60,136,24 | 25,16,12 |
| 103 | 35.5 | 677 | 56 | 125.6 | 2326 | 40 | 17.4 | 81 | 34 | 1.1 | 42,23,16 | 16,10,8 |
| 104 | 4.2 | 44 | 19 | 23.4 | 306 | 26 | 9.2 | 43 | 18 | 1.3 | 19,12,12 | 7,5,6 |
| 105 | 30.2 | 665 | 32 | 10.9 | 75 | 33 | 13.3 | 71 | 31 | 1.2 | 24,29,18 | 9,13,9 |
| 106 | 8.1 | 77 | 33 | 7.4 | 47 | 19 | 14.6 | 69 | 30 | 1.1 | 25,26,18 | 10,11,9 |
| 107 | 75.3 | 1112 | 94 | 600.0 | 8977 | — | 51.0 | 410 | 49 | 1.3 | 345,41,24 | 20,17,12 |
| 108 | 8.6 | 82 | 33 | 100.7 | 1785 | 31 | 14.6 | 67 | 27 | 1.1 | 31,24,12 | 13,8,6 |
| 109 | 9.5 | 96 | 34 | 600.0 | 8762 | — | 21.3 | 107 | 37 | 1.1 | 65,22,20 | 19,8,10 |
| 110 | 11.8 | 106 | 34 | 165.0 | 1762 | 21 | 17.8 | 73 | 31 | 1.3 | 37,20,16 | 15,8,8 |
| 111 | 219.4 | 2995 | 31 | 15.3 | 104 | 43 | 21.6 | 109 | 35 | 1.8 | 63,24,22 | 14,10,11 |
| 112 | 7.2 | 80 | 34 | 14.5 | 105 | 44 | 13.7 | 75 | 32 | 1.2 | 19,27,29 | 7,11,14 |
| 113 | 10.4 | 138 | 47 | 50.0 | 826 | 51 | 14.3 | 114 | 33 | 1.2 | 14,70,30 | 6,13,14 |
| 114 | 7.2 | 86 | 35 | 152.5 | 1878 | 36 | 15.0 | 77 | 31 | 1.3 | 32,29,16 | 12,11,8 |
| 115 | 10.8 | 127 | 33 | 600.0 | 9792 | — | 28.8 | 594 | 22 | 1.1 | 10,566,18 | 4,9,9 |
| 116 | 8.8 | 97 | 39 | 15.6 | 119 | 45 | 14.8 | 74 | 29 | 1.1 | 32,26,16 | 12,9,8 |
| 117 | 8.8 | 80 | 31 | 8.0 | 47 | 22 | 17.7 | 74 | 30 | 1.4 | 42,18,14 | 16,7,7 |
| 118 | 9.5 | 109 | 47 | 600.0 | 10149 | — | 16.4 | 96 | 39 | 1.1 | 30,44,22 | 11,17,11 |
| 119 | 7.4 | 88 | 29 | 600.0 | 10161 | — | 12.9 | 67 | 25 | 1.1 | 30,23,14 | 10,8,7 |
| 120 | 6.7 | 77 | 32 | 9.5 | 75 | 34 | 13.8 | 75 | 29 | 1.4 | 27,34,14 | 10,12,7 |
| 121 | 4.1 | 48 | 21 | 180.8 | 2073 | 31 | 8.2 | 47 | 21 | 1.4 | 8,25,14 | 3,11,7 |
| 122 | 8.7 | 103 | 37 | 12.4 | 87 | 37 | 19.6 | 104 | 37 | 1.1 | 42,46,16 | 17,12,8 |
| 123 | 36.1 | 333 | 65 | 25.1 | 151 | 63 | 35.7 | 181 | 44 | 1.4 | 144,19,18 | 27,8,9 |
| 124 | 93.0 | 1357 | 79 | 600.0 | 7919 | — | 70.6 | 529 | 59 | 1.4 | 468,39,22 | 31,17,11 |
| 125 | 12.6 | 124 | 49 | 18.1 | 119 | 42 | 28.5 | 126 | 50 | 1.6 | 67,41,18 | 25,16,9 |
| 126 | 9.0 | 77 | 33 | 166.6 | 1240 | 46 | 22.0 | 81 | 34 | 2.0 | 22,18,27,14 | 8,8,11,7 |
| 127 | 17.9 | 157 | 67 | 600.0 | 8442 | — | 35.1 | 144 | 61 | 1.7 | 77,41,26 | 31,17,13 |
| 128 | 35.3 | 677 | 56 | 243.7 | 4503 | 41 | 17.7 | 81 | 34 | 1.2 | 42,23,16 | 16,10,8 |
| 129 | 8.2 | 75 | 33 | 27.4 | 320 | 33 | 16.4 | 73 | 30 | 1.5 | 34,25,14 | 13,10,7 |
| 130 | 51.6 | 1084 | 43 | 600.0 | 11080 | — | 20.4 | 103 | 43 | 1.5 | 39,44,20 | 15,18,10 |
| 131 | 8.3 | 77 | 33 | 7.9 | 51 | 21 | 14.6 | 69 | 30 | 1.3 | 25,26,18 | 10,11,9 |
| 132 | 124.1 | 1798 | 123 | 600.0 | 9131 | — | 53.0 | 416 | 52 | 1.5 | 347,43,26 | 21,18,13 |

