Downward Refinement and the Efficiency of Hierarchical Problem Solving

Fahiem Bacchus and Qiang Yang*
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

September 24, 1993

Abstract

Analysis and experiments have shown that hierarchical problem-solving is most effective when the hierarchy satisfies the *downward refinement property* (DRP), whereby *every* abstract solution can be refined to a concrete-level solution without backtracking across abstraction levels. However, the DRP is a strong requirement that is not often met in practice. In this paper we examine the case when the DRP fails, and provide an analytical model of search complexity parameterized by the probability of an abstract solution being refinable. Our model provides a more accurate picture of the effectiveness of hierarchical problem-solving. We then formalize the DRP in ABSTRIPS-style hierarchies, providing a syntactic test that can be applied to determine if a hierarchy satisfies the DRP. Finally, we describe an algorithm called HIGHPOINT that we have developed. This algorithm builds on the ALPINE algorithm of Knoblock in that it automatically generates abstraction hierarchies. However, it uses the theoretical tools we have developed to generate hierarchies superior to those generated by ALPINE. This superiority is demonstrated empirically.[1].

Abbreviated Title: The Downward Refinement Property.

*This work is supported by grants from the Natural Science and Engineering Council of Canada, and ITRC: Information Technology Research Centre of Ontario. The authors' e-mail addresses are `fbacchus@logos.waterloo.edu` and `qyang@logos.waterloo.edu`.

[1]Parts of Section 2 have previously appeared at AAAI-92 [4], and parts of Section 3 have previously appeared at IJCAI-91 [3]

1 Introduction

In this paper we examine the benefits of hierarchical problem-solving under a new, and more accurate, model of search costs, and we demonstrate how a more efficient problem-solver can be constructed using the information provided by our model. Hierarchical problem-solving is accomplished by first searching for an abstract solution to the problem, that ignores certain details, and then using the intermediate states of the abstract solution as intermediate goals to decompose the search for the non-abstract solution. This technique has been used in a number of problem-solvers in AI [23, 24, 25, 26, 29, 30, 33, 32, 10].

It has long been known that the identification of intermediate states that decompose a problem can significantly reduce search [22, 21]. Hence, once we have an abstract solution providing a decomposition of the problem, we can refine it to a concrete solution with much less effort than if we had searched directly for a concrete solution. However, we also have to take into account the cost of finding the abstract solution. This is accomplished by searching in an abstract version of the problem-space. Since the abstract space is smaller and the abstract solution shorter, an abstract solution can generally be found with much less effort than a concrete solution. When the cost of finding an abstract solution and then refining it to a concrete solution is less than the cost of searching directly for a concrete solution, hierarchical problem-solving will be effective. The intuition behind using this technique, then, is that this will usually be the case.

However, there are two major difficulties with this naive analysis.

1. Not every abstract solution need be refinable. That is, the skeleton provided by the abstract solution might be inappropriate for building a concrete solution. In this case the problem-solver will have to examine alternate abstract solutions in an attempt to find one that can be refined. That is, backtracking across abstraction levels might be required. If multiple abstract solutions must be found and refinements attempted for them all, then the benefit of the hierarchical approach is no longer obvious.

2. Even if we can refine it, the abstract solution might fail to provide an effective decomposition of the problem. First, the intermediate goals provided by the intermediate states of the abstract solution might not be independent. If there is significant interaction between these intermediate goals the complexity of refining the abstract solution might approach the complexity of direct search for a concrete solution. Secondly, finding solutions to these intermediate goals might be as difficult as solving the original goal. That is, the abstract solution might fail to break the problem into easier subproblems.

Nevertheless, when it works, hierarchical problem-solving is one of the most effective techniques in practice, as demonstrated, e.g., in the work of Sacerdoti [25] and also Newell and Simon [23]. Analytical models that have been presented in the literature [15, 18] predict that an exponential speed-up is possible from this technique. However, these models ignore both of the problems presented above. In particular, they assume that every abstract solution

can be refined to a concrete solution without backtracking across abstraction levels, and they also assume that the intermediate goals generated by the abstract solution are easy and can be solved independently.

The work presented here originated with our observation that hierarchies don't always yield more efficiency, contrary to the predictions of these models. In fact, experiments with ABSTRIPS and ABTWEAK [36] demonstrated that abstraction only increases search efficiency in hierarchies where most abstract solutions are refinable (i.e., where we do not have to do much backtracking in the abstraction hierarchy). In hierarchies where this is not the case, using abstraction sometimes decreased the efficiency of the planner. Since, hierarchical problem-solving is so important in practice, and its potential benefits so great, we were motivated to understand this phenomenon more thoroughly.

To achieve this goal we construct an analytical model of the complexity of hierarchical problem-solving that takes into account the possibility of backtracking across abstraction levels. Our analytical model is a probabilistic one which yields an expected, or average, complexity, and it is parameterized by the probability that an abstract solution can be refined. When this probability is 1, we recapture the assumption of no backtracking (in this case every abstract solution can be refined). Hence, our model is a strict generalization of previous models. Using the model we are able to provide a lot of new information about the benefits of hierarchical problem solving.

Our model applies to a range of different types of hierarchies, but we were mainly interested in ABSTRIPS-style hierarchies, as these are the type most commonly found in planning systems. In ABSTRIPS-hierarchies the assumption of no backtracking across abstraction levels yields to further analysis. We say that a hierarchy possesses the *downward refinement property*, or DRP [3], if every abstract solution can be refined to a concrete solution, given that a concrete solution exists. For ABSTRIPS hierarchies we provide a formal semantic characterization of hierarchies that possess the DRP. This characterization is used to provide a useful *syntactic* condition that is sufficient to guarantee the DRP. Hence, in certain cases we can detect if a hierarchy is "good" in the sense that, under the assumption of easy and independent intermediate goals, we know, via the analytical models, that such hierarchies yield a significant speed-up over non-hierarchical planning.

Finally, we put all of these pieces together in the design of a new hierarchy generator called HIGHPOINT. This system builds on the ALPINE system of Knoblock [14], which is an automatic hierarchy generator for ABSTRIPS-hierarchies. HIGHPOINT takes the hierarchy suggested by ALPINE and improves it using information gathered during a testing phase. These improvements are based on the results of our analysis as well as the syntactic condition we develop. Our empirical results demonstrate that HIGHPOINT can generate hierarchies that offer a significant improvement in performance over those hierarchies generated by ALPINE.

In the sequel we follow the ordering used above. In Section 2, we present an analytical model of the complexity of the hierarchical approach and use that model to analyze in more depth the benefits of this technique. In Section 3, we turn our attention to the downward refinement property in ABSTRIPS-style hierarchies, presenting a semantic characterization

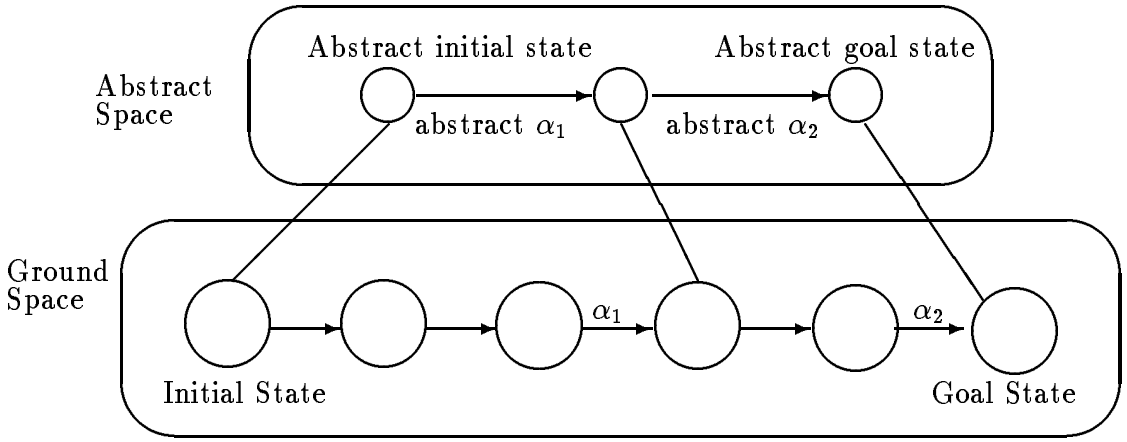

Figure 1: An abstract solution and its refinement.

of the property along with a syntactic condition sufficient to guarantee it. In Section 4, we present the hierarchy generation algorithm HIGHPOINT, along with a collection of empirical results demonstrating its effectiveness. Finally, we close with some conclusions and discussion of future work.

