To appear in *Artificial Intelligence*

Automatically Generating Abstractions for Planning

Craig A. Knoblock
Information Sciences Institute
& Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
Email: knoblock@isi.edu

Abstract

This article presents a completely automated approach to generating abstractions for planning. The abstractions are generated using a tractable, domain-independent algorithm whose only input is the definition of a problem to be solved and whose output is an abstraction hierarchy that is tailored to the particular problem. The algorithm generates abstraction hierarchies by dropping literals from the original problem definition. It forms abstractions that satisfy the *ordered monotonicity* property, which guarantees that the structure of an abstract solution is not changed in the process of refining it. 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 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 planning without using abstraction.

1 Introduction

General-purpose planning systems often solve problems by forging ahead, blindly addressing central and peripheral issues alike, without any attempt to decompose a problem and determine which parts should be solved first. This can result in a significant amount of wasted effort since a planner will spend time solving some aspect of a problem only to have to discard the solutions in the process of solving other aspects of the problem. Even for simple tasks, such as 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 planners are used in increasingly complex domains, the ability to decompose problems and focus on the more difficult aspects of a problem first becomes even more critical.

An effective approach to building more intelligent problem solvers is to use a set of abstractions for hierarchy planning in order to focus the search. In this paper the term "hierarchical planning" is used to refer to planners that use a distinct set of abstraction spaces to first solve a problem in an abstract space and then refine the abstract solution at successively more detailed levels in an abstraction hierarchy. This technique has been used successfully to reduce search in a number of planning systems, including GPS [48], ABSTRIPS [53], ABTWEAK [68], PABLO [11], and PRODIGY [32, 35].

While hierarchical planning is a widely used planning technique, there are only a few systems that automate the construction of abstraction hierarchies [3, 11, 53]. In most hierarchical planners, the designer of a planning domain must manually engineer the appropriate abstractions. This process is largely a black art since the properties of an effective abstraction hierarchy are not well understood. In addition, most existing hierarchical planners 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. The advantage of automatically generating abstraction hierarchies is that it frees the designer of a planning domain from concerns about efficiency and makes it practical to construct abstractions that are tailored to individual problems or classes of problems.

This article presents a tractable algorithm for automatically generating abstractions for hierarchical problem solving. The abstractions are based on the ordered monotonicity property, which guarantees that the structure of the abstract plan will be preserved while the plan is refined. This algorithm is implemented in the ALPINE system and the abstraction hierarchies generated by ALPINE are used in a version of the PRODIGY problem solver [9, 46] that was extended to plan hierarchically [35]. This article presents experimental results that demonstrate that ALPINE's abstractions provide significant reductions in search over planning without the use of abstraction.

1.1 Hierarchical Planning

Planning involves finding a sequence of operators that solves a problem within a problem space. A problem space is defined by the set of legal operators, where each operator consists of preconditions and effects. The preconditions must be satisfied before an operator can be applied, and the effects describe the changes to the state in which the operator is applied. A problem consists of an initial state and a set of goal conditions. A solution to a problem is a sequence of operators that transform the given initial state into a final state that satisfies the goal conditions.

Hierarchical planners[1] employ one or more abstractions of a problem space to reduce search. Instead of attempting to solve problems in the original problem space, a hierarchical planner first solves a problem in a simpler abstract space and then refines the abstract solution at successive levels of detail by inserting operators to achieve the conditions that were ignored in the more abstract spaces.

The potential search reduction of hierarchical planning is significant. It can reduce the size of the search space from exponential to linear in the size of the solution under certain assumptions. For single-level planning the size of the search space is exponential in the solution length. Hierarchical planning reduces this complexity by taking a large complex problem and decomposing it into a number of smaller subproblems. See [33, 35] for a formal definition of hierarchical planning and an analysis of the search reduction. In addition, hierarchical planning can improve the performance of a learning system. For an example see [36], which describes the integration of abstraction and explanation-based learning in the context of the PRODIGY problem solver.

1.2 Abstraction Hierarchies

While hierarchical planning has been used in a variety of planners to reduce search, the problem of how to find effective abstractions has not received as much attention. In most of the existing hierarchical planners, 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. Ideally one would like a simple and tractable criterion for generating the abstractions of a problem space. This article takes a major step in this direction by defining a heuristic criterion for identifying useful abstraction hierarchies and providing a polynomial-time algorithm for automatically generating abstractions that meet this criterion.

In this article, an *abstraction space* is formed by dropping certain terms from the language of a problem space. In the resulting abstraction space, a single abstract state corresponds to one or more states in the original problem space. An ordered sequence of abstraction spaces defines an *abstraction hierarchy*, where each successive abstraction space is an abstraction of the previous one.

The use of an abstraction hierarchy for hierarchical problem solving reduces search by partitioning a problem into a number of simpler subproblems. This reduction in search comes from the assumption that the subproblems are smaller and can be solved without violating the conditions achieved at the higher levels in the abstraction hierarchy. Thus, an abstraction hierarchy needs to partition a problem such that the parts of a problem that are solved in an abstract space can be held invariant while the remaining parts of a problem are solved. This property is captured by the *ordered monotonicity* property:

[1]The terms "hierarchy" and "abstraction" have been used in a number of different ways in the planning literature. For a discussion of this issue see [65, chapter 4], and for a discussion of systems using various forms of "hierarchical abstraction" see [60].

Ordered Monotonicity Property: *For all* abstract plans, *all* refinements of those plans leave the literals established in the abstract space unchanged.

This property captures an important feature of abstraction spaces and can be used to generate abstraction hierarchies. However, it is heuristic since it does not guarantee that a refinement of an abstract plan exists.

This article formally defines the ordered monotonicity property. First, it formalizes the process by which abstract plans are refined. Then the article identifies a restriction on this refinement process called *ordered refinement*, which requires that a refinement leaves the literals established in the abstract space unchanged. Finally, it defines an ordered monotonic abstraction hierarchy as a hierarchy in which every possible refinement is an ordered refinement.

An important feature of this property is that we can identify sets of constraints on the possible abstraction hierarchies that are sufficient to guarantee the ordered monotonicity property. The article first presents sufficient conditions to guarantee ordered monotonicity for every problem in a domain. Then it presents sufficient conditions to guarantee this property for a specific problem. The latter set of constraints is useful for generating problem-specific abstraction hierarchies.

1.3 Automatically Generating Abstractions

The sufficient conditions for ordered monotonicity serve as the basis of an algorithm for constructing hierarchies of abstraction spaces. This article presents a polynomial-time algorithm for automatically generating abstraction hierarchies from only the initial problem space definition and 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. The algorithm presented in this paper generates abstraction hierarchies that are tailored to the individual problems. For example, the STRIPS robot planning domain [20] 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 a hierarchical version of PRODIGY. Since ALPINE is generating abstractions for a particular planning system, it employs several planner-specific extensions to the basic algorithm in order to produce finer-grained abstraction hierarchies. The article describes these extensions in detail.

1.4 Experimental Results

ALPINE has been successfully tested on a number of planning domains including the Tower of Hanoi, the original STRIPS domain [20], a more complex robot planning domain [44], and a machine-shop process planning and scheduling domain [44]. In all these domains, the system generates problem-specific abstraction hierarchies that provide significant reductions in search. The algorithm for generating the abstractions is quite efficient and can generate an abstraction hierarchy for a problem in any of these domains in 0.3 to 4.5 CPU seconds.

The abstraction hierarchies generated by ALPINE were tested on a hierarchical version of the PRODIGY problem solver. PRODIGY 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 preserves both the problem-space language and control language of PRODIGY while providing the added functionality of hierarchical problem solving.

This article compares the performance of ALPINE's abstractions to other forms of control knowledge. First, it compares ALPINE's abstractions to the basic PRODIGY system and PRODIGY using hand-coded control knowledge. The results show that ALPINE reduces both solution time and solution length when compared with the basic PRODIGY system and performs comparably to hand-coded control knowledge. Second, it compares ALPINE's abstractions to the use of control knowledge acquired by explanation-based learning techniques [16, 44]. Again, the results show that ALPINE performs comparably to these techniques, but more importantly the combination of abstraction and control knowledge leads to performance that is better than any of the systems alone. Third, the article compares the abstractions produced by ALPINE to those generated by ABSTRIPS and shows that ALPINE produces significantly better abstractions than ABSTRIPS.

1.5 Outline

This article defines the ordered monotonicity property, presents the algorithms for generating abstraction based on this property, and describes the implemented system and results. The next section defines an abstraction space, presents the ordered monotonicity property, and identifies a set of sufficient conditions to guarantee this property. Section 3 presents the algorithms for automatically generating abstraction hierarchies that have the ordered monotonicity property. Section 4 describes the implementation of ALPINE, which produces problem-specific abstraction hierarchies using an extended version of this algorithm. Section 5 presents the empirical results for both generating and using the abstractions for problem solving. Section 6 compares and contrasts the work described here with other work related to generating and using abstractions for planning. Section 7 describes some of the limitations of this work and some extensions that address these limitations. The final section reviews the primary contributions of this work.

2 Abstraction Hierarchies

Abstraction hierarchies are used to guide the refinement of plans in a hierarchical problem solver. This section presents this refinement process and identifies a property of abstraction hierarchies that can be used to constrain this process. First we define problem spaces and abstraction hierarchies. Then we provide a precise definition of how abstract plans are refined. Next, we present the ordered monotonicity property. Finally, we identify a set of sufficient conditions for ordered monotonicity.

2.1 Problem Spaces

A *problem space* Σ 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*.[2] 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_α, D_α, A_α), where P_α, the preconditions, are a finite set of literals (positive or negative atomic formulas) in L, and both the deletes D_α and adds A_α are finite sets of atomic formulas in L. The combination of the adds and deletes comprise the effects of an operator E_α, 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 ρ consists of:

- an *initial state* $S_0 \in S$, which describes the initial configuration of the world, and
- a *goal* S_g is a partial description of a state and describes the desired configuration of the world.

The *solution* (or *plan*) Π 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 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.)

An *application* procedure $\mathcal{A}$ applies an operator α to a state S_i to produce a new state by first removing the deleted literals, and then inserting the added literals. For any state S_i (where '$\setminus$' represents set difference),

$$S_{i+1} = \mathcal{A}(\alpha, S_i) = (S_i \setminus D_\alpha) \cup A_\alpha$$

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 Π is *correct* whenever the preconditions of each operator are satisfied in the state in which the operator is applied:

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

[2] The formalization of problem solving presented in this section is loosely based on Lifschitz's formalization of STRIPS [42].

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

2.2 Abstraction Spaces and Hierarchies

In this paper, an *abstraction space* or *abstract problem space* is formed by dropping certain terms from the language of a problem space. In the resulting abstraction space, a single abstract state corresponds to one or more states in the original problem space. This type of abstraction space is called a *reduced model* [61]. A different approach was taken in ABSTRIPS [53], where the preconditions of the operators were assigned criticality values and all preconditions with criticality values below a certain threshold were ignored. These abstraction spaces are called *relaxed models* [50] since they are formed by weakening the applicability conditions of the operators. Both reduced and relaxed models are what Giunchiglia and Walsh [24] refer to as TI (theorem increasing) abstractions since any theorem that holds in the ground space will hold in the abstract space, while the inverse does not hold. For the purposes of this paper, an abstraction space will be used to refer to a reduced model unless stated otherwise.

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 terms from the language of 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 similar 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.

A k-level abstraction hierarchy is defined by the initial problem space $\Sigma = (L, S, O)$, 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:

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

Using the definition of an abstraction hierarchy, we can define a set of functions that map states, operators, plans, and problems from one problem space into an abstraction space. $\mathcal{M}_s^i(s)$ is a state mapping function that maps a state s at level j to a state at level i, where $j < i$. $\mathcal{M}_o^i(\alpha)$ is an operator mapping function that maps an operator α at level j to an operator at level i, where $j < i$. 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 plan Π at level j to a plan at level i, where $j < i$, by replacing each operator α in Π by $\mathcal{M}^i_o(\alpha)$. We can also define a *problem mapping* function $\mathcal{M}^i_\rho(\rho)$ that maps a problem $\rho = (S_0, S_g)$ at level j to a problem $\mathcal{M}^i_\rho(\rho) = (\mathcal{M}^i_s(S_0), \mathcal{M}^i_s(S_g))$ at level i, where $j < i$. In the remainder of this section, the subscript will be dropped from the mapping functions when it is clear from the context which mapping function is required.