| Prob | Prodigy | | | Prodigy + Abstrips | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 133 | 11.2 | 110 | 40 | 108.2 | 1793 | 36 | 16.9 | 71 | 29 | 1.3 | 37,20,14 | 15,7,7 |
| 134 | 10.7 | 113 | 40 | 600.0 | 8619 | — | 22.1 | 113 | 40 | 1.3 | 65,26,22 | 19,10,11 |
| 135 | 15.9 | 124 | 41 | 197.0 | 2048 | 23 | 23.5 | 86 | 36 | 1.6 | 50,18,18 | 20,7,9 |
| 136 | 18.3 | 190 | 56 | 138.8 | 1383 | 40 | 22.8 | 109 | 35 | 2.2 | 63,24,22 | 14,10,11 |
| 137 | 8.1 | 82 | 35 | 600.0 | 10584 | — | 17.5 | 79 | 34 | 1.4 | 36,27,16 | 15,11,8 |
| 138 | 12.2 | 162 | 57 | 44.0 | 694 | 50 | 15.2 | 118 | 35 | 1.4 | 14,72,32 | 6,14,15 |
| 139 | 12.4 | 135 | 54 | 257.1 | 3142 | 44 | 25.5 | 125 | 49 | 1.5 | 61,42,22 | 22,16,11 |
| 140 | 12.6 | 146 | 40 | 600.0 | 10072 | — | 36.2 | 729 | 26 | 1.3 | 10,701,18 | 4,13,9 |
| 141 | 9.0 | 97 | 39 | 15.9 | 123 | 46 | 14.8 | 74 | 29 | 1.3 | 32,26,16 | 12,9,8 |
| 142 | 9.4 | 84 | 33 | 8.9 | 53 | 25 | 18.5 | 78 | 32 | 1.6 | 42,20,16 | 16,8,8 |
| 143 | 9.7 | 109 | 47 | 600.0 | 10203 | — | 16.7 | 96 | 39 | 1.3 | 30,44,22 | 11,17,11 |
| 144 | 21.0 | 426 | 41 | 600.0 | 9803 | — | 24.7 | 320 | 35 | 1.2 | 30,270,20 | 10,15,10 |
| 145 | 11.5 | 197 | 37 | 13.0 | 105 | 38 | 14.6 | 79 | 31 | 1.6 | 27,36,16 | 10,13,8 |
| 146 | 4.5 | 48 | 21 | 180.0 | 2073 | 31 | 8.8 | 47 | 21 | 1.8 | 8,25,14 | 3,11,7 |
| 147 | 9.6 | 105 | 38 | 13.2 | 91 | 39 | 20.8 | 106 | 38 | 1.3 | 44,46,16 | 18,12,8 |
| 148 | 40.2 | 358 | 76 | 29.0 | 178 | 74 | 44.2 | 206 | 55 | 1.6 | 163,23,20 | 35,10,10 |
| 149 | 105.0 | 1618 | 79 | 600.0 | 7979 | — | 70.3 | 529 | 59 | 1.8 | 468,39,22 | 31,17,11 |
| 150 | 600.0 | 5571 | — | 600.0 | 7979 | — | 600.0 | 4923 | — | 1.8 | | |
| 151 | 9.5 | 85 | 37 | 95.6 | 854 | 46 | 23.5 | 89 | 38 | 2.2 | 22,18,33,16 | 8,8,14,8 |
| 152 | 600.0 | 9082 | — | 600.0 | 8372 | — | 53.7 | 234 | 82 | 2.0 | 148,56,30 | 44,23,15 |
| 153 | 36.6 | 683 | 59 | 600.0 | 11730 | — | 20.3 | 87 | 37 | 1.5 | 46,25,16 | 18,11,8 |
| 154 | 28.7 | 531 | 47 | 30.7 | 360 | 39 | 22.5 | 94 | 39 | 1.7 | 49,29,16 | 19,12,8 |