2 An Analytical Model

First, we present in more detail the basic problem-solving technique used by the hierarchical approach.

The problem-space consists of a collection of states along with a set of operators that map between the states. A problem consists of an initial state and a goal state, and it is solved by searching for a sequence of operators whose composition will map the initial state to the goal state. In hierarchical problem-solving an abstract version of the original, or *concrete*, problem-space is used. The abstract version is generated via some reduction, or generalization, of the operators or states in the ground space. For example, in ABSTRIPS-style hierarchies the operators are generalized by dropping some of their preconditions; this has the effect of increasing the domain of the partial function they define on the states.

A hierarchical problem-solver first searches the abstract space for a solution (see Figure 1). However, this solution cannot be used at the concrete level. For example, in ABSTRIPS-style hierarchies the abstract solution will have unfulfilled preconditions at the concrete level. Instead it can only serve as a skeletal plan for the concrete level. A correct solution at the lower level is generated by *refining* the abstract solution, and this is accomplished by inserting additional operators between the operators in the abstract solution. If we have m operators in the abstract solution, refinement to the next lower level can be viewed as solving

m "gap" subproblems. Solving the gaps amounts to finding new sequences of operators which when placed between the operators of the abstract solution generate a correct solution at the lower level.

This approach easily generalizes to a hierarchy with multiple levels of abstraction. When using such a hierarchy we plan at the most abstract level first and then successively refine our solution down the hierarchy, taking care of more and more details as we move down, until we generate a solution at the lowest, ground or concrete, level that takes care of all the details specified in the original problem. As we move down the hierarchy our solution becomes longer, and thus the number of "gap" subproblems we have to solve increases.[2]

2.1 The Space of Abstract Solutions

The total search space explored by a hierarchical problem-solver can be viewed as a tree generated by the abstraction hierarchy (see Figure 2). In this tree each node at level i represents a complete i-th level abstract solution. The children of a node represent all of the different refinements of that solution at the next (lower) level of abstraction. The leaf nodes are complete concrete-level solutions. The task in searching through the space of abstract solutions is to find a path from the root down to a leaf node representing a correct concrete-level solution. Each node on the path must be a legal i-th level abstract solution to the problem at hand and must be a refinement of the $i+1$-level abstract solution represented by its parent. That is, we are searching for an abstract solution that can be successfully refined through the levels of abstraction down to the concrete level.

The work in searching this tree comes from the work required to find the solution at each node. This solution is a refinement of the solution that has already been found at the node's parent. Hence, finding it simply involves solving all of the gap subproblems that arise when the parent's solution is moved down to this next lowest level. Since the number of gap subproblems increase as we move down the tree, the total amount of work will depend both on the number and depth of the nodes examined during search.

The root represents a special length one solution to every problem: a universal solution. Its presence is simply a technical convenience. The levels of the tree are numbered $\{n, \ldots, 0\}$ with the root being at level n and the leaves at level 0. Hence, discounting the universal solution at level n, our abstraction hierarchy has n levels.

The tree of abstract plans will have a branching factor that, in general, will vary from node to node. This branching factor is the number of $i-1$ level refinements possible for a given i level solution, i.e., the number of children a node at level i has. Let the maximum of these branching factor be B. For simplicity we will use B as the branching factor for all nodes in the tree. Note, B has no straightforward relationship with the branching factor

[2]Our model does not apply to hierarchies of macro-operators [18]. In this type of hierarchy lower-level operators are not inserted between the operators in the abstract solution. Instead, operators are inserted only at the head and the tail of the abstract solution. This means that there are a fixed number of gap subproblems (two) involved in every refinement.

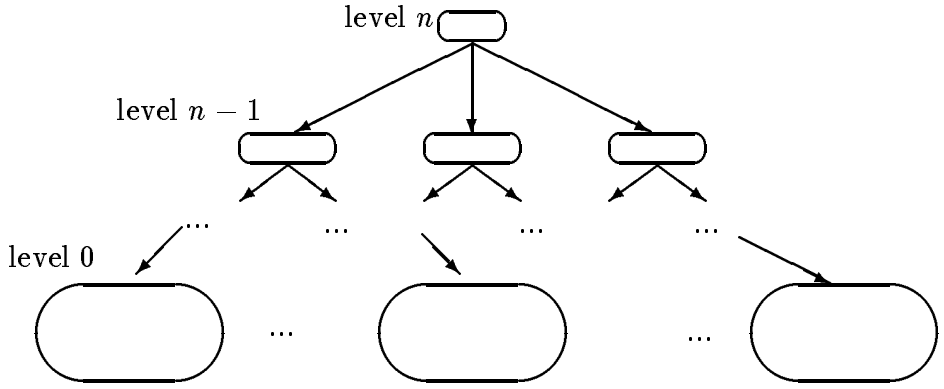

Figure 2: Our analytical model: a tree of abstract solutions, with a branching factor B and a depth $n+1$.

generated by the operators when searching for a specific solution, which we will denote by lower case b.

Under the assumption that the gap subproblems are independent, the number of refinements of a given i-th level solution, B, will be the product of the number of different solutions to each of the gap subproblems. Hence, it might be thought that B would grow exponentially as we move down the tree, as the number of gap subproblems is growing exponentially. However, as we move down the levels of abstraction the gap subproblems become more and more constrainted: at each level the gap solutions must preserve more and more conditions achieved at the higher levels of abstraction. Hence, the number of different solutions to the gap subproblems drops as we move down the tree. The exact balance between these two effects is difficult to determine, but we have found that our assumption of a constant branching factor B yields a model that is supported by our empirical results.

2.2 Initial Assumptions

At this stage for simplicity and ease of presentation we carry through our analysis under some initial assumptions.

1. *We assume that the abstraction hierarchy is regular.* In particular, we assume that it takes approximately k new operators to solve every gap subproblem where k is constant across abstraction levels. Refining a solution to the next level amounts to solving a gap subproblem between every pair of operators; hence the refined solution will be k times longer (see Figure 3). Since the root is a solution of length 1, this means that the solutions at level i are of length k^{n-i}, and that the concrete-level solution is of length k^n, which we also denote by ℓ.