2.3 Refinement of Abstract Plans

A hierarchical planner first finds an abstract plan in the most abstract version of a problem space, and then it refines the plan in successively more detailed problem spaces. The abstract plan is refined at each level by inserting any operators necessary to solve the problem at that level. This section formalizes the refinement of an abstract plan.[3] To define a refinement, this section first defines *establishment* and *justification.*

An operator α *establishes* a precondition of another operator β in a plan, if it is the last operator before β in the plan that achieves that precondition. More precisely, an operator α establishes precondition p of operator β whenever α precedes β, p is an effect of α and a precondition of β, and there are no operators between α and β that have p as an effect. The notation $\alpha \prec_\Pi \beta$ means that operator α precedes operator β in plan Π, and the notation $Ops(\Pi)$ refers to the set of instantiated operators in plan Π.

Definition 1 (Establishment) *Let* Π *be a correct plan,* $\alpha, \beta \in \mathrm{Ops}(\Pi)$, *and* $p \in E_\alpha, P_\beta$. *Then* α *establishes* p *for* β *in* Π *(*$\mathrm{Establishes}(\alpha, \beta, p, \Pi)$*) if and only if*

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

The first condition states that α must precede β in the plan. The second condition states that α must be the last operator preceding β that adds precondition p. Since Π is a correct plan, this implies that there is no operator between α and β that undoes p.

The definition of establishment is now used to define 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 establishes a literal that is either a goal or a precondition of a subsequent justified operator.

Definition 2 (Justification) *Let* Π *be a correct plan,* $\alpha \in \mathrm{Ops}(\Pi)$, *and* S_g *a goal. Operator* α *is justified with respect to* S_g *in* Π *(*$\mathrm{Justified}(\alpha, S_g, \Pi)$*) if and only if there exists* $u \in E_\alpha$ *such that either:*

1. $u \in S_g$, *and* $\forall \alpha' \in \mathrm{Ops}(\Pi)$, *if* $(\alpha \prec_\Pi \alpha')$ *then* $u \notin E_{\alpha'}$, *or*
2. $\exists \beta \in \mathrm{Ops}(\Pi)$ *such that* $\mathrm{Justified}(\beta, S_g, \Pi)$ *and* $\mathrm{Establishes}(\alpha, \beta, u, \Pi)$.

[3]The formalization of refinement as well as the formalization of the ordered monotonicity property presented in the next section is based on joint work with Josh Tenenberg and Qiang Yang [37].

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

Any operator that is not justified is not needed to achieve the goal and can be removed. Thus, an unjustified plan Π (one for which *Justified* is false) that achieves S_g can be justified by removing all unjustified operators. *JustifyPlan*(Π, S_g) is used to denote the justified version of Π. Under the above definitions, for any correct plan Π that achieves goal S_g, *Justified*(*JustifyPlan*$(\Pi, S_g), S_g)$ holds. By the above definitions, *JustifyPlan*$(\mathcal{M}^i(\Pi), \mathcal{M}^i(S_g))$ denotes the abstract plan that corresponds to the ground-level plan Π justified at level i with respect to goal S_g.

With the definition of justification, we can now define plan refinement. A plan Π^{i-1} is a *refinement* of an abstract plan Π^i, if Π^{i-1} solves the given problem, all operators and their ordering relations in Π^i are preserved in Π^{i-1}, and the new operators have been inserted for the purpose of satisfying the preconditions that are introduced at level $i-1$.

Definition 3 (Refinement) *Given a problem ρ and an abstract plan Π^i that solves $\mathcal{M}^i(\rho)$. A plan Π^{i-1} is a* refinement *of Π^i if and only if*

1. *Π^{i-1} solves $\mathcal{M}^{i-1}(\rho)$, and*
2. *there is a 1-1 function c (a* correspondence function*) mapping each operator of Π^i into Π^{i-1}, such that*
   (a) $\forall \alpha \in \mathrm{Ops}(\Pi^i), \mathcal{M}^i(c(\alpha)) = \alpha$,
   (b) *if* $\alpha \prec_{\Pi^i} \beta$, *then* $c(\alpha) \prec_{\Pi^{i-1}} c(\beta)$,
   (c) $\forall \gamma \in \mathrm{Ops}(\Pi^{i-1})$, *if* $\neg\exists \alpha \in \mathrm{Ops}(\Pi^i)$ *such that* $c(\alpha) = \gamma$, *then* $\exists \beta \in \mathrm{Ops}(\Pi^i)$ *such that* $c(\beta)$ *has precondition* p *where* $\mathrm{Justified}(\gamma, p, \Pi^{i-1})$ *and* $\mathrm{Level}(p) = i-1$.

Notice that γ establishes a precondition at level $i-1$, but can have preconditions at a level greater than $i-1$ or can have additional effects that undo conditions that were already established at a level greater than $i-1$. So refining an abstract plan at level $i-1$ can involve establishing literals at level $i-1$ and above.

This formal definition captures the notion of plan refinement used in a number of different planners, including ABSTRIPS [53], ABTWEAK [68], and PABLO [11].

2.4 Ordered Monotonicity Property

Hierarchical planning reduces search by partitioning a problem into a number of smaller subproblems [33]. An effective partitioning of a problem requires that the subproblems can be solved without violating the conditions that were already achieved in the more abstract levels of the abstraction hierarchy. In other words, a hierarchical planner ideally finds a solution at one level and then maintains the structure of that solution while the remaining parts of a solution are filled in. If this is not possible, then there may be little gain from the use of hierarchical planning since solving a subproblem could involve re-solving a large part if not the entire problem.

For example, consider the planning for the design of a house. The problem is naturally decomposed into different abstraction levels where first one might plan the basic layout of the house, then plan the details of the framing, then select the location of fixtures and outlets, and so on. This abstraction is only useful if selecting the location of fixtures and outlets does not require changing the plans for the layout or the details of the framing. If it did, the decomposition of the problem would not be a good one since changing one of these more abstract plans could potentially affect many other parts of the plan.

The example illustrates that a desirable property of an abstraction hierarchy is that it minimize interactions across abstraction levels. A special case of minimizing the interactions is to require that there are no possible interactions across levels of an abstraction hierarchy. Note that this is stronger than simply preventing interactions across levels since it requires that it is an inherent property of a problem space. While this constraint may be more restrictive than necessary, it provides a very effective heuristic for generating useful abstraction hierarchies. This constraint is captured by the *ordered monotonicity property* [31].

The ordered monotonicity property requires that every refinement of an abstract plan leaves the literals that comprise the abstract space unchanged. This property has two important features. First, it is computationally tractable to find abstraction hierarchies with this property from only the definition of the problem space. Section 3 presents the algorithms for automatically generating ordered monotonic abstraction hierarchies. Second, the property captures a large class of abstractions that provide significant reductions in search on a variety of planning domains. Section 5 presents empirical results that demonstrate the effectiveness of these abstractions.

On the other hand, this property is a heuristic and does not guarantee that an ordered monotonic abstraction hierarchy will reduce search. A limitation of the property is that it may still be necessary to backtrack across abstraction levels when using an abstraction hierarchy with this property. The cause for backtracking arises not because of an interaction across abstraction levels, but because in some cases no refinement exists. However, abstraction hierarchies can easily be empirically tested to identify abstractions that require extensive backtracking across levels.

In order to formally define the ordered monotonicity property, we first define an ordered refinement, which is a restriction on the refinement definition presented in the last section. An ordered refinement of an abstract plan Π^i is a refinement Π^{i-1} in which no literals in the abstract level are changed by the operators inserted to refine the abstract plan.

Definition 4 (Ordered Refinement) *Let Π^i be an abstract plan that solves $\mathcal{M}^i(\rho)$ at level i and is justified relative to $\mathcal{M}^i(S_g)$. A level $i-1$ plan Π^{i-1} is an* ordered refinement *of a level i plan Π^i if and only if*

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

The first condition requires that Π^i is a refinement of Π^{i-1}. The second condition above states that in plan Π^{i-1}, the only operators that add or delete literals at level i or above are refinements of the operators in Π^i. An ordered refinement of a level i abstract plan only involves establishing literals at level $i-1$.

Definition 5 (Ordered Monotonic Abstraction Hierarchy) *An abstraction hierarchy is* ordered monotonic *if and only if, for all problems ρ and for all justified plans Π^i that solve $\mathcal{M}^i(\rho)$ at level i, for $i > 0$, every refinement of Π^i 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.

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.

Definition 6 (Problem-Specific Ordered Monotonic Hierarchy)
An abstraction hierarchy is ordered monotonic *relative to a specific problem ρ, if and only if for all justified plans Π^i that solve $\mathcal{M}^i(\rho)$ at level i, for $i > 0$, every refinement of Π^i at level $i-1$ is an ordered refinement.*

2.5 Sufficient Conditions for Ordered Monotonicity

An important feature of the ordered monotonicity property is that ordered monotonic abstractions can be generated from just the initial definition of a problem space. To construct hierarchies of abstraction spaces that have this property, the literals of a problem space are partitioned 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. These constraints require that all of the effects of a given operator are placed in the same level of the hierarchy, and all of the preconditions of an operator are placed at the same or lower level in the hierarchy. This set of constraints is sufficient to guarantee the ordered monotonicity 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 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. These two conditions are sufficient to guarantee the ordered monotonicity property of an abstraction hierarchy.

Theorem 1 *Every abstraction hierarchy satisfying Restriction 1 is an ordered monotonic hierarchy.*

The proof of this theorem is given in Appendix A. The theorem follows from the fact that the restriction guarantees that any justified plan for achieving a given literal will not add or delete a literal in a higher abstraction level.

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. Instead of attempting to find a single abstraction hierarchy that can be used for all problems in a domain, a refinement of this approach is to select each abstraction hierarchy based on a problem or class of problems to be solved. Thus, a set of constraints can be extracted from the operators that guarantee the ordered monotonicity property for a given problem. These constraints require that the effects of each operator that are relevant to the goal of the problem are placed at the same or higher level than the other effects of the same operator, and they are placed at the same or higher level than the preconditions of the operator. This set of constraints is sufficient to guarantee the problem-specific ordered monotonicity property.

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 α relative to a goal S_g (denoted $\mathit{Relevant}(\alpha, S_g)$) are those effects of α that are either in S_g, or are preconditions of operators that have relevant effects with respect to S_g.

Definition 7 (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 α with respect to S_g (or $e \in \mathrm{Relevant}(\alpha, S_g)$) if and only if*

1. *$e \in S_g$, or*
2. *$\exists \beta \in O$, $\mathrm{Relevant}(\beta, S_g) \neq \emptyset$ 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 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 $e \in \mathrm{Relevant}(\alpha, S_g)$ then*

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

The restriction requires that all the relevant effects of an operator α to be at the same or higher levels of abstraction than both the effects that are not relevant and the preconditions of α.

Theorem 2 *Every abstraction hierarchy satisfying Restriction 2 with respect to a problem ρ is a problem-specific ordered monotonic hierarchy with respect to ρ.*

The proof of this theorem is also provided in Appendix A. The idea is analogous to the proof of Theorem 1, where the restriction guarantees that any plan for achieving a literal will not add or delete any conditions in a more abstract problem space.

3 Automatically Generating Abstraction Hierarchies

The previous section presented restrictions on the possible abstraction hierarchies that are sufficient to guarantee the ordered monotonicity property. These restrictions serve as the basis for automatically generating 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. It also 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 4.2 describes the extensions to the algorithms to handle operator schemas.

3.1 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.[4] 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.

3.1.1 Problem-Independent Constraints

A set of problem-independent constraints can be generated for a problem space based on Restriction 1. This restriction requires that all the effects of each operator must be placed

[4] As noted in [56], distinguishing between positive and negative literals would provide slightly finer-grained hierarchies in some cases. This is a straightforward extension, which could be done without any changes to the basic algorithms.

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 1 finds exactly this set of constraints and records them in a directed graph. For each operator, the algorithm arbitrarily 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.

Table 1: Problem-Independent Algorithm for Determining Constraints

---

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);
```

The complexity of this algorithm is $O(l)$, where l 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.[5]

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 find abstractions for a limited class of problem spaces. The next section describes a problem-specific version of this algorithm, which will produce abstractions for a wider class of problem spaces.

[5] Charles Elkan pointed out that my original $O(l^2)$ algorithm [30] could be transformed into a $O(l)$ algorithm.

3.1.2 Problem-Specific Constraints

Restriction 2 can be used to generate a set of problem-specific constraints that are sufficient to guarantee the ordered monotonicity property. This restriction requires that for all of the relevant effects of each operator, those effects must be placed at the same or higher level than the other effects and preconditions of the same operator. An algorithm that implements this restriction is shown in Table 2. The algorithm is similar to the problem-independent one, but forms the constraints based on a particular goal to be solved.

Table 2: Problem-Specific Algorithm for Determining Constraints

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

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

The algorithm is given the operators and the goal of the problem, 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 (line 4). The algorithm then adds constraints between any effect that matches the goal and the other effects and preconditions of the operator (lines 5-10). The algorithm is called recursively on the preconditions of the operator since these could arise as subgoals during problem solving (line 11). The algorithm records the goals that have been considered (line 3) and terminates once it has considered all of the conditions that could arise as goals or subgoals during problem solving.

An important advantage of 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.

3.2 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 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.

- 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 1 or Table 2. By Theorems 1 and 2, 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 [1]. 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.

Table 3: Algorithm for Creating an Abstraction Hierarchy

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.   abs_hierarchy ← Topological_Sort(partial_order);
5.   return(abs_hierarchy)
```

- 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 [2]. 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 4.3.3 describes the set of heuristics used to choose between the possible total orders.