| 155 | 77.1 | 1495 | 49 | 600.0 | 11079 | — | 22.5 | 107 | 45 | 1.7 | 45,40,22 | 17,17,11 |
| 156 | 8.3 | 77 | 33 | 8.3 | 55 | 23 | 15.0 | 69 | 30 | 1.5 | 25,26,18 | 10,11,9 |
| 157 | 123.4 | 1798 | 123 | 600.0 | 9241 | — | 53.1 | 416 | 52 | 1.9 | 347,43,26 | 21,18,13 |
| 158 | 13.9 | 133 | 49 | 184.0 | 3033 | 43 | 18.0 | 75 | 31 | 1.5 | 39,20,16 | 16,7,8 |
| 159 | 10.9 | 115 | 41 | 600.0 | 8556 | — | 23.8 | 117 | 42 | 1.8 | 67,26,24 | 20,10,12 |
| 160 | 16.5 | 128 | 43 | 248.4 | 2832 | 30 | 23.8 | 90 | 38 | 1.7 | 50,20,20 | 20,8,10 |
| 161 | 20.7 | 202 | 62 | 232.0 | 2305 | 51 | 26.4 | 123 | 42 | 2.4 | 67,28,28 | 16,12,14 |
| 162 | 11.8 | 111 | 45 | 600.0 | 9502 | — | 23.8 | 102 | 42 | 1.5 | 57,27,18 | 21,12,9 |
| 163 | 12.7 | 164 | 58 | 45.5 | 715 | 55 | 19.9 | 120 | 36 | 1.9 | 14,16,72,18 | 6,7,14,9 |
| 164 | 12.5 | 139 | 56 | 206.2 | 2476 | 54 | 27.2 | 129 | 51 | 1.7 | 61,44,24 | 22,17,12 |
| 165 | 30.3 | 593 | 55 | 600.0 | 10104 | — | 57.2 | 1150 | 41 | 1.5 | 10,1112,28 | 4,23,14 |
| 166 | 12.6 | 129 | 52 | 600.0 | 10541 | — | 23.9 | 106 | 42 | 1.8 | 51,37,18 | 20,13,9 |
| 167 | 10.1 | 98 | 38 | 9.4 | 57 | 27 | 19.1 | 82 | 34 | 1.8 | 42,22,18 | 16,9,9 |
| 168 | 10.4 | 111 | 48 | 600.0 | 10197 | — | 17.4 | 98 | 40 | 1.5 | 30,46,22 | 11,18,11 |
| 169 | 27.8 | 567 | 51 | 600.0 | 9948 | — | 37.6 | 569 | 40 | 1.6 | 36,511,22 | 13,16,11 |
| 170 | 11.3 | 197 | 37 | 49.5 | 555 | 39 | 14.9 | 79 | 31 | 1.8 | 27,36,16 | 10,13,8 |
| 171 | 18.8 | 355 | 44 | 600.0 | 11812 | — | 17.6 | 84 | 36 | 2.0 | 27,37,20 | 11,15,10 |
| 172 | 600.0 | 7218 | — | 600.0 | 11812 | — | 600.0 | 4921 | — | 1.8 | | |
| 173 | 40.0 | 358 | 76 | 600.0 | 7641 | — | 46.7 | 206 | 55 | 1.8 | 163,23,20 | 35,10,10 |
| 174 | 118.3 | 1879 | 79 | 600.0 | 7916 | — | 67.5 | 529 | 59 | 2.2 | 468,39,22 | 31,17,11 |
| 175 | 600.0 | 5512 | — | 600.0 | 7916 | — | 600.0 | 5020 | — | 2.1 | | |
| 176 | 187.6 | 3783 | 59 | 600.0 | 10266 | — | 49.5 | 598 | 53 | 1.9 | 75,503,20 | 23,20,10 |