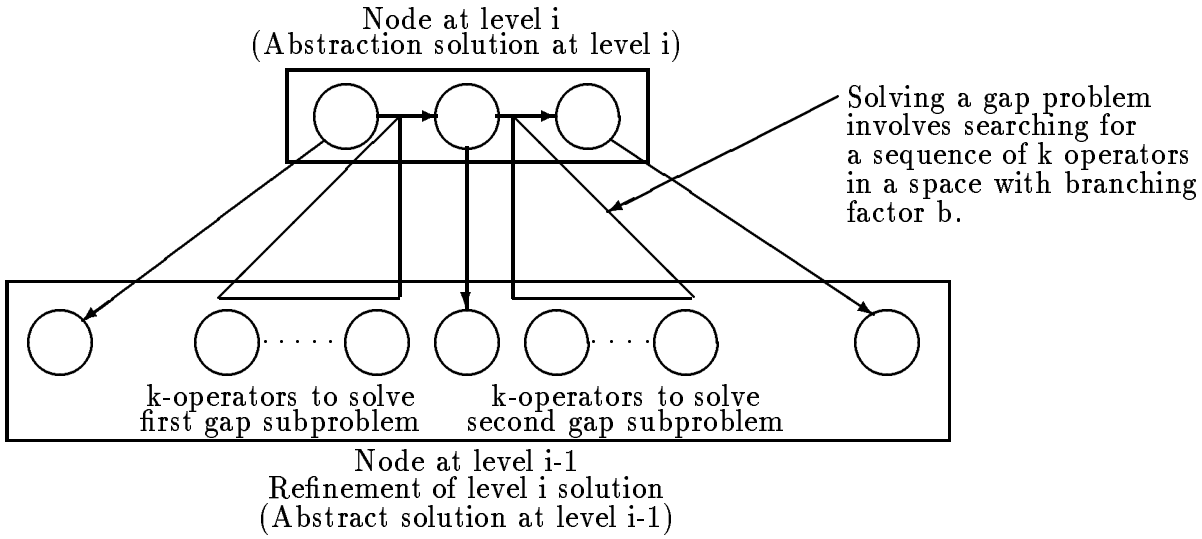

Figure 3: Refining an abstract solution.

As this assumption degenerates the value of abstraction degenerates. If we end up having gap subproblems which require solutions of length $O(\ell)$ instead of $O(k) = O(\ell^{1/n})$, then solving them will require search of $O(b^{\ell})$ where b is the branching factor generated by the operators. This is no better than search without abstraction.

2. *We assume that the individual gap subproblems can be solved without significant interaction.* If, say, r gap subproblems interact we will have to search for a plan that solves all of them simultaneously. Such a plan would be of length $O(rk)$ and would require $O(b^{rk})$ search. As rk approaches ℓ we once again degenerate to search complexity of $O(b^{\ell})$ and abstraction yields no benefits.

   These two assumptions correspond to ignoring, for now, the second problem mentioned in the introduction, i.e., that the abstract solution might fail to provide an effective decomposition of the problem. The brief discussion above indicates, however, that when these assumptions fail, hierarchical problem-solving can quickly degenerate to being worse than non-hierarchical problem-solving. Hence, these two assumptions are required before we can obtain any interesting behavior from the abstraction hierarchy. We will return to discuss in greater detail the effect of lifting these assumptions after we carry through our analysis.

3. *We assume that a concrete level solution exists.* This assumption is only to simplify our initial presentation. Actually, the analytical model we develop will also cover the case where this assumption fails, as will be discussed in the sequel.

4. *We assume that the upward-solution property holds of the hierarchy.* This property simply says that if a concrete level solution Π_0 exists then there exists a sequence of abstract solutions ending with Π_0, $\{\Pi_n, \ldots, \Pi_0\}$, such that each Π_i is an i-th level abstract solution, and Π_{i-1} is a refinement of Π_i. That is, there is a sequence of refinements that yield the concrete level plan. Tenenberg [31] has shown that this property *always* holds in any ABSTRIPS-style abstraction hierarchy. In general, it will be true of any hierarchy in which abstract solutions ignore details. In this case we can generate the required sequence of abstract solutions by starting with the concrete solution and successively simplifying it as we move up the levels of abstraction, ignoring more and more details as we move up. In the sequel we will expand our model to handle the case where this assumption fails.

2.3 The Probability of Refinement

If a hierarchy has the DRP then every solution at abstraction level i can be refined to a solution at abstraction level $i-1$. This means that once we have found a path down to level i, whose terminal node represents a correct i-th level solution, we are assured that this node has a child representing a valid $i-1$-th level solution. That is, we are assured that the node can be refined to the next lower level. Hence, we need never reconsider the initial part of the path, we just need to extend it until we reach a leaf: there is no backtracking across abstraction levels.

A reasonable way to examine the behavior of hierarchies in which the DRP fails is to assign a probability, p, to the event that a given i-th level solution can be refined to level $i-1$. The DRP now corresponds to the case $p = 1$. When $p < 1$, however, we might build a path of correct solutions from the root down to a node at level i, and then find that this node is not refinable to the next level. This will force a backtrack to the penultimate node at level $i+1$ to find an alternate level i solution, one which is refinable. This may cause further backtrack to level $i+2$, or search may progress to lower levels before backtracking occurs again.

Since we are assuming that a concrete-level solution exists, and that the upward solution property holds, we know that there is at least one path of correct solutions in the tree from the root to a leaf node. Hence, although the worst case will require an exhaustive search of the tree, search will eventually succeed in finding a good path.

What we wish to accomplish, then, is to determine the average case complexity of search in abstraction hierarchies in which (1) the probability that a given node in the abstraction search tree can be refined is p, and (2) there is at least one *good path*, i.e., a path of good nodes, from the root to a leaf in the tree.

If a node is refinable to the next level, it will have B children, under our assumption that the number of refinements be treated as a constant. Each of these children is itself refinable with probability p. During search if we encounter a node that is refinable we would have to examine all of the subtrees headed up by its B children before we can conclude that it

is a deadend. On the other hand, if the node is not refinable we can backtrack right away. Hence, as far as search is concerned, a node that is refinable has B children, while a node that is not refinable has no children.

Analytically, however, we can find the average case complexity of searching such trees by considering randomly labeled *complete* trees. A complete abstraction tree is simply a tree in which *every* node, refinable or not, has B children and which has height $n+1$. Hence, it has $N = \frac{B^{n+1}-1}{B-1}$ nodes. We randomly label each node in this tree as being refinable (*good*) with probability p, or not refinable (*bad*) with probability $1-p$. Each of the 2^N distinct trees that can be generated by this process has probability $p^g(1-p)^{N-g}$, where g is the number of good nodes in that tree. Now the average case complexity for searching this collection of trees is simply the sum of the search effort required for each tree times the probability of that tree. The average case complexity for searching all of these trees is the same as the average case complexity of searching the original set of trees in which a bad node has no children. Each original tree in which bad nodes have no children represents a set of complete trees: each one generated by a different labeling of the subtrees under the bad nodes. The probability of the original tree is exactly the same as the sum of the probabilities of the complete trees in this set.

For now, we wish to restrict our attention to the case where each tree contains at least one good path. When we generate the set of differently labeled complete trees, some of them will not contain a good path from the root to a leaf. We remove these trees, and renormalize the probabilities of the remaining trees so that they sum to 1. That is, we take conditional probabilities.

This means that if we select a random node from a random tree that contains a good path, then the probability that the node is refinable will be a bit higher than p. There is some positive probability that the node selected is on the good path, in which case it will definitely be refinable, otherwise if it is not on the good path it will be refinable with probability p. These exclusive events sum, and the probability the node is refinable becomes somewhat larger than p.

One other piece of notation we will use is $b(K, N, P)$ to denote the binomial distribution, i.e., the probability of K successes in N independent Bernoulli trials each with probability P of success.

2.4 Analytic Forms

Now we present the analytic forms that arise from the framework and assumptions presented above. It should also be pointed out that our analysis assumes that search of the abstraction tree is performed in a depth-first manner as this is the standard way in which abstraction planners like ABSTRIPS [25] and ABTWEAK [36] work.