The complexity of steps 2-4 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 $O(max(e, v))$ [1], 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. Thus, the complexity of steps 2-4 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 $O(l)$, where l is the length of the encoding, and the number of possible constraints, e, and the number of possible literals, v, is bounded by $O(l)$, the complexity of the entire algorithm is $O(l)$.

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.

3.3 Applying the Algorithms

This section presents a detailed description of how the algorithms are used to generate abstractions in the Tower of Hanoi puzzle. The Tower of Hanoi is used as an example because it clearly illustrates how the abstractions are constructed. This section describes how the algorithm generates the abstractions for this domain and shows the intermediate results at each step in the algorithm.

Given the three-disk Tower of Hanoi problem shown in Figure 1, 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.

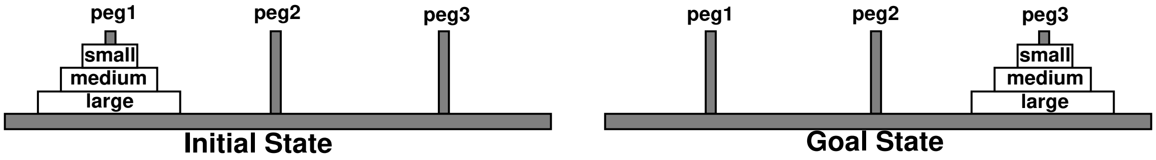

Figure 1: Initial and Goal States for the Tower of Hanoi

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 2. 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 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 for a fully-instantiated operator of the Tower of Hanoi, as shown in Table 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 large peg1)` and `(on large 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 large peg3)` to be higher or at the same level as `(on medium 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.)

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 3 shows the three connected components in the graph, where the literals involving each disk form a component. Each

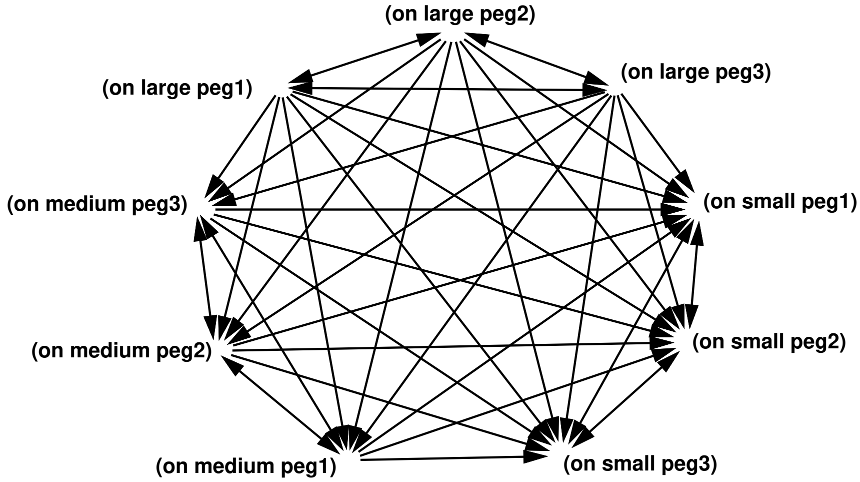

Figure 2: Constraints on the Literals for the Tower of Hanoi

Table 4: Instantiated Operator for the Tower of Hanoi

```
(Move_Large_From_Peg1_to_Peg3
  (preconds (and (on large peg1)
                 (not (on medium peg1))
                 (not (on small peg1))
                 (not (on medium peg3))
                 (not (on small peg3))))
  (effects ((del (on large peg1))
            (add (on large peg3)))))
```

of these components contains a set of literals that must be placed in the same abstraction level.

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, 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.

Using a topological sort, the fourth step in the algorithm converts the partially-ordered directed graph into a total order that represents the final abstraction hierarchy. In the case of the Tower of Hanoi there is only one possible abstraction hierarchy, where the disks are ordered by size. The resulting abstraction hierarchy is shown in Figure 5. For an n-disk problem, the algorithm would produce a n-level abstraction hierarchy.

Using this abstraction hierarchy, a problem solver would first find a plan in the most

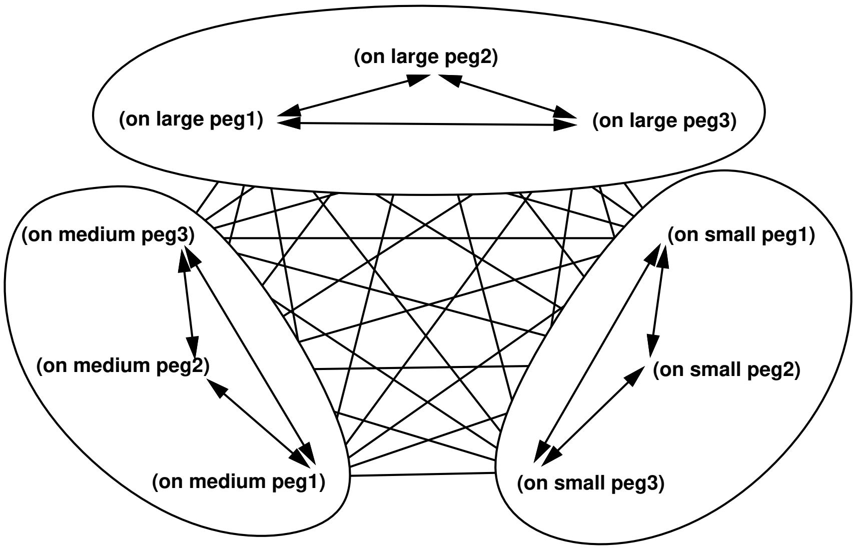

Figure 3: Connected Components for the Tower of Hanoi

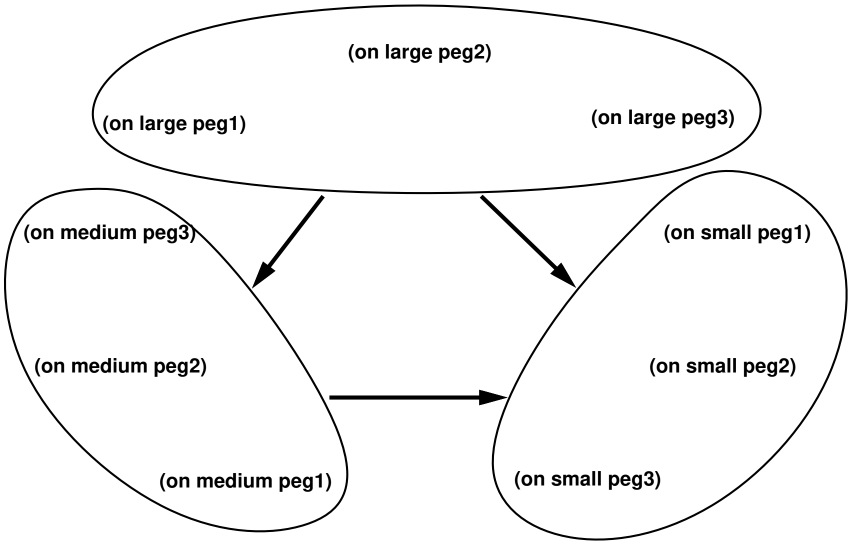

Figure 4: Reduced Graph for the Tower of Hanoi

abstract space for moving 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-size disk would need to be inserted. At the final level, the steps for moving the smallest disk would be inserted to complete the plan. As shown in [33], the use of this particular abstraction hierarchy reduces the size of the search space from exponential to linear in the length of the solution. Holte, Zimmer and MacDonald [27] also showed both analytically and empirically that this decomposition of the problem will produce the shortest solution with the least amount of work.

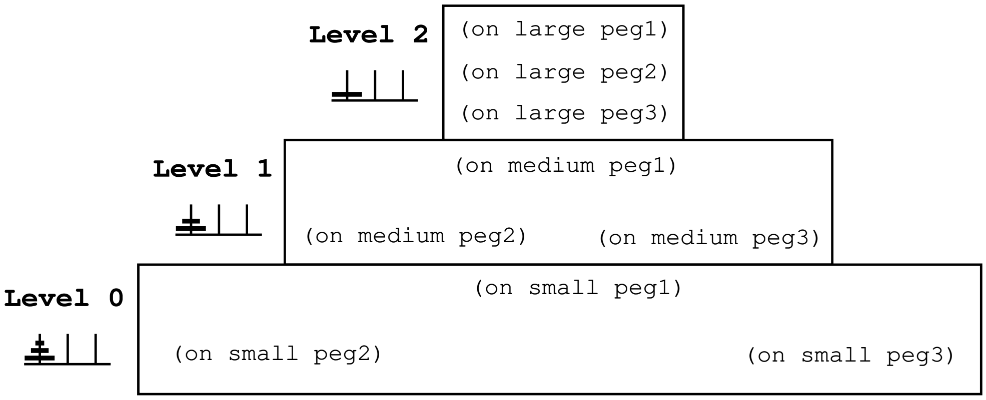

Figure 5: Abstraction Hierarchy for the Tower of Hanoi

4 Generating Abstractions in ALPINE

ALPINE is a fully implemented system that generates abstraction hierarchies for the PRODIGY problem solver [46]. 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 in a hierarchical version of the PRODIGY problem solver [35].

To generate abstraction hierarchies, ALPINE uses an extended version of the problem-specific algorithm described in Section 3. Since the abstractions are to be used by a specific hierarchical problem solver, ALPINE employs several extensions that allow it to produce finer-grained abstraction hierarchies, but still preserve the ordered monotonicity property for the given problem solver. 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 [20], an extended version of the STRIPS domain [44], and a machine-shop scheduling domain [44]. These results are described in Section 5.

To illustrate these extensions, this section uses examples from the extended robot planning domain [44]. This domain is an augmented version of the original STRIPS robot planning domain [20]. 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 three sections. The first section describes the problem-space specification that serves as the input to ALPINE. The second section presents the representation of the abstraction hierarchies that is output by ALPINE. The third section describes the extensions to the basic algorithm that ALPINE uses to generate abstraction hierarchies.

4.1 Problem-Space Specification

The input to ALPINE is 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.[6]

4.1.1 Operators

The first component of a problem space is a set of PRODIGY operators. 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 5 shows an example operator for pushing a box between rooms in the extended robot-planning domain (variables are shown in italics).

Table 5: Example Operator for the Extended 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)))))
```

The effects of an operator are divided into primary and secondary effects, where the primary effects specify the purpose of an operator and the secondary effects are side effects of the operator. A problem solver is only permitted to use an operator to achieve a goal if the desired effect is listed as a primary effect. In the case of the `Push_Box_Thru_Dr` operator in Table 5, the effect (`in-room` *box room.x*) is designated as primary, which means that this operator can only be used to move a box from one room to another. The problem solver would not attempt to use the operator to move the robot to another room. Of course, when a box is moved, the robot would 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 effects match a goal. The information about which effects are primary must be explicitly stated, however, recent work by Fink and Yang [22] describes an algorithm for automatically determining this information.

[6] A problem space in PRODIGY can also include a set of control rules, but they only constrain the search space so ALPINE does not need to consider them to create ordered monotonic abstraction hierarchies.

4.1.2 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 6. The types, shown in boldface, are on the interior nodes of the tree and the instances are on the leaves.

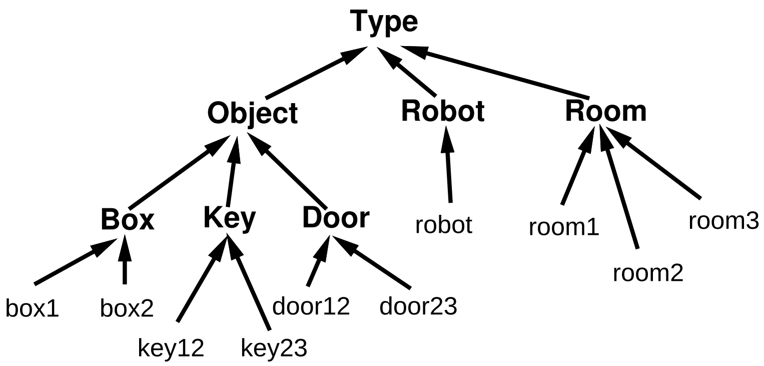

Figure 6: Type Hierarchy for the Extended Robot-Planning Domain

4.1.3 Axioms

The third component of the problem-space specification is a set of axioms that describe invariants of the states of a problem space. The axioms are conditionals with a single antecedent and one or more consequents. All variables in a conditional are universally quantified over the entire expression. A list of axioms for the robot-planning domain is shown in Table 6. The first axiom in the table states that if a door is open then it must be unlocked. These facts cannot be derived from the operators because they describe conditions that hold in every state.

4.2 Representation of Abstraction Spaces

ALPINE generates an ordering of the literals in a domain. 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 a schema corresponds to an operator. A schema can contain both instantiated and uninstantiated literals. Since the algorithms generate abstractions by analyzing potential interactions between the literals used in the operators, the operator representation limits the representation of the abstractions.

Table 6: Example Axioms for the Robot-Planning Domain