| Prob | Prodigy | | | Prodigy + Abstrips | | | Prodigy + Alpine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Num | Time | Nodes | Len | Time | Nodes | Len | Time | Nodes | Len | ACT | AbNodes | AbLen |
| 177 | 107.8 | 2305 | 74 | 144.3 | 1207 | 82 | 34.4 | 486 | 44 | 1.8 | 23,428,35 | 9,18,17 |
| 178 | 115.0 | 2650 | 56 | 418.1 | 4936 | 65 | 25.1 | 340 | 37 | 1.7 | 16,294,30 | 6,16,15 |
| 179 | 180.0 | 3656 | 90 | 600.0 | 10640 | — | 36.5 | 327 | 44 | 2.3 | 101,212,14 | 19,18,7 |
| 180 | 41.2 | 820 | 68 | 131.6 | 1143 | 36 | 47.4 | 667 | 46 | 2.1 | 80,569,18 | 21,16,9 |
| 181 | 53.7 | 1003 | 54 | 78.6 | 1065 | 49 | 41.2 | 340 | 51 | 2.4 | 93,225,22 | 25,15,11 |
| 182 | 34.2 | 732 | 55 | 600.0 | 11400 | — | 20.6 | 85 | 36 | 2.8 | 14,16,41,14 | 6,7,16,7 |
| 183 | 20.8 | 293 | 65 | 600.0 | 9194 | — | 38.5 | 289 | 60 | 1.7 | 68,199,22 | 28,21,11 |
| 184 | 18.0 | 149 | 60 | 600.0 | 7006 | — | 35.9 | 134 | 56 | 2.1 | 85,29,20 | 34,12,10 |
| 185 | 59.9 | 1245 | 39 | 600.0 | 10137 | — | 30.4 | 283 | 43 | 2.2 | 77,186,20 | 13,20,10 |
| 186 | 20.7 | 195 | 78 | 342.9 | 4956 | 29 | 40.1 | 144 | 61 | 2.8 | 90,34,20 | 37,14,10 |
| 187 | 12.4 | 111 | 49 | 600.0 | 11646 | — | 20.3 | 86 | 37 | 1.7 | 43,21,22 | 17,9,11 |
| 188 | 402.9 | 7733 | 69 | 600.0 | 6821 | — | 52.9 | 747 | 52 | 2.2 | 47,676,24 | 20,20,12 |
| 189 | 10.7 | 108 | 42 | 17.9 | 150 | 49 | 21.8 | 105 | 42 | 2.4 | 43,40,22 | 15,16,11 |
| 190 | 69.9 | 1324 | 67 | 600.0 | 9674 | — | 45.4 | 400 | 46 | 1.9 | 195,183,22 | 19,16,11 |
| 191 | 11.6 | 121 | 51 | 472.3 | 3234 | 68 | 19.3 | 100 | 44 | 1.7 | 29,43,28 | 13,17,14 |
| 192 | 600.0 | 8395 | — | 600.0 | 10231 | — | 32.2 | 181 | 48 | 2.7 | 117,48,16 | 22,18,8 |
| 193 | 459.0 | 8474 | 131 | 267.9 | 2678 | 56 | 44.5 | 207 | 59 | 2.5 | 147,42,18 | 35,15,9 |
| 194 | 131.8 | 1952 | 68 | 134.3 | 1348 | 51 | 31.6 | 122 | 55 | 2.3 | 53,45,24 | 24,19,12 |
| 195 | 46.5 | 571 | 101 | 460.0 | 5863 | 63 | 29.0 | 136 | 44 | 2.1 | 92,22,22 | 23,10,11 |
| 196 | 15.6 | 145 | 57 | 600.0 | 10000 | — | 28.9 | 125 | 48 | 1.7 | 75,28,22 | 24,13,11 |
| 197 | 33.8 | 439 | 42 | 600.0 | 8055 | — | 31.0 | 164 | 38 | 2.3 | 126,24,14 | 20,11,7 |
| 198 | 10.2 | 98 | 41 | 15.9 | 113 | 46 | 17.6 | 88 | 39 | 1.7 | 27,41,20 | 12,17,10 |
| 199 | 354.0 | 8068 | 81 | 600.0 | 10279 | — | 244.7 | 4987 | 54 | 1.6 | 30,4925,32 | 12,26,16 |
| 200 | 133.6 | 2898 | 91 | 141.0 | 1799 | 59 | 47.4 | 874 | 45 | 1.7 | 22,814,38 | 8,18,19 |