2.4.1 NodeWork(i)

First, we examine the amount of work required to refine a node (see Figure 3) at level i, NodeWork(i). At level n we have one "gap" subproblem to solve which requires $O(b^k)$ computation: by our first assumption each gap subproblem requires k operators for its solution, and search for k operators in a space with branching factor b requires $O(b^k)$. At level $n-1$ the nodes are abstract solutions of length k, resulting in k subproblems each requiring $O(b^k)$ computation. Hence, under our second assumption of independent "gap" subproblems, we require $O(kb^k)$ computation to refine a node at this level. This trend continues so that at level i the abstract solutions generate k^{n-i} subproblems each requiring $O(b^k)$ computation. However, at level 0 the solutions are concrete; hence, they do not need to be refined. Hence, we have

$$\begin{aligned} \mathsf{NodeWork}(i) &= k^{n-i}b^k \qquad (0 < i \leq n) \\ \mathsf{NodeWork}(0) &= 0. \end{aligned} \tag{1}$$

2.4.2 Failure Probability—$F(i)$

Now we develop an expression for $F(i)$, the probability that a random subtree with root at level i fails to contain good path from its root to a leaf. A subtree can fail to contain a good path in two exclusive ways: (a) the root could be a bad node or (b) the root could be good but somewhere among its descendants all the good paths terminate before reaching level 0.

Case (b) can be analyzed using the theory of branching processes [7, 2]. A branching process is a probabilistic model with wide ranging applications. In its basic form we start with a single parent who produces some number of offspring. There is a fixed probability that n offspring will be generated, for every $n \geq 0$. Each child has in turn its own children, and the number of its children is again governed by the same distribution. The theory of branching processes analyses the number of offspring after i generations, the probability of eventual extinction, and other questions.

In case (b) if the root is good it initiates a branching process where it might have some number of good children and they in turn might have some number of good children and so on. We can consider the production of bad children as points where the process terminates. If the root is good it always produces B children (the branching factor in the space of abstract solutions), and each of those children is labeled good or bad by an independent random process, with probability of a good label being p. Hence, the number of good children of the root will be binomially distributed: $b(m, B, p)$ is the probability of it having m good children. The generating function of the binomial distribution is

$$G(s) = (1 - p + ps)^B.^3 \tag{2}$$

[3]When this generating function is expanded as a power series in s the coefficient of s^m is equal to the probability of m good (refineable) children among the B offspring, i.e., $b(m, B, p)$.

Let $G_1(s) = G(s)$ and $G_j = G_{j-1}(G(s))$, i.e., the j-th iterate of $G(s)$.

From the theory of branching processes it is known that the probability that there is no path of i good nodes from the root is $G_i(0)$. This corresponds to the probability of the branching process being extinct after i generations. For example, the probability of there being no paths of 2 good nodes from the root (i.e., the probability of no good child having a good child) is $G_2(0) = G(G(0)) = G((1-p)^B) = ((1-p) + p(1-p)^B)^B$.

Hence, for a random subtree with root at level i to fail to contain a good path from its root to a leaf, it must either have a bad root with probability $1-p$, or it must fail to have a path of good nodes of length $i-1$ from its good root. If we get to a good node at level 1 we know that there is a successful concrete solution in the subtree. This yields

$$\begin{aligned} F(i) &= 1 - p + p[G_{i-1}(0)] \qquad (i > 1), \\ F(1) &= 1 - p \end{aligned} \tag{3}$$

and we can compute $F(i)$ directly for any value of i and p.

Also of use is the fact that asymptotically $G_i(0)$ approaches a particular value Q called the *extinction probability* as $i \to \infty$.[4] When $p \leq 1/B$, $Q = 1$, and when $p > 1/B$, Q is the least positive root of the equation $G(s) = s$. So, for example, when $B = 2$ and $p = 3/4$ we have that $G(1/9) = (1/3 + 3/41/9)^2 = 1/9$, and this is the least positive number for which this is true; i.e., $p = 3/4 \wedge B = 2 \to Q = 1/9$.

Hence we obtain

$$\lim_{i \to \infty} F(i) = \begin{cases} 1 & \text{if } p \leq 1/B \\ 1 - p + p[Q] & \text{otherwise, where } G(Q) = Q. \end{cases} \tag{4}$$

From this we can identify three regions of importance:

$$\begin{array}{ll} \lim_{i \to \infty} F(i) = 1 & (p \leq 1/B), \\ \lim_{i \to \infty} F(i) \in (0,1) & (1/B < p < 1), \\ \forall i . F(i) = 0 & (p = 1). \end{array} \tag{5}$$

Intuitively, when $p < 1/B$ the expected number of good children a node has (i.e., Bp) will be less that 1. So as we move down the tree extending any particular path the chances are that eventually we will hit a node with no good children. That is, eventually all paths will terminate, and the failure probability will be one. Asymptotically the case $p = 1/B$ turns out to be the same. When $p > 1/B$ the expected number of good children rises above 1, and since the number of paths is increasing exponentially, there will be a non-zero probability that at least one of them is a good path, even as $i \to \infty$.

As p increases in the region $(1/B, 1]$ the expected number of good children of the nodes continues to climb, hence the failure probability, i.e., the limiting value of $F(i)$, decreases. In fact, $F(i)$ decreases monotonically as p increases in this region. Eventually, when $p = 1$

[4] Convergence is quite rapid: see [2] for theorems about rate of convergence.

every child must be good and all paths are good paths; hence, the failure probability becomes zero.

When we develop our complexity results for searching the tree, the failure probability will figure prominently. And we will see that these transitions in the failure probability will show up as changes in the expected complexity of search.

2.4.3 BadTreeWork(i)

Let BadTreeWork(i) be the expected amount of computation required to search a subtree with root at level i that does not contain a good path. To ensure that such a tree is a dead end we have to search until we have exhausted all candidate good paths. We always have to expand the root which is at level i and hence requires NodeWork(i) $= k^{n-i}b^k$ computation. With probability p the root is refinable and we will then have to examine all of the B subtrees under the root, all of which must be bad (otherwise the initial tree would not be bad). This process must stop by level 1, as if a node at level 1 is good this means that it can be refined to a good ground-level solution and the initial tree would not be bad. Hence, BadTreeWork(0) $= 0$, and we obtain the recurrence: BadTreeWork(i) $= k^{n-i}b^k + pB($BadTreeWork($i-1$)). By expanding the first few terms of this recurrence we can find a general expression:

$$\text{BadTreeWork}(i) = b^k k^{n-i} \frac{(pBk)^i - 1}{pBk - 1}. \tag{6}$$

2.4.4 GoodTreeWork(i)

Let GoodTreeWork(i) be the expected amount of computation required to search a *good subtree* with root at level i, i.e., a subtree which contains at least one good path from its root to a leaf. To examine a good tree we have to expand its root node. Then we must search the subtrees under the root, looking for a good subtree rooted at the next level. Once we find such a subtree we never need backtrack out of it. (There may however, be any amount of backtracking involved while searching the bad subtrees encountered before we find a good subtree.)

The root has B children, and hence between 1 and B good subtrees under it. Let m be the number of good subtrees under the root. The probability of m taking any particular value is $b(m, B, 1-F(i-1))$: each subtree can be viewed to be the result of a single Bernoulli trial where the probability of failure (a bad subtree) is $F(i-1)$. However, we also know that the case $m = 0$ is impossible; therefore, we must condition on the impossibility of this case. This amounts to taking conditional probabilities which can be computed by dividing the probabilities $b(m, B, 1 - F(i-1))$, for $m > 0$, by $1 - b(0, B, 1 - F(i-1))$ which is the probability that $m \neq 0$. If there are in fact m good subtrees, then Lemma 2.1, below, tells us that on average we will have to search $(B - m)/(m + 1)$ bad subtrees before finding a good subtree. The expected number of bad subtrees that must be searched can then be computed

by summing the average number of trees for each value of m times the probability of that value of m holding.