```
(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))
```

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 constraint graph. For example, in the robot planning domain, (`inroom robot` *room*) and (`inroom` *box room*) are of distinct types since they differ by the first argument.[7]

4.3 Abstraction Hierarchy Construction

The algorithm described in Section 3 presented a general approach to finding ordered monotonic abstraction hierarchies. ALPINE employs this basic algorithm for constructing abstraction hierarchies, but uses refinements of several steps to produce better hierarchies. The first part describes the extensions to the constraint generation algorithms. The second part describes the automatic reformulation of the problem and domain to exploit the extensions in the constraint generation. The third part presents the algorithms for selecting the final ordered monotonic abstraction hierarchies.

[7]In the current implementation 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, "object" is a type at an interior node in the hierarchy, so it is not possible to have the literal `(inroom object room)` in the final abstraction hierarchy.

4.3.1 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. To avoid these unnecessary constraints, ALPINE employs the algorithm shown in Table 7, which extends the basic algorithm in two ways. First, it uses information about the primary effects of operators to reduce the constraints on the effects. Second, it analyzes the structure of a problem space to determine which preconditions can actually become subgoals, and it uses this information to reduce the constraints on the preconditions. These extensions preserve the ordered monotonicity property for the PRODIGY problem solver and allow the system to form finer-grained hierarchies than would otherwise be possible.

Table 7: Alpine's Algorithm for Determining Constraints

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,goal):
     for each literal in goal do
          if not(Constraints_Determined(graph,literal,goal))
               begin
               Constraints_Determined(graph,literal,goal) ← true;
               for each op in operators do
                    if literal in Primary_Effects(op) do
                         begin
                         for each effect in Effects(op) do
                              Add_Directed_Edge(literal,effect,graph);
                         preconds ← Preconditions(op);
                         subgoals ← Subgoalable_Preconds(preconds,op,literal,goal);
                         for each subgoal in subgoals do
                              Add_Directed_Edge(literal,subgoal,graph);
                         Find_Constraints(graph,operators,preconds)
                         end;
               end;
     return(graph)
```

ALPINE avoids unnecessary constraints generated from the effects by using knowledge about the primary effects of operators. The Find_Constraints algorithm presented in Section 3 considers every operator that has an effect that matches a goal. The algorithm shown in Table 7 extends the algorithm by considering only those operators that have a primary effect that matches a goal. The primary effects specify which operators can be used to achieve a given goal, so this extension eliminates unnecessary constraints by only considering the relevant operators. Since the planning system also uses the primary effects to determine which operators can be used to achieve a given goal, this extension preserves the ordered monotonicity property.

ALPINE also avoids unnecessary constraints by determining which constraints on preconditions are needed to preserve the ordered monotonicity property. If an operator is used to achieve a given goal, it may be necessary to subgoal on any of the preconditions of the operator. To avoid any threats to literals in higher abstraction levels, the basic algorithm adds constraints on each of the preconditions. However, under some conditions the preconditions of an operator will hold and would not be subgoaled on, making the constraints on the precondition unnecessary. Instead of adding constraints on *all* of the preconditions, the extended algorithm only adds constraints on the preconditions that could require subgoaling on a literal that is higher in the abstraction hierarchy. This extension preserves the ordered monotonicity property since the only constraints that are dropped are those that can be shown to be unnecessary.

There are three ways in which the system can show that a given precondition will not require subgoaling on a condition that has been placed higher in the abstraction hierarchy.

1. **The precondition is static.** A static precondition cannot be changed by any operators, so 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. **The precondition occurs in the context of some other operator that also requires the same precondition to hold.** Consider the case where an operator op_a has an effect e that achieves a precondition of an operator op_b, and both op_a and op_b have a precondition p. There are two possible situations can arise if the constraint from e to p is not generated. If p is placed in a higher abstraction level than e, then p would be achieved to satisfy the preconditions of op_b and would already hold when op_a is inserted into the plan. If p is not placed in a higher abstraction level then e, then it would be no different from the situation where the constraint was generated. Thus, it is unnecessary to add the constraint from e to p.

   For example, two preconditions of the `Push_Box_Thru_Dr` operator are that the door is open `(open-dr door)` and the robot is in the room next to the door `(in-room robot room)`, and a precondition of the `Open_Door` operator is also `(in-room robot room)`. If the `(in-room robot room)` precondition is placed higher in the abstraction hierarchy than the `(open-dr door)` precondition, then the system can prove that when the `Open_Door` operator is used to satisfy the `(open-dr door)` precondition of `Push_Box_Thru_Dr`, it will not require achieving the `(in-room robot room)` precondition.

3. **The precondition is the negation of the goal that the operator is used to achieve.** In this case the precondition would not be subgoaled on since the negation must already hold or the operator would not have been selected. The axioms described

in Section 4.1.3 are used to determine whether a precondition is the negation of a goal. 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. If an operator also achieves some other goal, a constraint would be added when the other goal is processed.

The analysis to determine whether a constraint must be added for a precondition in a given context 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` to retrieve the potential subgoals given the preconditions, operator, goal and context. The analysis simply requires checking the three cases described above, but it must be done for each literal in the context of each operator.

ALPINE handles the full PRODIGY language, but does so by possibly overconstraining the final abstraction hierarchy. In the algorithm, disjunctions are treated as conjunctions and conditional effects are treated as unconditional effects. Similarly, universal quantifiers are handled in the same manner as existentials since the type hierarchy will automatically group the instances of the universal. ALPINE essentially transforms PRODIGY's more complex language features into ones that it knows how to handle, so the algorithms, properties, and theorems all apply directly. These particular transformations ensure that there are sufficient constraints on all the preconditions and effects to guarantee ordered monotonicity.

4.3.2 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. This section describes how the original problem space can be reformulated to increase the granularity of the abstraction hierarchies.

ALPINE reformulates a problem space by augmenting both goals and preconditions with additional conditions that necessarily hold. The reformulation is useful because it allows the abstraction mechanism to form abstractions that would not have otherwise been possible. Consider a problem that requires achieving a 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 using the extensions described in the previous section. It does this by proving that if the operator that achieves P has a precondition of Q, then Q will already hold when it attempts to achieve P, so it unnecessary to add a constraint from P to Q. If this constraint was added, then it would not be possible to drop P. 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 considerably easier to achieve it once Q has been achieved.

The algorithm for performing the reformulation is the same for both preconditions and goals. Given a list of goal conditions (or preconditions), each axiom is considered in turn to see if the antecedent of the axiom matches any of the conditions. Before adding the consequents of the axioms to the list of conditions, the algorithm attempts to match each consequent against the conditions. If it finds a match, that consequent is redundant and is not added to the list of conditions. In addition, any variable bindings are recorded and propagated to the other consequents. The algorithm terminates when the axioms have been processed once for each set of goal conditions or preconditions.

ALPINE performs the following reformulation in the robot-planning domain. The goal is to get `boxA` and `boxB` next to each other and to place `boxA` in `room2`:

`(and (next-to boxA boxB)(inroom boxA room2))`.

This problem space has an axiom (shown in Table 6) 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 `boxB` must also be in `room2`:

`(and (next-to boxA boxB)(inroom boxA room2)(inroom boxB 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 place the conditions in the same abstraction level. Augmenting the goal provides the context required to prove that the boxes will already be in the appropriate room to achieve the next-to condition. As a result, ALPINE avoids adding a constraint on the inroom condition. This process was described in the previous section.

The reformulation replaces the original problem of getting the two boxes next to each other with the more abstract problem of getting the two boxes into the same room. Once the problem solver solves the abstract problem, it then refines the plan and adds the additional steps for moving the two boxes next to each other.

ALPINE augments the preconditions of operators in exactly the same manner as goals. For example, the operator `Push_Box_Thru_Dr` would be augmented as shown in Table 8. 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. This reformulation makes the 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. If the operator had not been augmented with these additional conditions, then achieving `(dr-open` *door*`)` could have resulted in a subgoal involving either condition.

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 an augmented problem that can be solved more easily.

Table 8: Reformulated Operator for the Robot Planning Domain

```
(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)))))
```

4.3.3 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 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 an ordered 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 7 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.[8] 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.

5 Empirical Results

This section describes the results of both generating and using abstractions for problem solving. The abstractions are generated by ALPINE and then used in the hierarchical ver-

[8]The idea of separating out the recursive literals was inspired by the work of Etzioni [17], which identified the importance of nonrecursive explanations for explanation-based learning.

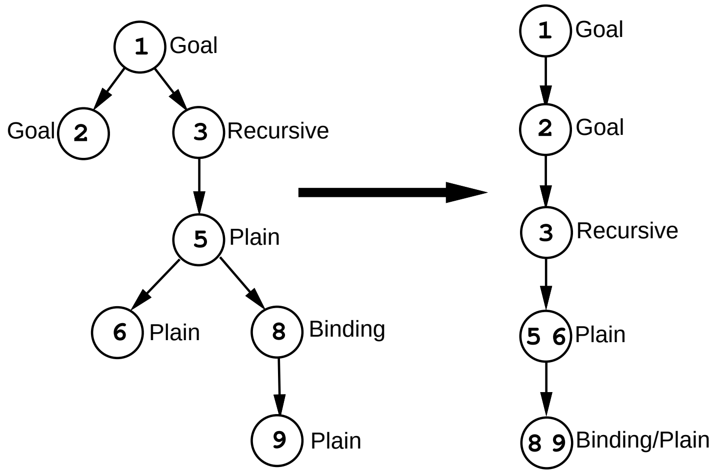

Figure 7: Selecting a Total Order from a Partial Order

sion of PRODIGY. The section is divided into three subsections. The first subsection shows that ALPINE can generate effective abstraction hierarchies for a variety of problem domains. The second subsection compares the performance of ALPINE's abstractions with both hand-coded and automatically generated search control knowledge. The third subsection 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. The raw data from the experiments described in this section is available in [32].

5.1 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 domains, a robot planning domain and a machine-shop planning and scheduling domain, and presents empirical results on the effectiveness of these abstractions at reducing search in PRODIGY. These domains were previously described in [44], where they were used to evaluate the effectiveness of the explanation-based learning (EBL) module in PRODIGY.

5.1.1 Extended STRIPS Domain

This section describes the abstraction hierarchies generated by ALPINE for the extended version of the robot planning domain [20], which includes locks, keys, and a robot that can both push and carry objects. The extensions to this domain make it 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 unavailable keys.

To construct 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 section 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 8. 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. For example, 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 it must also be in `room4`. Second, some of the doors in the initial state are locked and the robot, which starts out in `room5`, will need to go through at least two of the locked doors to solve the problem.

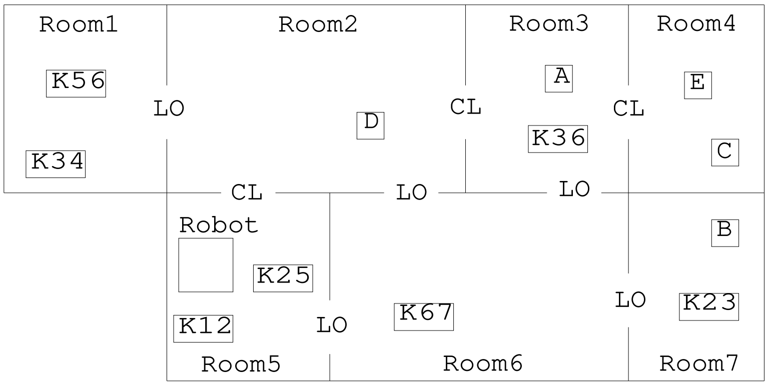

Figure 8: Initial State for the Extended Robot-Planning Problem

To construct an abstraction hierarchy for this problem, ALPINE first augments the goal using the axioms described in Section 4.1.3 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 section. The resulting three-level abstraction hierarchy is shown in Figure 9. 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.