# Bibliography

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[Aho *et al.*, 1983] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press, Edinburgh, Scotland, 1968.

[Amarel, 1984] Saul Amarel. Expert behaviour and problem representations. In *Artificial and Human Intelligence*, pages 1–41. North-Holland, New York, NY, 1984.

[Anderson and Farley, 1988] John S. Anderson and Arthur M. Farley. Plan abstraction based on operator generalization. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 100–104, Saint Paul, Minnesota, 1988.

[Anzai and Simon, 1979] Yuichiro Anzai and Herbert A. Simon. The theory of learning by doing. *Psychological Review*, 86:124–140, 1979.

[Banerji and Ernst, 1977a] Ranan B. Banerji and George W. Ernst. A comparison of three problem-solving methods. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 442–449, Cambridge, MA, 1977.

[Banerji and Ernst, 1977b] Ranan B. Banerji and George W. Ernst. Some properties of GPS-type problem solvers. Technical Report 1179, Department of Computer Engineering, Case Western Reserve University, 1977.

[Benjamin *et al.*, 1990] Paul Benjamin, Leo Dorst, Indur Mandhyan, and Madeleine Rosar. An introduction to the decomposition of task representations in autonomous systems. In D. Paul Benjamin, editor, *Change of Representation and Inductive Bias*, pages 125–146. Kluwer, Boston, MA, 1990.

[Campbell, 1988] Murray S. Campbell. *Chunking as an Abstraction Mechanism.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[Carbonell and Gil, 1990] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In *Machine Learning, An Artificial Intelligence Approach, Volume III*, pages 191–213. Morgan Kaufman, San Mateo, CA, 1990.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 241–278. Lawrence Erlbaum, Hillsdale, NJ, 1991.

[Carbonell, 1986] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In *Machine Learning, An Artificial Intelligence Approach, Volume II*, pages 371–392. Morgan Kaufman, San Mateo, CA, 1986.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[Cheng and Irani, 1989] Jie Cheng and Keki B. Irani. Ordering problem subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 931–936, Detroit, MI, 1989.

[Christensen, 1990] Jens Christensen. A hierarchical planner that creates its own hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, Boston, MA, 1990.

[Eavarone, 1969] Daniel S. Eavarone. A program that generates difference orderings for GPS. Technical Report SRC-69-6, Systems Research Center, Case Western Reserve University, 1969.

[Ernst and Goldstein, 1982] George W. Ernst and Michael M. Goldstein. Mechanical discovery of classes of problem-solving strategies. *Journal of the Association for Computing Machinery*, 29(1):1–23, 1982.

[Ernst and Newell, 1969] George W. Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving.* ACM Monograph Series. Academic Press, New York, NY, 1969.

[Ernst, 1969] George W. Ernst. Sufficient conditions for the success of GPS. *Journal of the Association for Computing Machinery*, 16(4):517–533, 1969.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Available as Technical Report CMU-CS-90-185.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[Flann, 1989] Nicholas S. Flann. Learning appropriate abstractions for planning in formation problems. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 235–239, Ithaca, NY, 1989.