These observations can be put together to yield the recurrence:

$$\begin{aligned}\mathsf{GoodTreeWork}(i) &= \mathsf{NodeWork}(i) \\ &\quad + \sum_{m=1}^{B} \frac{b(m,B,1-F(i-1))}{1-b(0,B,1-F(i-1))}\Big(\big(\frac{B-m}{m+1}\big)\mathsf{BadTreeWork}(i-1)\Big) \\ &\quad + \mathsf{GoodTreeWork}(i-1).\end{aligned}$$

This expression can be simplified to the following form. Let

$$\Gamma(i) = \frac{F(i)\big(1-B(1-F(i))F(i)^{B-1}-F(i)^{B}\big)}{(1-F(i))(1-F(i)^{B})}. \tag{7}$$

Then we can express $\mathsf{GoodTreeWork}(i)$ in the following form:

$$\mathsf{GoodTreeWork}(i) = k^{n-i}b^{k}\big(\frac{k^{i}-1}{k-1}\big) + \sum_{j=1}^{i-1}\mathsf{BadTreeWork}(j)\Gamma(j). \tag{8}$$

For the derivation of this simplification see the appendix. One can see from this equation that $\Gamma(i)$ represents the average number of bad subtrees we need to examine at level i. It can also be seen, through elementary calculus, that $\Gamma(i)$ increases monotonically with $F(i)$. Intuitively, as $F(i)$ climbs the probability of a subtree rooted at level i being bad increases. Hence, the average number of subtrees that need to be searched at that level, i.e., $\Gamma(i)$, will also climb.

Finally, we close our presentation of the analytic forms with the probability lemma used in the derivation of $\mathsf{GoodTreeWork}(i)$.

Lemma 2.1 *Say we have a urn with B balls of which m are labeled 'good.' If we sample balls from this urn, at random and* without replacement *we will have to select $\frac{B-m}{m+1}$ bad balls on average until we obtain a 'good' ball.*

Proof: Let X be a random variable that takes on the value k if the $k+1$ selection is the first to yield a good ball, i.e., if we first have to select k bad balls. The lemma claims that $\mathsf{E}[X] = \frac{B-m}{m+1}$. We have

$$\mathsf{prob}[X=k] = \big(\frac{B-m}{B}\big)\big(\frac{B-m-1}{B-1}\big)\cdots\big(\frac{B-m-k+1}{B-k+1}\big)\big(\frac{m}{B-k}\big).$$

As the event $X=k$ consists of the events of k failed trials, each of which has the effect of reducing the number of bad balls by one, and a successful trial at $k+1$ at which point there are still the original m good balls but only $B-k$ balls in total. Clearly, we can have

at most $B - m$ failed trials, as by that point we have exhausted the bad balls. Hence, $\mathsf{prob}[X > B - m] = 0$, and

$$\mathsf{E}[X] = \sum_{k=0}^{B-m} k \frac{m(B-m)_k}{(B)_{k+1}},$$

where $(x)_k = x(x-1)\cdots(x-k+1)$. We can now derive the required result through standard techniques for the manipulation of sums involving binomial coefficients (see [11]). ■

2.5 Predictions of the Model

We can now examine what these expressions tell us about the complexity of hierarchical problem-solving. First, we consider the case where our initial assumptions hold. That is, we assume that there is no interaction between the gap subproblems, and that a concrete level solution exists. Since we are assuming the upward solution property (which always holds for ABSTRIPS-style hierarchies), we know that there is a good path from the root to a leaf in the tree of abstract solutions. We wish to examine how much work is required to find that path.

2.5.1 A concrete solution exists

Since the root of the tree of abstract solutions is at level n, and we know that there is a path of good nodes down to a leaf, the complexity of search is given by $\mathsf{GoodTreeWork}(n)$. There are three qualitatively different regions.

$p = 1$. In this region we have the DRP, and the assumptions of previous analyses hold. When $p = 1$, $\forall i.F(i) = 0$. Hence, $\forall i.\Gamma(i) = 0$. That is, at every level no bad subtrees need be searched as the probability of a bad subtree is zero. Our expression for $\mathsf{GoodTreeWork}(n)$, Eq. 8, thus simplifies to $\mathsf{GoodTreeWork}(n) = b^k \frac{k^n - 1}{k-1}$. Hence, we have

$$\mathsf{GoodTreeWork}(n) = O(b^k k^{n-1}) \qquad p = 1. \tag{9}$$

This result agrees with Knoblock's previous analysis which assumed the DRP [15].

This result also shows that at $p = 1$ the work required to search a good tree is simply $\sum_{i=1}^{n} \mathsf{NodeWork}(i)$. This is to be expected. With $p = 1$ we never have to backtrack, and we simply need to do the work required to refine a single abstract solution down through the levels of abstraction, i.e., one unit of $\mathsf{NodeWork}(i)$ at each level i.

$p \in (0, 1/B]$. For this region and the next it is useful to know

$$\sum_{j=1}^{n-1} \mathsf{BadTreeWork}(j) = \begin{cases} O(b^k k^{n-1}) & p < 1/B \\ O(b^k k^{n-1} n) & p = 1/B \\ O(b^k k^{n-1} (pB)^{n-2}) & p > 1/B. \end{cases} \tag{10}$$

This result is derived in the appendix.

We cannot allow $p = 0$, since we are conditioning on the existence of a good path. When $p = 0$ the probability of a good path existing will be 0. Nevertheless, we can take the limit as $p \to 0$. Analytically, this corresponds to taking the limit of $\Gamma(i)$ as $F(i) \to 1$, since $F(i) \to 1$ monotonically as $p \to 0$. Both the numerator and the denominator of $\Gamma(i)$, Eq. 7, tend to 0, but we can find its limit by a double application of l'Hôpital's rule. This yields $\Gamma(i) \to (B-1)/2$. When $p \to 0$ and we are conditioning on the existence of a good path, it is easy to see that all subtrees under a good node will become bad except a single one that must be good. $\Gamma(i)$ is the average number of bad subtrees we need to examine until we find a good subtree. Hence, on average we will have to examine half the subtrees until we find the good one.

It turns out that the region $p \leq 1/B$ is asymptotically identical to the case $p \to 0$, as in this region $\lim_{i\to\infty} F(i) = 1$. Hence, throughout the region $0 < p \leq 1/B$ we have that $\Gamma(i) \to (B-1)/2$. Applying Eq. 10 and the fact that $\Gamma(i)$ approaches a value that is independent of n to Eq. 8 we obtain:

$$\mathsf{GoodTreeWork}(n) = \begin{cases} O(b^k k^{n-1}) & p < 1/B \\ O(b^k k^{n-1} n) & p = 1/B, \end{cases} \tag{11}$$

$1/B < p < 1$. In this region for any fixed value of p, $\lim_{i\to\infty} F(i)$ lies between 1 and 0, and the limiting value decreases monotonically as p increases. Therefore, for any fixed value of p, $\Gamma(i)$ tends to a value independent of i. This value lies between the extreme values of 0, when $p = 1$, and $(B-1)/2$, when $p = 0$. From Equation 7 we can further see that this limiting value for Γ also decreases monotonically as p increases (as it increases monotonically with $F(i)$).

Again applying Eq. 10 and the independence of $\Gamma(i)$ on i to Eq. 8 we obtain:

$$\mathsf{GoodTreeWork}(n) = O(b^k k^{n-1}) + O(b^k k^{n-1}(pB)^{n-2}\gamma(p)) \qquad 1/B < p < 1, \tag{12}$$

where $\gamma(p)$ is the limiting value of $\Gamma(i)$; it is a monotonically decreasing function of p.

For any fixed value of p, $\gamma(p)$ is a constant, and we can combine the two O terms to obtain

$$\mathsf{GoodTreeWork}(n) = O(b^k k^{n-1}(pB)^{n-2}).$$

But as p approaches 1, $\gamma(p)$ drops to 0, the second O term vanishes, and we obtain our previous expression for GoodTreeWork at $p = 1$, Equation 9.