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

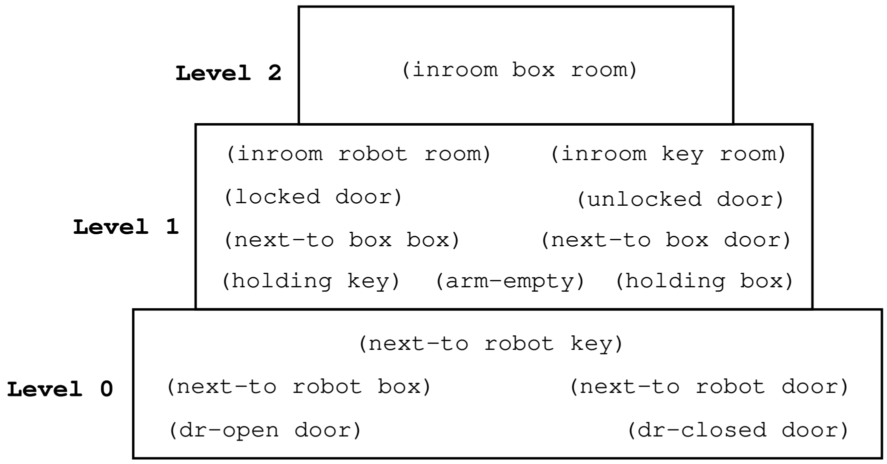

Figure 9: Abstraction Hierarchy for the Extended Robot-Planning Problem

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 such as whether doors are locked or unlocked are considered, it will always be possible to open and close the doors.

The abstractions generated for the example problem produce a significant performance improvement in the hierarchical problem solver. On this example, the abstraction hierarchy reduced the CPU time from 194.6 seconds to 19.2 seconds and reduced the total number of nodes searched from 4069 to 194. In addition, it reduced the solution length from 76 to 45 steps. As described above, the CPU times reported for ALPINE include the time for both generating and using the abstractions.

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 potential problems, the use of abstraction still produces significant performance improvements overall.

To evaluate the abstraction hierarchies produced by ALPINE, this section compares PRODIGY using ALPINE's abstractions (PRODIGY + ALPINE) to problem solving in PRODIGY with no control knowledge (PRODIGY) and to problem solving in PRODIGY with a set of hand-coded control rules (PRODIGY + HCR). These hand-coded control rules correspond to the ones that were used in the EBL experiments. The comparison was made on a set of 250 randomly generated problems, where the different configurations were each allowed to work on a problem until it was solved or the 600 CPU second time limit was exceeded. Of these problems, 100 were used in Minton's experiments [44] to test the EBL module. 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.

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 [45], but Segre et al. [55] 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 of 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 10 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 + ALPINE can solve all the solvable problems in less than 200 CPU seconds. In contrast, both PRODIGY and PRODIGY + HCR cannot solve some of the problems within 600 CPU seconds. In addition, the total time spent by PRODIGY is over three times that of PRODIGY + ALPINE. On the unsolvable problems the difference between the use of abstraction and no abstraction is less dramatic, although PRODIGY + ALPINE has solved more of the problems in considerably less time than PRODIGY.

To evaluate the statistical significance of the results in the experiment, we can apply the signed-rank test as presented by Etzioni and Etzioni [18]. This test generates an upper bound on what is called the p-value. The p-value is the probability that conclusions drawn from the data are in error. The lower the p-value, the stronger the evidence that the hypotheses are correct. In all of these comparisons the significance level is taken to be 0.05. When the p-value is below the significance level the results are considered to be statistically significant.

Applying the signed-rank test to all of the problems in the experiment indicates that there is insufficient evidence to conclude that one system is statistically better than another

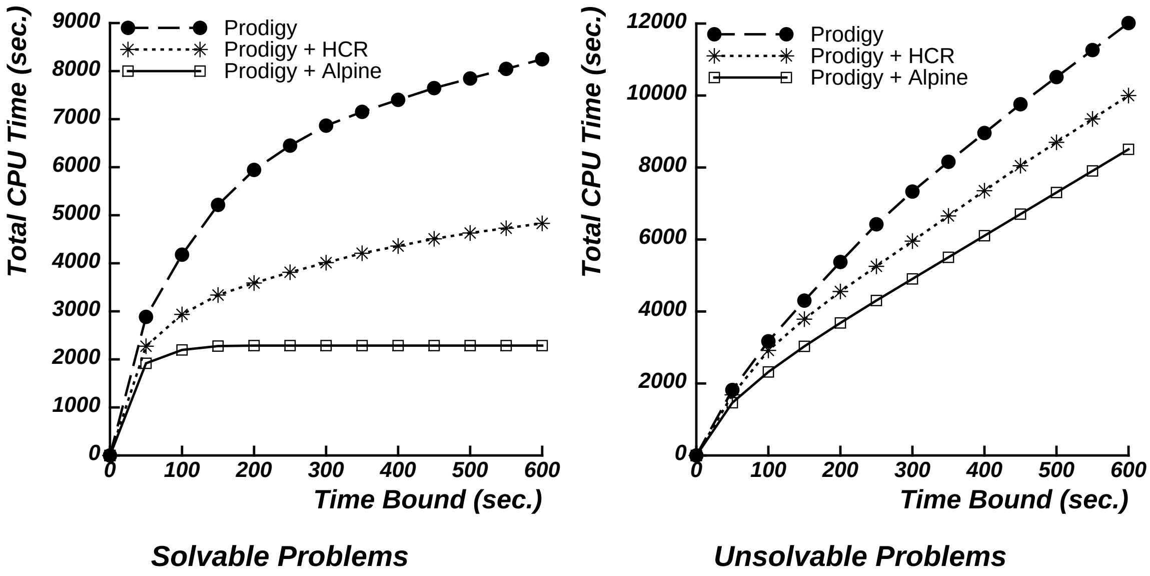

Figure 10: Total CPU Times in the Robot Planning Domain

on either the solvable or unsolvable sets of problems. This can be explained by the fact that PRODIGY performs better on the smaller problems due to the overhead of generating and using the abstractions, while PRODIGY + ALPINE performs better on the larger problems. This is illustrated in Figure 11, which graphs the solution time against the average problem size for problems grouped by size. On the larger problems, those with a solution length of 40 or greater, there is sufficient evidence to conclude that the difference between PRODIGY + ALPINE and PRODIGY is significant with a p-value of 0.000. However, the difference between PRODIGY + ALPINE and PRODIGY + HCR on the larger problems is not significant since the p-value is 0.224.

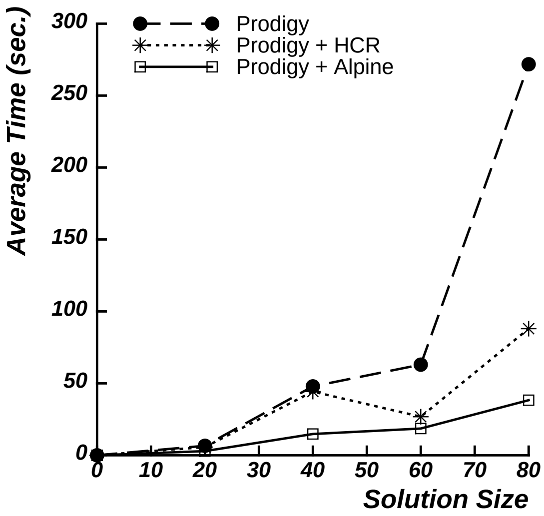

Figure 11: Average Solution Times in the Extended Robot-Planning Domain

Another important property not shown in the graphs is that PRODIGY + ALPINE produces shorter solution than both PRODIGY and PRODIGY + HCR. Using the signed rank test and assuming the standard significance level of 0.05, the difference between PRODIGY + ALPINE and both PRODIGY and PRODIGY + HCR is statistically significant with p-values of 0.003 and 0.000, respectively.

5.1.2 Machine-Shop Scheduling Domain

This section describes the abstractions generated by ALPINE in a machine-shop process planning and scheduling domain. This 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.

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 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 section 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, `d` and `e`:

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

The problem requires making a hole in part `d`, making it cylindrical, painting it white, and also making part `e` smooth. The resulting abstraction hierarchy for this problem is shown in Figure 12. The hierarchy separates the selection and ordering of the various operations and performs the scheduling last. This abstraction produces a considerable improvement in problem-solving performance. The total search time is reduced from 164.7 seconds to 7.0 seconds and the number of nodes searched is reduced from 5150 to 39. The solution length remained the same.

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 PRODIGY + ALPINE to PRODIGY with no control knowledge and PRODIGY with a set of hand-coded control rules

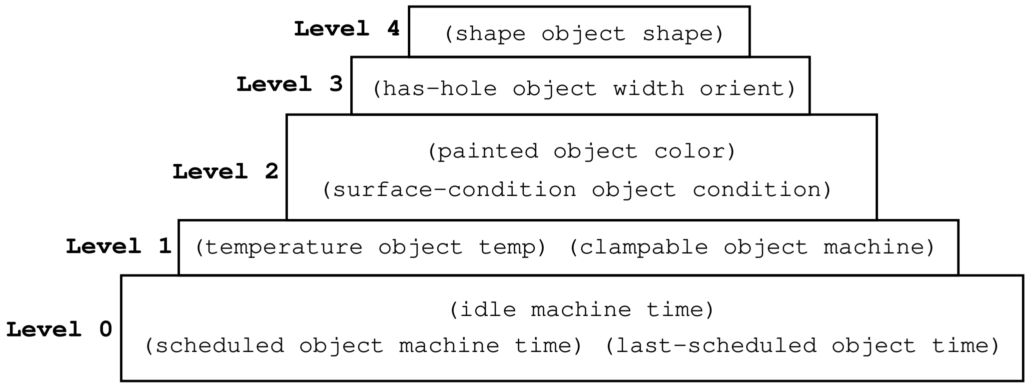

Figure 12: Abstraction Hierarchy for the Machine-Shop Problem

(PRODIGY + HCR). The hand-coded rules are the same rules that were used in the original comparisons with the EBL system [44]. All the configurations were run on 250 randomly generated problems including the 100 problems used for testing the EBL system.

The comparison, shown in Figure 13, graphs the total time against an increasing time bound for solvable and unsolvable problems. On the 186 solvable problems, PRODIGY + ALPINE performs better than both PRODIGY and PRODIGY + HCR. On the 64 unsolvable problems, PRODIGY + ALPINE performs better than PRODIGY. With control knowledge PRODIGY + HCR can quickly show for most of the problems that the problems have no solution. After 600 CPU seconds PRODIGY + ALPINE and PRODIGY + HCR have used the same total time, but the slopes of the lines at 600 seconds show that PRODIGY + ALPINE has completed more of the problems. This can be explained by the fact that the problem solver can often use control knowledge to 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 + HCR would have to search the entire space. Thus, using control knowledge the problem solver can quickly determine that a problem is unsolvable, but the use of abstraction produces better coverage.

As in the previous section, if we apply the signed rank test to the entire set of problems, there is insufficient evidence to conclude that ALPINE is better than the other two systems on the solvable problems. However, Figure 14 shows that the important difference between PRODIGY + ALPINE and the other two systems is on the harder sets of problems. If we apply the signed rank test to the set of problems with an average solution length of 10 or greater, then the difference between PRODIGY + ALPINE and both PRODIGY and PRODIGY + HCR is significant with p-values of 0.006 and 0.027. The difference between PRODIGY + ALPINE and PRODIGY on the entire set of unsolvable problems is also significant with a p-value of 0.000. As in the extended-strips domain, PRODIGY + ALPINE produces shorter solutions than both PRODIGY and PRODIGY + HCR, and the differences are significant with p-values of 0.000 and 0.040, respectively.