[Giunchiglia and Walsh, 1990] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. Technical Report 9001-14, Istituto per la Ricerca Scientifica e Technologica, Trento, Italy, 1990.

[Goldstein, 1978] Michael M. Goldstein. The mechanical discovery of problem solving strategies. Technical Report ESCI-77-1, Systems Engineering, Computer Engineering and Information Sciences, Case Western Reserve University, 1978.

[Guvenir and Ernst, 1990] H. Altay Guvenir and George W. Ernst. Learning problem solving strategies using refinement and macro generation. *Artificial Intelligence*, 44(1-2):209–243, 1990.

[Hansson and Mayer, 1989] Othar Hansson and Andrew Mayer. Subgoal generation from problem relaxation. In *Proceedings of the AAAI Symposium on Planning and Search*, 1989.

[Hobbs, 1985] Jerry R. Hobbs. Granularity. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 432–435, Los Angeles, CA, 1985.

[Iba, 1989] Glenn A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317, 1989.

[Irani and Cheng, 1987] Keki B. Irani and Jie Cheng. Subgoal ordering and goal augmentation for heuristic problem solving. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1018–1024, Milan, Italy, 1987.

[Joseph, 1989] Robert L. Joseph. Graphical knowledge acquisition. In *Proceedings of the Fourth Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada, 1989.

[Joslin and Roach, 1989] David Joslin and John Roach. A theoretical analysis of conjunctive-goal problems. *Artificial Intelligence*, 41(1):97–106, 1989.

[Kaplan and Simon, 1990] Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, 22:374–419, 1990.

[Kibler, 1985] Dennis Kibler. Natural generation of heuristics by transforming the problem representation. Technical Report TR-85-20, Department of Computer Science, University of California at Irvine, 1985.

[Knoblock *et al.*, 1990] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. A spectrum of abstraction hierarchies for planning. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, Boston, MA, 1990.

[Knoblock *et al.*, 1991a] Craig A. Knoblock, Steven Minton, and Oren Etzioni. Integrating abstraction and explanation-based learning in PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 541–546, Anaheim, CA, 1991.

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

[Knoblock, 1990a] Craig A. Knoblock. Abstracting the Tower of Hanoi. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, pages 13–23, Boston, MA, 1990.

[Knoblock, 1990b] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, Boston, MA, 1990.

[Knoblock, 1990c] Craig A. Knoblock. A theory of abstraction for hierarchical planning. In D. Paul Benjamin, editor, *Change of Representation and Inductive Bias*, pages 81–104. Kluwer, Boston, MA, 1990.

[Knoblock, 1991] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, Anaheim, CA, 1991.

[Korf, 1980] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14:41–78, 1980.

[Korf, 1985a] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Korf, 1985b] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[Lifschitz, 1986] Vladimir Lifschitz. On the semantics of STRIPS. In *Proceedings of the Workshop on Reasoning about Actions and Plans*, pages 1–9, Timberline, Oregon, 1986.

[McCarthy, 1964] John McCarthy. A tough nut for proof procedures. Stanford Artificial Intelligence Project Memo 16, Computer Science Department, Stanford University, 1964.

[McDermott, 1982] John McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[Minsky, 1963] Marvin Minsky. Steps toward artificial intelligence. In Edward A. Feigenbaum, editor, *Computers and Thought*, pages 406–450. McGraw-Hill, New York, NY, 1963.

[Minton *et al.*, 1989a] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1-3):63–118, 1989.

[Minton *et al.*, 1989b] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[Minton, 1985] Steven Minton. Selectively generalizing plans for problem solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 596–599, Los Angeles, CA, 1985.

[Minton, 1988a] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[Minton, 1988b] Steven Minton. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer, Boston, MA, 1988.

[Minton, 1990] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3):363–392, 1990.

[Mostow and Prieditis, 1989] Jack Mostow and Armand E. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 701–707, Detroit, MI, 1989.