Discussion. There are two cases to consider: hierarchies with a constant number of levels and those with a variable number. In certain domains we can make n, the number of abstraction levels, vary with ℓ, the length of the concrete solution. For example, in the Towers of Hanoi domain we can place each disk at a separate level of abstraction [15].

p	$(0, 1/B)$	$1/B$	$(1/B, 1)$	1
Variable n	$O(\ell)$	$O(\ell \log(\ell))$	$O(\ell(pB)^{\log_k(\ell)})$	$O(\ell)$
Constant n	$O(\ell b^{\sqrt[n]{\ell}})$	$O(\ell b^{\sqrt[n]{\ell}})$	$O(\ell b^{\sqrt[n]{\ell}})$	$O(\ell b^{\sqrt[n]{\ell}})$

Table 1: Asymptotic Search Complexity for Different Regions when a solution exists.

In other domains, e.g., blocks world, it is not so easy to construct a variable number of abstraction levels, and n is generally fixed over different problem instances.[5]

Since each refinement multiplies the length of the abstract solution by k, we have that the concrete solution will be of size k^n, i.e., $k^n = \ell$. We want to express our results in terms of ℓ. If we can vary n with ℓ then we can ensure that k remains constant and we have that $n = \log_k(\ell)$. In this case, b^k will become a constant. Otherwise, if n is constant, $k = \sqrt[n]{\ell}$ will grow slowly with ℓ. In this case, $b^k = b^{\sqrt[n]{\ell}}$ grows exponentially with ℓ, albeit much more slowly than b^ℓ (c.f., [15]). This essential difference results in different asymptotic behavior for the two cases, n variable and n constant. Table 1 gives the results of our analysis for these two cases expressed in terms of the length of solution ℓ.

Non-hierarchical search requires $O(b^\ell)$; hence, it is evident from the table that when $0 \leq p \leq 1/B$ and when $p = 1$ abstraction has a significant benefit. If we can vary n we can obtain an exponential speed-up, and even if n is not variable, we still obtain a significant speed up by reducing the exponent ℓ to its n-th root. Our result for $p = 1$ agrees with that of Knoblock [15]: here we have the DRP and all of his assumptions hold. Our results for the region $0 \leq p \leq 1/B$, however, extend his analysis, and indicate that abstraction is theoretically useful when the probability of refinement is very low.

As we have pointed out above, $\Gamma(i) \rightarrow (B-1)/2$ in this region. That is, the average number of bad subtrees we have to examine at each abstraction level tends to its maximum value. However, the overall complexity of search remain low as it does not require much effort to search these bad subtrees. This can be seen from Equation 10. We see that the work required to search a bad subtree at each level does not depend on the size of the search tree in this region. That is, the factor B^{n-2} is missing. Since p is small we only have to search an essentially constant number of nodes in each bad subtree before realizing that all paths in that subtree terminate. Intuitively, this arises because the expected number of refinable children of every node is less that 1 when $p < 1/B$. When $p > 1/B$, on the other hand, searching a bad subtree becomes hard as the expected number of refinable children of each node becomes greater than one. This means that some proportion of all of the nodes

[5]Our asymptotic results are functions which become more and more accurate as n gets larger, i.e., they are limiting results as $n \rightarrow \infty$. Hence, in the case where n is constant it is neccessary that n be relatively large for these results to be reasonably accurate. If n is small, and the other parameters can be estimated, Equation 8 can be applied directly to give exact answers.

in the subtree (an exponential number) must be examined, which gives rise to the $(pB)^{n-2}$ factor.

Our experiments have shown, however, that this region of low refinement probability is not of much practical use. The problem is that typically B is quite large, as most abstract solutions admit a large number of alternate possible refinements. Hence, this region of low complexity is very small and close to zero. Although the probability of refinement in a hierarchy can be low, it is exceptional for it to be lower than $1/B$. That is, hierarchies with low refinement probability will typically have refinement probabilities that fall into the worst region $1/B < p < 1$, rather than in the better region $p < 1/B$. Another factor that makes hierarchies with low refinement probability bad in practice is that the constant time overhead of using a hierarchy is quite high. Hence, one would have to be solving quite large problems before the asymptotic benefits come into play.

The asymptotics given for the case of constant n are somewhat misleading. The asymptotic order of search complexity is independent of p for constant n simply because $\ell b^{\sqrt[n]{\ell}}$ dominates all of the other terms as ℓ grows. However, Equations 9, 11, and 12, indicate that the constant factors can be significantly different. In the worst region $1/B < p < 1$, for example, we have a constant factor, $(pB)^{n-2}$, that is exponential in the number of levels in the hierarchy. For the case of variable n, on the other hand, there is a quantitative difference in the asymptotic complexity of the different regions.

As p approaches $1/B$ we see that the search complexity increases by a factor of n, and as we move to the region $1/B < p < 1$ things are worse: we increase by a factor, $O((pB)^n)$, that is exponential in n. In these regions it is not always advantageous to increase the number of abstraction levels n, especially in the region $1/B < p < 1$. As p increases in this region search first becomes harder and then easier. This can be seen from Equation 12. The first O term $O(b^k k^{n-1})$ is unaffected by p, but the second term $O(b^k k^{n-1}(pB)^{n-2}\gamma(p))$, contains two competing factors. As p increases the factor $(pB)^{n-2}$ grows and the factor $\gamma(p)$ shrinks. By running a number of numerical examples, using the exact equations, we have found that search effort first grows as the $(pB)^{n-2}$ factor dominates, and then starts to shrink as the $\gamma(p)$ factor dominates. As p approaches 1, $\gamma(p)$ approaches 0, and the second term drops out and leaves us with the lower complexity of the first O term, i.e., the $p = 1$.

Intuitively, what is occuring in this region is that the bad subtrees are becoming increasingly difficult to search. We have to search a larger and larger proportion of a bad subtree before we can recognize that it is bad: analytically, the $(pB)^{n-2}$ factor in Equation 10 is growing. However, the chance that a given subtree is bad is also diminishing as p grows. So the number of bad subtrees, $\gamma(p)$, we have to search is decreasing. These two trends fight each other, with the total work required first growing and then shrinking, until we reach $p = 1$ where the number of bad subtrees we have to search, $\gamma(p)$, falls to 0.

Our analysis also tells us that if the number of possible refinements for an abstract solution, B, is large, then searching the abstraction tree is more expensive in the worst region $1/B < p < 1$. This is to be expected: the abstraction tree is bushier and in this region we have to search a significant proportion of it. Also of interest is that B does not

play much of a role outside of this region, except, of course, that it determines the size of the region. Hence, if we know that the DRP holds or if the probability of refinement is very low, we do not have to worry much about the shape of the abstraction tree. However, without such assurances it is advantageous to choose abstraction hierarchies where abstract solutions generate fewer refinements. For example, this might determine the choice of one criticality ordering over an alternate one in ABSTRIPS-style abstraction.

Another interesting result of our analysis is that search is most complex in the middle region. When the probability is low that an abstract solution can be refined more abstract solutions need to be examined before a refinable one is found. However, not much work is required to detect unrefinability. On the other hand when the probability of refinability is high not too many abstract solutions need be checked before a good one is found. The worst case is in the middle. There a significant fraction of the abstract solutions are unrefinable, and it can take a great deal of work to discover that they are unrefinable. The existence of such a phase boundary agrees with recent empirical studies of Cheeseman et al. [5], who found that the hard cases of many problems tend to cluster in the phase boundary between very many solutions and very few solutions.