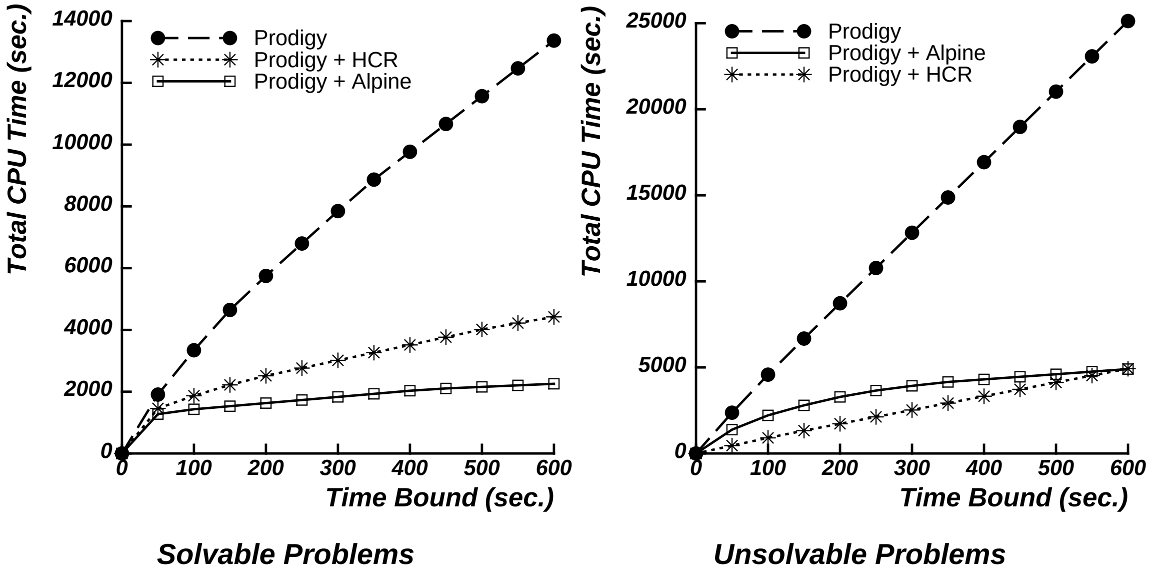

Figure 13: Total CPU Times in the Machine-Shop Domain

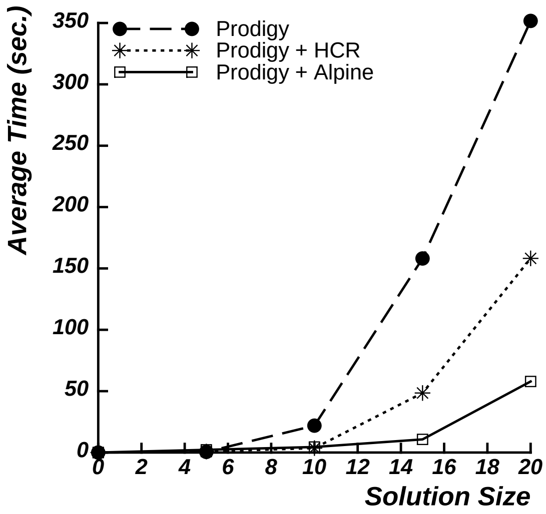

Figure 14: Average Solution Times in the Machine-Shop Domain

5.2 Comparison of ALPINE and EBL

A significant amount of work in PRODIGY has focused on learning search control to reduce search. Minton [44] developed a system called PRODIGY/EBL that learns search control rules using explanation-based learning. More recently, Etzioni [16] 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 described in the last section. In addition to PRODIGY alone, with the hand-coded control rules (PRODIGY + HCR), and with ALPINE's abstractions (PRODIGY + ALPINE), the graphs also include PRODIGY with the control rules produced by EBL (PRODIGY + EBL), with the control rules produced by STATIC (PRODIGY + STATIC), and the combination of ALPINE's abstractions and the hand-coded control rules (PRODIGY + ALPINE + HCR).

The comparison shown in Figure 15 graphs the total time against an increasing time bound for the solvable and unsolvable problems. On the solvable problems, the difference between PRODIGY + ALPINE and PRODIGY + STATIC or PRODIGY + EBL is not statistically significant. However, on the set of larger problems (those with an solution length greater than 10), the difference between PRODIGY + ALPINE and PRODIGY + EBL is significant with a p-value of 0.000. On the unsolvable problems, PRODIGY + STATIC and PRODIGY + EBL perform the same as PRODIGY + HCR and use about the same total amount of time on the unsolvable problems, but PRODIGY + ALPINE completes more of the problems after 600 CPU seconds than the other configurations.

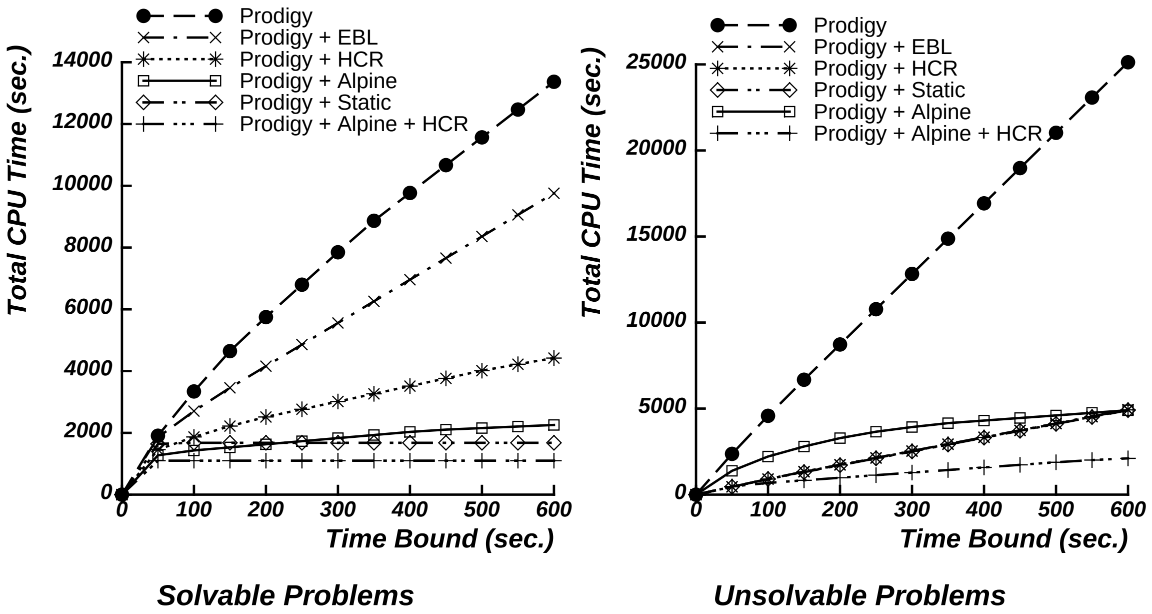

Figure 15: Total CPU Times for the Learning Systems in the Machine-Shop Domain

The use of abstraction and search-control knowledge can be combined since they provide complementary sources of knowledge. The figures above 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 15, produces significantly better performance than any system alone on the larger solvable problems. The differences between PRODIGY + ALPINE + HCR and the other systems not using abstraction are statistically significant with a p-value of 0.003 for PRODIGY and p-values of 0.000 for the rest. 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. In [36] we show that integrating abstraction and the rules learned from EBL produce similar results.

5.3 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 automated the construction of abstraction hierarchies for problem solving. The resulting abstraction hierarchies were then used for problem solving in an extended version of the STRIPS planner [20]. This section compares the abstraction hierarchies generated by ABSTRIPS and ALPINE in the STRIPS domain. In order to evaluate the effectiveness of the different abstraction hierarchies, the abstractions generated by each system are 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. The technique used to construct the abstraction hierarchy is described in Section 6. 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 generation of abstraction hierarchies in 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, whereas 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 16. 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.

The abstraction hierarchies generated by each system are shown in Figure 17. 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.

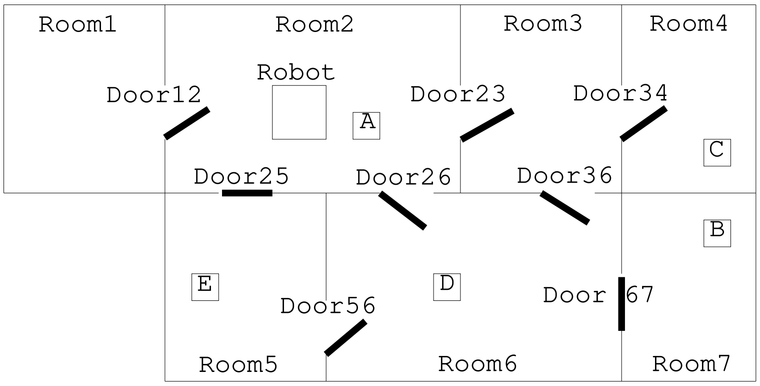

Figure 16: Initial State for the Example STRIPS Problem

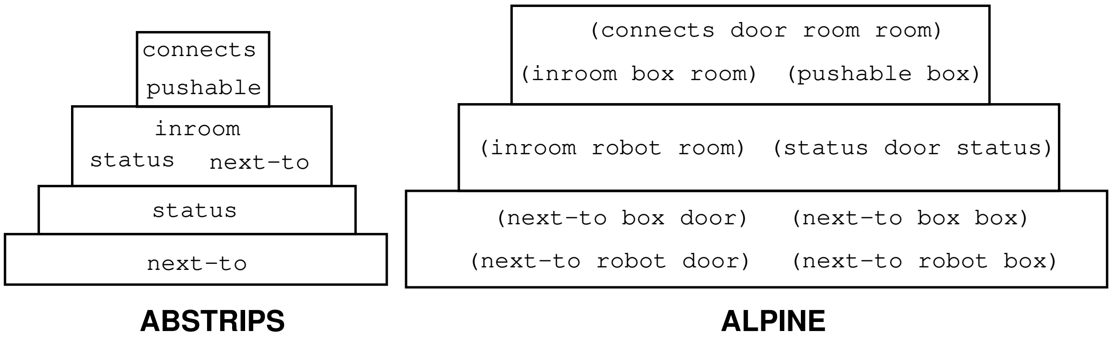

Figure 17: Abstraction Hierarchies Generated by ABSTRIPS and ALPINE

ALPINE can build finer-grained hierarchies using the type hierarchy (Section 4.1.2) 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 4.3.3 combine the bottom two levels of an initial four-level hierarchy 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 limitation of the abstraction hierarchies that are formed by ABSTRIPS. Since ABSTRIPS only drops preconditions and does not drop conditions from the states or goals, all of the goal conjuncts must be considered in the abstract space. As such, 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 ordered monotonicity property). At this point the problem solver would need to either backtrack or insert additional steps for closing the door again.

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 `next-to` preconditions.

To illustrate the difference between ALPINE's and ABSTRIPS's abstractions, the use of these abstractions are compared in PRODIGY. This is not a completely fair comparison since 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 probably 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 9 shows the CPU time, nodes searched and solution length. PRODIGY + ALPINE produces a small performance improvement over PRODIGY and generates shorter solutions. In contrast PRODIGY + ABSTRIPS takes almost 6 times longer than PRODIGY, although it too produces the same length solution as PRODIGY + ALPINE.

Table 9: Performance Comparison for the Example STRIPS Problem

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

The graphs in Figure 18 compare the average solution times and average solution lengths of PRODIGY without using abstraction, PRODIGY using the abstractions produced by ABSTRIPS (PRODIGY + ABSTRIPS), and PRODIGY using the abstractions produced by ALPINE (PRODIGY + ALPINE). (This section presents average time instead of total time since almost all of the problems were solvable.) Each configuration was run on 200 randomly generated problems in the STRIPS 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 solvable in principle. The solution time graph in Figure 18 shows the average solution times for the 197 solvable problems. The solution length graph shows the average solution lengths on the 153 problems that were solved by all three configurations. In both

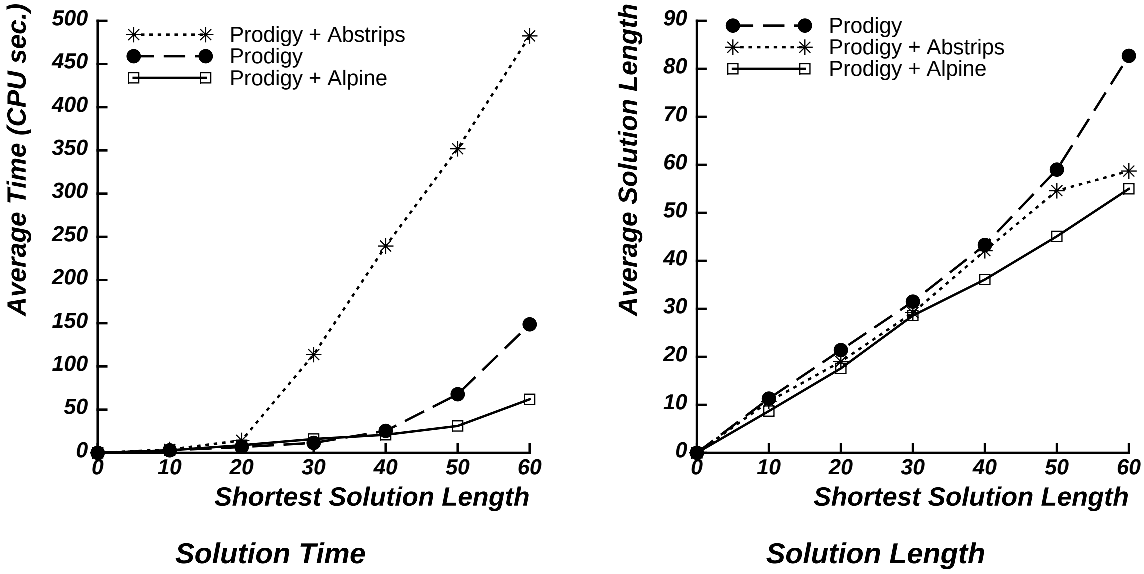

Figure 18: Comparison of the Average Solution Times and Average Solution Lengths

graphs the problems are ordered by the shortest solution found by any of the configurations.