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

[Newell *et al.*, 1962] Allen Newell, J. C. Shaw, and Herbert A. Simon. The processes of creative thinking. In *Contemporary Approaches to Creative Thinking*, pages 63–119. Atherton Press, New York, 1962.

[Nilsson, 1971] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, NY, 1971.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, 1980.

[Pearl, 1984] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[Plaisted, 1981] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16(1):47–108, 1981.

[Polya, 1945] George Polya. *How to Solve It.* Princeton University Press, Princeton, NJ, 1945.

[Riddle, 1990] Patricia Riddle. Automating problem reformulation. In D. Paul Benjamin, editor, *Change of Representation and Inductive Bias*, pages 105–124. Kluwer, Boston, MA, 1990.

[Rosenbloom *et al.*, 1985] Paul S. Rosenbloom, John E. Laird, John McDermott, Allen Newell, and E. Orciuch. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561–569, 1985.

[Rosenschein, 1981] Stanley J. Rosenschein. Plan synthesis: A logical perspective. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 331–337, Vancouver, B.C., Canada, 1981.

[Ruby and Kibler, 1989] David Ruby and Dennis Kibler. Learning subgoal sequences for planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 609–614, Detroit, MI, 1989.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

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

[Segre *et al.*, 1991] Alberto Segre, Charles Elkan, and Alex Russell. A critical look at experimental evaluations of EBL. *Machine Learning*, 6(2):183–195, 1991.

[Shell and Carbonell, 1989] Peter Shell and Jaime Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 596–602, Detroit, MI, 1989.

[Simon, 1977] Herbert A. Simon. The theory of problem solving. In Herbert A. Simon, editor, *Models of Discovery*, chapter 4.3, pages 214–244. D. Reidel, Boston, MA, 1977.

[Stallman and Sussman, 1977] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[Stefik, 1981] Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.

[Subramanian and Genesereth, 1987] Devika Subramanian and Michael R. Genesereth. The relevance of irrelevance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, 1987.

[Subramanian, 1989] Devika Subramanian. *A Theory of Justified Reformulations.* PhD thesis, Department of Computer Science, Stanford University, 1989.

[Tate, 1976] Austin Tate. Project planning using a hierarchic non-linear planner. Research Report 25, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, 1976.

[Tenenberg, 1988] Josh D. Tenenberg. *Abstraction in Planning.* PhD thesis, Computer Science Department, University of Rochester, 1988.

[Unruh and Rosenbloom, 1989] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 681–687, Detroit, MI, 1989.

[Unruh and Rosenbloom, 1990] Amy Unruh and Paul S. Rosenbloom. Two new weak method increments for abstraction. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, pages 78–86, Boston, MA, 1990.

[VanLehn, 1989] Kurt VanLehn. Discovering problem solving strategies: What humans do and machines don't (yet). In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 215–217, Ithaca, NY, 1989.

[Veloso and Carbonell, 1990] Manuela M. Veloso and Jaime G. Carbonell. Integrating analogy into a general problem-solving architecture. In *Intelligent Systems*. Ellis Horwood Limited, West Sussex, England, 1990.

[Weld and Addanki, 1990] Daniel S. Weld and Sanjaya Addanki. Task-driven model abstraction. In *Proceedings of the Fourth International Workshop on Qualitative Physics*, Lugano, Switzerland, 1990.

[Wilkins, 1984] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.

[Wilkins, 1986] David E. Wilkins. High-level planning in a mobile robot domain. Technical Report 388, Artificial Intelligence Center, SRI International, 1986.

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

[Wilkins, 1989] David E. Wilkins. Can AI planners solve practical problems? Technical Report 468R, Artificial Intelligence Center, SRI International, 1989.

[Yang and Tenenberg, 1990] Qiang Yang and Josh D. Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 204–209, Boston, MA, 1990.

[Yang, 1989] Qiang Yang. *Improving the Efficiency of Planning*. PhD thesis, Department of Computer Science, University of Maryland, 1989.

[Yang, 1990] Qiang Yang. Solving the problem of hierarchical inaccuracy in planning. In *Proceedings of the Eighth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 140–145, Ottawa, Canada, 1990.