2.5.2 A concrete solution does not exist

We derived an expression for $\mathsf{GoodTreeWork}(i)$ under the assumption that a concrete solution, and hence a good path from the root to a leaf, exists. However, in doing so we also obtained an expression for $\mathsf{BadTreeWork}(i)$. When no concrete solution exists neither does a good path from the root to a leaf, and hence the entire tree of abstract solutions is a bad tree. Therefore, to eliminate our third assumption and examine what occurs when no solution exists we simply have to evaluate $\mathsf{BadTreeWork}(n)$. Eq. 6 evaluated at $i = n$ yields:

$$\mathsf{BadTreeWork}(n) = b^k \frac{(pBk)^n - 1}{pBk - 1}.$$

This yields the following asymptotic behavior:

$$\mathsf{BadTreeWork}(n) = \begin{cases} O(b^k) & p < 1/Bk \\ O(b^k n) & p = 1/Bk \\ O(b^k k^{n-1}(pB)^{n-1}) & p > 1/Bk. \end{cases} \tag{13}$$

Again we can consider the cases where the number of levels in the hierarchy is constant, in which case $k = \sqrt[n]{\ell}$ and $b^k = b^{\sqrt[n]{\ell}}$, and when the number of levels is variable, in which case k remains constant and $n = \log_k(\ell)$. Table 2 gives the results of our analysis for these two cases expressed in terms of the length of solution ℓ.[6]

These results tell us that under the assumption of non-interacting "gap" problems, hierarchical planning can often detect the lack of a solution quickly. This is particularly the case

[6]Of course, no solution exists. What is meant here is that ℓ is the length of plans that would have to be examined when looking for a solution.

p	$[0, 1/Bk)$	$1/Bk$	$(1/Bk, 1)$
Variable n	$O(1)$	$O(\log(\ell))$	$O(\ell(pB)^{\log_k(\ell)})$
Constant n	$O(b^{\sqrt[n]{\ell}})$	$O(b^{\sqrt[n]{\ell}})$	$O(\ell b^{\sqrt[n]{\ell}})$

Table 2: Search Complexity for when no solution exists.

when we can construct a variable number of levels. In the optimal case when the refinement probability is very low, detection of unsolveability can occur in constant time, or in time logarithmic in the length of the plans that would have been examined during the search for a solution. When the refinement probability is high, however, to detect the lack of a solution may take as much time as finding a solution in the worst region ($1/B < p < 1$). Nevertheless, this is still a significant speed up over the $O(b^\ell)$ complexity of non-hierarchical search.

In the case of a constant number of hierarchical levels the complexity is asymptotically only slightly affected by p: there is an additional linear factor of ℓ in the worst region $(1/Bk, 1)$. However, as in the case where a solution exists, the constant factors can be significantly different. Eq. 13 shows that when the probability of refinement is high, detection of unsolveability is much harder. In this case search of the bad tree will progress a long ways down the tree before finding that there is no good path. There are B^{n-1} nodes in the tree of abstract solutions not counting level 0, and the worst case nodes are at level 1 requiring $O(b^k k^{n-1})$ computation to examine. Hence, our analysis indicates that essentially all abstract solutions will be searched, i.e., $O(b^k k^{n-1} B^{n-1})$, when p is high and no solution exists.

2.5.3 Variable Refinement Probabilities

One limitation of our model is that the refinement probability p is constant. It is not unreasonable to suppose that this probability might vary from level to level. As we refine a solution down each level we must resolve a different set of details. It is quite possible that different sets of details might vary in their ease of resolution. In this section we examine how our model can be manipulated to make some predictions about the variable probability case.

Let p_i denote the probability that an abstract solution at level i is refinable to level $i-1$. Our previous assumption of a constant refinement probability p now corresponds to the case where $\forall i. p_i = p$.

Our equation for $\mathsf{GoodTreeWork}(i)$, Eq. 8, tells us that the only changes we need to consider when we have variable refinement probabilities are those that occur to the term

$$\sum_{j=1}^{i-1} \mathsf{BadTreeWork}(j)\Gamma(j). \tag{14}$$

First, let us examine what happens to BadTreeWork(i). Our original equation for BadTreeWork(i) was derived from the recurrence BadTreeWork(i) $= k^{n-i}b^k + pB$(BadTreeWork($i-1$)). That is, to search a bad tree we need to search the root (at level i) and then with probability p the root will be good forcing a search of all B bad subtrees at the next level. When the probability of refinement varies at each level we obtain a sightly altered form of this recurrence: BadTreeWork(i) $= k^{n-i}b^k + p_iB$(BadTreeWork($i-1$). That is, now the probability the level i solution is refinable, forcing a search of all B bad subtrees under it, is p_i not p. As before, we can expand this recurrence by simply unwinding a few of its terms. The pattern is clear and we obtain

$$\text{BadTreeWork}(i) = k^{n-i}b^k(1 + p_iBk + p_ip_{i-1}(Bk)^2 + \cdots + p_i \ldots p_2(Bk)^{i-1}). \qquad (15)$$

From this equation we can observe that, typically, hierarchies where $i > j \rightarrow p_i < p_j$ perform better than hierarchies where $i > j \rightarrow p_i > p_j$. That is, hierarchies where the probability of refinement increases as we move down to lower levels are usually to be preferred to those where the probability decreases. We say typically because this characterization is too strong to hold in general. Consider, for example, a hierarchy where the probabilities are all very close to 1 but are decreasing as we move to lower levels. Such a hierarchy will still be more efficient than a hierarchy where the probabilities are increasing but are all in the middle of the worst region, $(1/B, 1)$. That is, the performance of a hierarchy will depend on the actual values of the p_i. Nevertheless, if we fix the set of possible values for the p_i we demonstrate that it is generally better to have lower refinement probabilities higher up in the tree.

Say we have a hierarchy H_1 with refinement probabilities p_n^1, ..., p_2^1. Say that we can locate two adjacent refinement probabilities, p_{j+1} and p_j, such that $p_{j+1} > p_j$, violating our desire that the refinement probabilities increase as we move down the hierarchy. Then if we can define a new hierarchy H_2 with these two refinement probabilities switched we can diminish the work required to search the bad subtrees. To be precise, let H_2 be a new hierarchy with refinement probabilities p_i^2, such that $p_i^2 = p_i^1$ for all i except that $p_j^2 = p_{j+1}^1$ and $p_{j+1}^2 = p_j^1$, i.e., p_j and p_{j+1} have been switched. Then the following proposition can be proved.

Proposition 2.2 BadTreeWork(i) *is smaller in* H_2 *for all levels* $i > j$.

Proof: Eq. 15 evaluated at $i > j$ on H_1 yields

$$k^{n-i}b^k(1 + p_iBk + \cdots + p_i \ldots p_{j+1}(Bk)^{i-j} + p_i \ldots p_{j+1}p_j(Bk)^{i-j+1} + \cdots$$

On the other hand, for H_2 we obtain

$$k^{n-i}b^k(1 + p_iBk + \cdots + p_i \ldots p_j(Bk)^{i-j} + p_i \ldots p_jp_{j+1}(Bk)^{i-j+1} + \cdots$$

All of the terms are identical except that H_1 contains the term $p_i \ldots p_{j+1}(Bk)^{i-j}$ while H_2 contains the term $p_i \ldots p_j(Bk)^{i-j}$. Since $p_j < p_{j+1}$, the work required to search a bad subtree at the top levels is less in H_2 than in H_1. ■

A similar examination of BadTreeWork(i) for $i < j$ demonstrates that the work to a bad subtree at these levels is unaffected by the switch. The only other case is $i = j$. Examining BadTreeWork(j) we find that this is the only level at which the switch will cause more work. Hence, the switch trades off more work at level j for less work at all of the levels above j. Generally, this will be a good trade off. By repeated application of this result we can see that it is generally a good idea to always switch unordered refinement probabilities. Such switching halts only when the refinement probabilities are in increasing order as we move down the hierarchy.