The graphs show that the use of ABSTRIPS' abstractions significantly degrades performance, while ALPINE's abstractions improve performance over the basic PRODIGY system. The difference between PRODIGY + ABSTRIPS and both PRODIGY + ALPINE and PRODIGY is significant with p-values of 0.000 for both systems. The reason for the poor performance using ABSTRIPS' abstractions is that in the analysis ABSTRIPS performs to assign criticalities it assumes that the preconditions are independent. When this assumption fails to hold, the abstraction generated by ABSTRIPS may be inappropriate for the given problem.

The differences between the solution times for PRODIGY + ALPINE and PRODIGY are not significant. This is because PRODIGY only needs to search a small portion of the search space since most mistakes can be undone by adding additional steps. Thus, on problem-solving time PRODIGY performed quite well, but it achieved this performance by trading solution quality. On the hardest set of problems, PRODIGY produces solutions that were on average fifty percent longer than PRODIGY + ALPINE. In contrast, the use of ABSTRIPS' abstractions significantly increased the problem solving time, but they did improve the quality of the solutions. Both PRODIGY + ALPINE and PRODIGY + ABSTRIPS produces shorter solutions than PRODIGY and the differences are significant with p-values of 0.000. In addition, the difference in solution length between PRODIGY + ALPINE and PRODIGY + ABSTRIPS is significant with a p-value of 0.008.

6 Related Work

This section describes the related work on abstraction in problem solving. The first subsection describes how abstractions are used in various systems. The second and third subsections compare techniques for generating abstractions and other types of related control knowledge. And the fourth subsection contrasts the various properties related to abstraction.

6.1 Using Abstraction

One approach to using abstractions is to employ a hierarchy of abstraction spaces, where a problem is solved in the most abstract space and then refined at successively more detailed levels. Each abstraction space is a "simplification" of the original problem space, such as the relaxed and reduced models described in Section 2.2. This general approach has been referred to as *state abstraction* and was first used in Planning GPS [48]. This is also the approach used in ABSTRIPS [53] and PABLO [11], ABTWEAK [68, 67], as well as ALPINE. ABSOLVER [47] also employs a form of state abstraction, but instead of refining abstract plans found using this simplified model, the abstract plans are used in the evaluation function of an admissible search procedure.

Another commonly used 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* [66], which specify the relationship between an abstract operator and the refinements of that operator. This approach differs from state abstraction in that there is no abstraction of the state, but only of the operators. As such, this approach is sometimes referred to as *operator abstraction*. 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 during the refinement of a plan. Operator abstractions have been used extensively in least-commitment problem solvers such as NOAH [54], MOLGEN [58], NONLIN [59], and SIPE [64].

The difference between operator abstraction and state abstraction is small since operator abstraction can be used to implement state abstraction by imposing constraints on the order in which the operator abstractions are expanded. This is the approach taken in SIPE [64, 65], where the domain is partitioned into literals at different abstraction levels and operators for achieving those literals.

Another problem-solving method, similar to the use of state abstractions, is the use of macro problem spaces. Instead of forming abstract problem spaces by constructing relaxed or reduced models of a problem space, operators are combined into macro operators to form a macro problem space [39]. This approach is similar to using state abstractions 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. A related idea is to construct macro objects instead of operators and then reason about the macro objects [8].

Other systems have also used macro operators, but instead of constructing a new macro space the macro operators are simply added to the original space [19, 26, 28, 38, 41, 52]. While this approach may reduce the depth of the search by providing sequences of operators that can solve entire problems, it has the problem that it can significantly increase the branching factor since the problem solver must consider the original operators as well as the new macros [43].

6.2 Generating Abstractions

ALPINE forms abstractions based on the ordered monotonicity property. This property guarantees that any plan for achieving a literal ignored at an abstract level will not add or delete a literal in a more abstract space. In effect, the ordered monotonicity property partitions those literals that interact with one another and orders the partitioned sets of literals in a way that minimizes the interactions among them. An important feature of the ordered monotonicity property is that ALPINE can tractably generate problem-specific abstractions that have this property.

ABSTRIPS [53] was the first system to automate the formation of abstraction hierarchies for hierarchical planning. The system only partially automates this process since the user must provide an initial partial order of predicates, which is used to assign criticalities to the preconditions of the 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.

ABSTRIPS 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. This criterion is quite different than the one used by ALPINE since it attempts to guarantee that the conditions ignored in the abstract space can be achieved in a few steps. However, it does not actually guarantee this property since the algorithm assumes that the different goal conditions will not interact with one another. (See [34] for a detailed discussion of this point.) Another limitation of this approach is that the conditions that cannot be achieved by a short plan are placed in the same abstraction level. This limits the usefulness of the abstraction hierarchies since the bulk of the work would occur in the level with conditions that cannot be achieved by a short plan.

PABLO [11] is another system that generates abstractions for hierarchical planning. It uses a technique called *predicate relaxation* to determine the number of steps needed to achieve each predicate by partially evaluating the operators. This information is then used to focus the problem solver on the conditions that requires the greatest number of steps. The approach is quite similar to the one used by ABSTRIPS in that the abstractions are based on how many steps (in the worst case) it will take to achieve a given precondition instead of whether or not a condition can be achieved in a few steps. While this approach allows an arbitrary number of abstraction spaces, PABLO also assumes that the preconditions will not interact, so it may believe that a subgoal is achievable when it is not, and it may underestimate the number of steps required to achieve a subgoal. Another limitation is that the predicate relaxation process may be very expensive and result in complex expressions that must be evaluated at planning time.

Anderson [3] 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. This approach is different than the previous ones since PLANEREUS forms abstract operators by ignoring the differences between operators without regard to the difficulty of achieving those differences. In contrast, ABSTRIPS and PABLO consider the number of steps required to achieved the conditions and ALPINE considers the potential interactions between the conditions being ignored and those remaining.

6.3 Generating Control Knowledge

The use of abstraction in problem solving is a form of control knowledge. An alternative to explicitly constructing the abstractions is to represent analogous control knowledge in a different form. There are many systems that use control knowledge, but this section only describes the most closely related ones. All of these systems use techniques related to abstraction to learn information to guide the problem solver at various control choices.

Unruh and Rosenbloom [62, 63] developed a weak method for SOAR [40] that dynamically forms control knowledge by dropping preconditions of operators. When SOAR is working on a goal and reaches an impasse, a point in the search where it does not know how to proceed, it performs a look-ahead search to resolve this impasse. Since this search can be expensive, the abstraction mechanism performs a look-ahead search that ignores all of the unmatched preconditions that are encountered during the search. The choices made in the look-ahead search are then stored by SOAR's chunking mechanism and the chunks are used to guide the search in the original space. This is essentially a generate and test method for using abstractions to find control knowledge. The approach differs from the one used by ALPINE in that there is no analysis of the problem space.

GPS [48] 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 and is closely related to the techniques for generating abstractions in ALPINE. The criterion for ordering the differences in [12, 15] 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. The algorithm for finding an ordering requires building a table of differences and finding a lower-triangular difference table. This is similar to the analysis performed by ALPINE, except the ordering of differences 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.

In a more recent system for generating difference orderings, Goldstein [14, 25] incorporated an additional restriction that also takes the preconditions into account and is thus analogous to the ordered monotonicity property. The system produces difference orderings by creating a difference table for the top-level goals and each set of precondition of the operators. Using this difference table, it then tries to find an ordering of all the goals such that achieving one goal in the ordering will not interact with goals earlier in the ordering. The algorithm for generating the difference ordering requires searching through the space of all possible differences until one is found that does not depend on any other differences. This difference is then placed on the bottom of the order and the process is repeated until all of the differences are ordered. This algorithm is less efficient than the one used by ALPINE and does not provide a mechanism for grouping together differences if an ordering does not exist that satisfies the property.

Irani and Cheng [10, 29] present an approach to ordering goals based on interactions determined statically from the operator definitions. For each problem the goal orderings are determined by backpropagating 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 4.3.2), but the approach to ordering the goals is much more similar to the analysis in PABLO [11].

Etzioni [16] 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.

Smith and Peot [57] developed an approach to analyzing potential conflicts in order to delay resolving conflicts for partial-order planning. The analysis used to determine which conflicts can be delayed is similar to the analysis performed by ALPINE. In their analysis graph, they distinguish between conflict links (generated from the effects of operators) and causal links (generated from the preconditions of operators). From this graph they can then determine whether resolving a potential conflict can be safely delayed.

6.4 Properties of Abstractions

When abstractions are used for hierarchical problem solving, the potential difficulties are in refining an abstract plan into a plan in the original problem space. The various properties that have been identified are all related to this general problem in one way or another. There is also a closely related issue in ordering goals. This section describes the various properties for both abstraction and goal ordering.

The *downward solution property*, identified by Tenenberg [61] states that the existence of an abstract-level solution implies the existence of a ground-level solution. Ideally an abstraction space would have the downward solution property since once an abstract solution is found it is just a matter of refining it into a ground-level solution. However, if an abstraction space is formed by dropping conditions from the original 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. Thus, using an abstraction space formed by dropping information it is difficult to guarantee the downward solution property. The same problem arises in the use of abstraction in theorem proving, where it is called the false proof problem [23, 51].

Bacchus and Yang [4, 5] identified a closely related property called the *downward refinement property* (DRP), which states that if a problem is solvable then any abstract solution must have a refinement. Thus, if a solution to a problem exists, then the conditions ignored at the abstract level will be achievable. They also developed a set of sufficient conditions that can be used to identify abstractions that have this property. The property can only be guaranteed in restricted cases, although they have developed some techniques to find "near-DRP" abstractions.

The *ordered monotonicity* property is orthogonal to both the downward solution and downward refinement properties. The ordered monotonicity property, as described in Section 2, imposes the additional restriction that all refinements of those plans leave the literals established in the abstract plans unchanged. This property does not guarantee that any abstract solution can be refined. What it does guarantee is that some abstract solution can be refined and more importantly, that it can be refined in a particular manner. The property captures the idea that an abstract solution should solve some aspect of the problem, which can then be held invariant while the remaining unsolved aspects of the problem are successively elaborated. As noted by Smith and Peot [56], this property addresses the problem of operator interference, but does not deal with the problem that the planner may select a set of bindings that prevent a solution from being refined and force the system to backtrack across abstraction levels. More recently, Bacchus and Yang [5] developed a system called HIGHPOINT that addresses this problem by combining their "near-DRP" property with ALPINE to generate abstractions.

Fink [21] recently identified a number of refinements to the definition of justification that eliminate unnecessary operators in plans to various degrees. He uses these refinements to restrict the definition of ordered monotonicity. These restricted definitions avoid certain pathological cases such as plans that achieve, undo and then reachieve the same condition. However, it is unclear whether these refinements will generate improved hierarchies over those produced by ALPINE.

In work on ordering goals for problem solving there are a set of closely related properties to those described above. In Korf's work on generating macro operators [38], 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. This analogue of this property for an abstraction space is a property that guarantees both the downward solution and ordered monotonicity property. This is because it guarantees both that the problem will be solvable and that once a goal is satisfied it never needs to be violated to satisfy the remaining goals.

Banerji and Ernst [6, 7] developed a formal model of difference ordering in GPS [13], which requires that any goal condition that is already achieved cannot be reintroduced. This means that after a given difference is solved, the problem solver is prevented from reintroducing that difference. This restriction on difference ordering serves as the foundation for the work by Goldstein described in Section 6.3.

7 Limitations and Future Work

While the techniques described in this article are effective in generating useful abstractions for a variety of problem solving domains, they are not without their limitations. This section describes some of the limitations of both the theory and approach for generating abstractions and presents some ideas about how to produce better abstraction hierarchies automatically

7.1 Ordered Monotonicity Property

The ordered monotonicity property does not guarantee that an abstraction hierarchy will be useful. 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 into 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 property on which the abstractions is based does not take into account the difficulty of achieving the conditions that are ignored. It only considers whether the achievement of the conditions can be delayed without interfering with those parts of the problem that have already been solved.

A direction for future work would be to consider not only whether the ordered monotonicity property can be ensured, but also the difficulty of achieving those conditions that are ignored. This could be dealt with in several ways. The system could attempt to prove that the conditions are achievable. The problem with this approach is that either the proofs must be done assuming the conditions are independent, as done in ABSTRIPS, or in general the proofs will be as hard as the original planning problem. Another problem is that requiring that the ignored conditions are always achievable is not necessary for producing useful abstractions. The empirical results indicate that even on problems that require backtracking across levels, the use of the abstraction may still reduce search overall. A more attractive approach is to maintain statistics on the costs and benefits of each abstraction and eliminate those abstraction whose cost outweigh their benefit.

7.2 Generating Abstraction Hierarchies

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 parameterized literal for either a precondition or effect, then whether or not ALPINE can place two instances of this literal at different levels in the hierarchy depends on whether the two literals are distinguishable in the type hierarchy. This is because the algorithm determines the interactions between literals based on the typed preconditions and typed 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 4, 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, a representation consisting of one operator for moving each disk would constrain the literals for each different size disk to be in the same abstraction level. For example, `(on large peg1)`, `(on large peg2)`, and `(on large peg3)` would be 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 large` *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 10. 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 described earlier would not find any abstractions given this representation of the problem.

To avoid this problem, an extended version of ALPINE was built that does find the abstraction of the Tower of Hanoi described earlier from the single-operator representation of the problem. The extended system partially evaluates the operators and determines 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. Once the potential interactions are determined for each literal in the domain, the basic algorithm for generating abstractions is used to construct the abstraction hierarchy.

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.

Table 10: Single-Operator Version of the Tower of Hanoi

```
(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)))))
```

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 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. A good direction for future work would be to find ways to selectively instantiate literals. 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 analyzing the operator schemas, 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" [49] 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 on the next level, and A 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.

A promising direction for future work is to use explanation-based learning to acquire a set of necessary conditions to guarantee the ordered monotonicity property. The system would begin with no constraints on the abstraction hierarchy and add constraints on the possible hierarchies whenever the ordered monotonicity property is violated. When a violation is detected, which occurs any time an operator is applied at one level and changes a condition in a more abstract level, 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.

8 Conclusion

This paper presented an approach to automatically generating abstraction hierarchies. This approach takes a problem and 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. This section summarizes the contributions of this work.

First, this article identified the *ordered monotonicity* property, which is based on the idea that the structure of an abstract plan should not be changed in the process of refining the plan. This property provides an effective criterion for generating useful abstraction hierarchies. It requires that the literals in an abstraction hierarchy are ordered such that achieving literals at one level will not interact with a solution at a more abstract level.

Second, this article provided a completely automated approach to generating abstraction hierarchies based on this property. The algorithm presented in this article 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. These abstraction spaces are then used for hierarchical problem solving.

Third, this article described an implementation of these ideas and demonstrated empirically the effectiveness of the resulting abstractions. The abstractions are generated by a system called ALPINE and then used in a hierarchical version of the PRODIGY problem solver. The article presented results on both generating and using abstractions on large sets of problems in multiple problem spaces. 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 [44] and STATIC [16]. The results show that the abstractions provide considerable reductions in search and improvements in solution quality over the basic PRODIGY system and provide comparable results to the EBL methods for learning control knowledge.

Acknowledgments

There are many people that contributed in one way or another to this work. Jaime Carbonell, my thesis advisor, provided invaluable guidance on this work. The other members of me thesis committee, Tom Mitchell, Paul Rosenbloom, and Herb Simon, all contributed to both the ideas and presentation of this work. Steve Minton, Oren Etzioni, Manuela Veloso, Yolanda Gil, Qiang Yang, Josh Tenenberg, and Claire Bono all provided a great deal of inspiration and support, as well as a lot of good ideas. Qiang and Josh also worked on the initial formalization of the ordered monotonicity property with me. Oren gave me the code to run the statistical significance test. Finally, Oren, Yolanda, and the anonymous reviewers all provided very helpful comments on earlier drafts of this article. I am grateful to all of these people.

The research reported here was supported in part by an Air Force Laboratory Graduate Fellowship through the Human Resources Laboratory at Brooks AFB, in part 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, and in part by Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency under contract no. F30602-91-C-0081. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, HRL, AL, RL, the U.S. Government, or any person or agency connected with them.

A Proofs

Lemma 1 *If an abstraction hierarchy satisfies Restriction 1, then any justified plan for achieving a literal l does not add or delete any literal whose level is higher than* Level(l).

Proof: Let Π^i be a justified plan at level i that achieves l. Since Π^i is justified, every operator in Π^i is used either directly or indirectly to achieve l. Thus, the establishment relations in Π^i 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 $\forall \alpha \in \mathit{Ops}(\Pi^i), e \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e)$. This condition will guarantee that no operator in Π^i affects any literal higher than $\mathit{Level}(l)$ in the hierarchy.

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

For the inductive case, assume that for each operator β at depth i, $\forall e \in E_\beta$, $\mathit{Level}(l) \geq \mathit{Level}(e)$. Let α be an operator at depth $i+1$. Since Π^i is justified, there exists an operator β at depth i with $p \in P_\beta$, such that $p \in E_\alpha$. From Restriction 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 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 1 *Every abstraction hierarchy satisfying Restriction 1 is an ordered monotonic hierarchy.*

Proof: From Definition 5 we need to show that every refinement of a justified plan Π^i is an ordered refinement. By way of contradiction, assume that there exists a plan Π^{i-1} that is a refinement of Π^i at level $i-1$, but is not an ordered refinement. It follows from Definition 4 that an operator α in Π^{i-1} changes a literal l, with $\mathit{Level}(l) \geq i$, but the corresponding abstract operator $\mathcal{M}^i(\alpha)$ is not in Π^i. Since Π^{i-1} is a refinement, it follows from Definition 3 that Π^{i-1} is justified. Since Π^{i-1} is justified and $\alpha \in \mathit{Ops}(\Pi^{i-1})$, α must achieve some condition p and be justified with respect to that condition. In addition, since $\mathcal{M}^i(\alpha)$ is not in Π^i, it follows from Definition 3 that $\mathit{Level}(p) = i-1$. But α also achieves l, where $\mathit{Level}(l) \geq i$, which contradicts lemma 1. □

Lemma 2 *If an abstraction hierarchy satisfies Restriction 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 1. As above, let Π^{i-1} be a justified plan at level i that achieves l, where the establishment relations in Π^{i-1} 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 $\forall \alpha \in \mathit{Ops}(\Pi), e \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e)$.

For the base case, consider the operator α at depth 1. Since Π^{i-1} achieves l and $\mathit{Justified}(\Pi, l)$, then $l \in E_\alpha$. From Restriction 2, since $l \in \mathit{Relevant}(\alpha, l)$, $\forall e \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e)$.

For the inductive case, assume that for each operator β at depth i, $\forall e \in E_\beta$, $\mathit{Level}(l) \geq \mathit{Level}(e)$. Let α be an operator at depth $i+1$. Since Π^{i-1} is justified, there exists an operator β at depth i with precondition $p \in P_\beta$, such that $p \in E_\alpha$. From Restriction 2, $\forall q \in \mathit{Relevant}(\beta, S_g)$, $\mathit{Level}(q) \geq \mathit{Level}(p)$. From the inductive hypothesis, $\mathit{Level}(l) \geq \mathit{Level}(q)$. Therefore, $\mathit{Level}(l) \geq \mathit{Level}(p)$. Since $p \in \mathit{Relevant}(\alpha, l)$, from Restriction 2, $\forall e' \in E_\alpha$, $\mathit{Level}(p) \geq \mathit{Level}(e')$. Thus, $\forall e' \in E_\alpha$, $\mathit{Level}(l) \geq \mathit{Level}(e')$. □

Theorem 2 *Every abstraction hierarchy satisfying Restriction 2 with respect to a problem ρ is a problem-specific ordered monotonic hierarchy with respect to ρ.*

Proof: The proof is the same as the proof of Theorem 5 with Lemma 1 replaced by Lemma 2. □

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[3] John S. Anderson and Arthur M. Farley. Plan abstraction based on operator generalization. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, MN, 1988, 100–104.

[4] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991, 286–292.

[5] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. Research Report CS-92-45, Department of Computer Science, University of Waterloo, 1992.

[6] 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*, Cambridge, MA, 1977, 442–449.

[7] 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.

[8] 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*, Kluwer, Boston, MA, 1990, 125–146.

[9] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*, Lawrence Erlbaum, Hillsdale, NJ, 1991, 241–278.

[10] Jie Cheng and Keki B. Irani. Ordering problem subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989, 931–936.

[11] Jens Christensen. *Automatic Abstraction in Planning.* PhD thesis, Department of Computer Science, Stanford University, 1991.

[12] Daniel S. Eavarone. A program that generates difference orderings for GPS. Technical Report SRC-69-6, Systems Research Center, Case Western Reserve University, 1969.

[13] George W. Ernst. Sufficient conditions for the success of GPS. *Journal of the Association for Computing Machinery*, 16(4), 1969, 517–533.

[14] George W. Ernst and Michael M. Goldstein. Mechanical discovery of classes of problem-solving strategies. *Journal of the Association for Computing Machinery*, 29(1), 1982, 1–23.

[15] George W. Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving*. ACM Monograph Series. Academic Press, New York, NY, 1969.

[16] Oren Etzioni. *A Structural Theory of Explanation-Based Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1990.

[17] Oren Etzioni. A structural theory of explanation-based learning. *Artificial Intelligence*, 60(1), 1993, 93–140.

[18] Oren Etzioni and Ruth Etzioni. Statistical methods for analyzing speedup learning experiments. *Machine Learning*, forthcoming.

[19] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972, 251–288.

[20] 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), 1971, 189–208.

[21] Eugene Fink. Justified plans and ordered hierarchies. Master's Thesis, Department of Computer Science, University of Waterloo, 1992.

[22] Eugene Fink and Qiang Yang. Automatically abstracting the effects of operators. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, Morgan Kaufmann, San Mateo, CA, 1992, 243–251.

[23] Fausto Giunchiglia and Toby Walsh. Abstract theorem proving. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989, 372–377.

[24] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3), 1992, 323–390.

[25] 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.

[26] H. Altay Guvenir and George W. Ernst. Learning problem solving strategies using refinement and macro generation. *Artificial Intelligence*, 44(1-2), 1990, 209–243.

[27] Robert Holte, Robert Zimmer, and Alan MacDonald. When does changing representation improve problem-solving performance? In *Proceedings of the Workshop on Change of Representation and Problem Reformulation*, Nasa Ames Research Center, Technical Report FIA-92-06, 1992, 100–105.

[28] Glenn A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4), 1989, 285–317.

[29] 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*, Milan, Italy, 1987, 1018–1024.

[30] Craig A. Knoblock. Abstracting the Tower of Hanoi. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, Boston, MA, 1990, 13–23.

[31] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990, 923–928.

[32] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[33] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, 1991, 686–691.

[34] Craig A. Knoblock. An analysis of ABSTRIPS. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, Morgan Kaufmann, San Mateo, CA, 1992, 126–135.

[35] Craig A. Knoblock. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning.* Kluwer Academic Publishers, Norwell, MA, 1993.

[36] 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*, Anaheim, CA, 1991, 541–546.

[37] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, 1991, 692–697.

[38] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1), 1985, 35–77.

[39] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1), 1987, 65–88.

[40] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1987, 1–64.

[41] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986, 11–46.

[42] Vladimir Lifschitz. On the semantics of STRIPS. In *Proceedings of the Workshop on Reasoning about Actions and Plans*, Timberline, Oregon, 1986, 1–9.

[43] Steven Minton. Selectively generalizing plans for problem solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985, 596–599.

[44] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[45] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3), 1990, 363–392.

[46] 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), 1989, 63–118.

[47] 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*, Detroit, MI, 1989, 701–707.

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

[49] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, 1980.

[50] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[51] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16(1), 1981, 47–108.

[52] Patricia Riddle. Automating problem reformulation. In D. Paul Benjamin, editor, *Change of Representation and Inductive Bias*, Kluwer, Boston, MA, 1990, 105–124.

[53] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2), 1974, 115–135.

[54] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York, NY, 1977.

[55] Alberto Segre, Charles Elkan, and Alex Russell. A critical look at experimental evaluations of EBL. *Machine Learning*, 6(2), 1991, 183–195.

[56] David E. Smith and Mark A. Peot. A critical look at Knoblock's hierarchy mechanism. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, Morgan Kaufmann, San Mateo, CA, 1992, 307–308.

[57] David E. Smith and Mark A. Peot. Postponing conflicts in nonlinear planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC, 1993.

[58] Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2), 1981, 111–140.

[59] Austin Tate. Project planning using a hierarchic non-linear planner. Research Report 25, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, 1976.

[60] Austin Tate, James Hendler, and Mark Drummond. A review of AI planning techniques. In *Readings in Planning*, Morgan Kaufmann Publishers, San Mateo, CA, 1990, 26-49.

[61] Josh D. Tenenberg. *Abstraction in Planning*. PhD thesis, Computer Science Department, University of Rochester, 1988.

[62] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989, 681–687.

[63] 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*, Boston, MA, 1990, 78–86.

[64] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3), 1984, 269–301.

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

[66] Qiang Yang. *Improving the Efficiency of Planning*. PhD thesis, Department of Computer Science, University of Maryland, 1989.

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

[68] Qiang Yang and Josh D. Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990, 204–209.