Of course this proposition is not sufficient to *prove* that a hierarchy with increasing refinement probabilities is always better. First, there may be cases where the increase in BadTreeWork(j) is greater than the sum of the decreases in BadTreeWork(i) for $i > j$. Furthermore, Equation 14 indicates that we must also consider the changes in $\Gamma(i)$ that arise as a result of switching the refinement probabilities.

It is much more difficult to obtain analytical results about the effects of changes in the refinement probabilities on $\Gamma(i)$. However, in various numerical computations, using our closed form for $\Gamma(i)$, Eq. 7, we have found that $\Gamma(i)$ does not change much under different orderings of the refinement probabilities.

This rough analytical analysis has been backed up by numerical studies using different values for B, k, and p_j. We have found that typically hierarchies with increasing refinement probabilities do display better performance. Intuitively, such hierarchies have the property that as a particular path becomes longer the chance of successfully completing that path becomes greater. Hence, there is less of a chance of wasting a lot of effort by growing a path almost to completion only to fail at the last moment. In hierarchies with decreasing probabilities, on the other hand, the chances of this occurring are much greater.

2.5.4 No Upwards Solution Property

The third assumption we made was that the hierarchy satisfies the upwards solution property. As we noted this assumption is always satisfied by hierarchies in which the abstract levels ignore successively more details, as in ABSTRIPS hierarchies. Nevertheless, since it is not difficult to extend our model to remove this assumption, it is worthwhile to examine what happens in this case.

If the upwards solution property fails then a concrete solution might exist without there being a good path in the tree of abstract solutions. That is, when we fail to find an abstract solution that can be refined down to a concrete solution, we can no longer conclude that a concrete solution does not exist. To preserve the completeness of our search strategy we would then have to resort to a non-hierarchical search for a solution in the concrete space.

To derive an expression for search complexity in this situation we note that the probability that the space of abstract solutions fails to have a good path from its root to a leaf is given by $F(n)$. Hence, with probability $1 - F(n)$ the tree does have a good path, in which case we must do GoodTreeWork(n) search, otherwise with probability $F(n)$ we must search a bad

tree, BadTreeWork(n) work, and then do a search of the concrete space, which takes $O(b^\ell)$ search, where ℓ is the length of the concrete solution. Hence, we obtain the expression

$$(1 - F(n))(\mathsf{GoodTreeWork}(n)) + F(n)(\mathsf{BadTreeWork}(n) + b^\ell), \tag{16}$$

for the amount of work required when the upwards solution property fails.

As discussed in Section 2.4.2, $F(n)$ approaches different limits as p varies. In the region $p < 1/B$, $F(n)$ is large, with limit 1. In this region, we will almost always fail to find a refinable abstract solution. Hence, we will perform BadTreeWork(n) amount of search and then have to search the concrete space anyway. In this case the hierarchical approach is not advantageous. As p increases beyond $1/B$, $F(n)$ decreases, and the hierarchical approach will succeed more often, resulting in significant savings. Since b^ℓ grows much faster than GoodTreeWork(n), which at worst grows as $\ell b^{\sqrt[n]{\ell}}$, we have that for any $F(n) < 1$ we will gain by first using a hierarchical search.

This can be seen as follows. Both GoodTreeWork(n) and BadTreeWork(n) are $O(\ell b^{\sqrt[n]{\ell}})$ in the region $p > 1/B$. So Eq. 16 becomes

$$\ell b^{\sqrt[n]{\ell}} + F(n)b^\ell.$$

In comparison with always doing concrete search which requires b^ℓ we will gain when

$$\ell b^{\sqrt[n]{\ell}} + F(n)b^\ell - b^\ell < 0,$$

or equivalently when

$$\ell b^{\sqrt[n]{\ell}} < (1 - F(n))b^\ell.$$

For $F(n) < 1$ this will always hold when ℓ becomes large.

Of course, this is an asymptotic result that is true only as ℓ becomes large. In practice the advantage of first doing a hierarchical search will depend on the particular value of $F(n)$. Nevertheless, if $F(n)$ is significantly below 1 it won't require very large ℓ before the advantage manifests itself.

2.5.5 The Length and Interaction of Gap Subproblems

One of the most interesting part of our analysis is what it tells us about the assumptions we have made. Our strongest assumptions were the first, that the gap subproblems required approximately the same number of operators in their solutions, and the econd, that they could be solved independently.

It is from the first assumption that we obtain the b^k factor that is pervasive throughout our analysis. By assuming that the gap subproblems each require $O(k)$ operators in their solution (or add $O(k)$ operators to the final solution) we see that these subproblems require $O(b^k)$ search to solve. If any of these gap subproblems required a longer solution, say jk, we would need $O(b^{jk})$ search to solve it. Similarly, by assuming that they could be solved

independently we get that solving j of them only requires j times the work required to solve one. If j of them interact then we might have to do $O(b^k)$ search to solve the first, then the second might generate $O(b^k)$ backtracks to find new solutions to the first before a solution could be found to the second, and so on. Hence, we might require $O(b^{jk})$ instead of $O(jb^k)$ search to solve them.

It is not difficult to see that if either of these things occur our final expressions for search complexity would have their b^k factors replaced by b^{jk} factors. This means that in the case of a variable number of hierarchical levels all of the constant b^k factors would be replaced by $O(C^j)$, i.e., some constant to the power j. In the case of a constant number of levels, all of the $b^{\sqrt[n]{\ell}}$ factors would be replaced by $(b^{\sqrt[n]{\ell}})^j$ factors. So we see that in both cases when j grows, i.e., when the number of gap subproblems that interact grows or when the length of the solutions for these subproblems grows (by a multiple of j), the complexity of search will grow exponentially with j.

This is one more reason why in practice using a hierarchy will sometimes yields worse performance. In these situations the abstraction hierarchy may be failing in its basic responsibility to break the problem down into smaller problems. For example, if the hierarchy generates gap subproblems that require solutions as long as the concrete solution, i.e., j becomes $O(l)$, then there will be no benefit to abstraction. When the abstraction hierarchy does succeed in breaking the problem into independent smaller problems then we see that we still gain asymptotically even if we have to search the entire space of abstract solutions. In the case of a constant number of levels this space has a fixed size $O(B^n)$, hence even if we only break the original problem into half we will still benefit. That is, as ℓ grows B^n independent searches of $O(b^{\sqrt{\ell}})$ will require much less work than a single search of $O(b^\ell)$.

2.6 Related Work

Korf [18] provided one of the first analyses of the benefits of using a hierarchy of abstractions in problem-solving. He investigated the use of macro operators as an abstraction device, and was able to demonstrate that an exponential speed-up was possible through the use of a logarithmic number of levels in the hierarchy. However, with a solution involving macro operators we never need to insert low-level operators in between the macro operators. Instead, we only have to map the initial state to a state where the first macro operator can be applied, and map the outcome of the last macro operator to the goal state. That is, the number of gap subproblems remains constant, unlike the situation we are modeling. Furthermore, Korf assumed that once a solution with macro operators was found it need never be reconsidered. That is, he assumed that one could always solve the initial and final gap subproblems without having to search for a new abstract plan, thus implicitly assuming that the DRP holds. In this case his results agree with ours: with the DRP an exponential speed-up is possible when we have a variable number of levels in the hierarchy.

More recently Knoblock [15] provided an analysis that inspired much of our work. He developed a framework in which the abstract solutions generate gap subproblems that must

be solved to refine the solution to the next level, and showed that this is a good model of ABSTRIPS hierarchies and the hierarchies used originally by Newell and Simon in their GPS system [23]. We have adopted this framework here. Furthermore, he also makes our first and second assumptions, i.e., that the solutions of the gap subproblems are of approximately the same length k, and that the gap subproblems are independent. However, his analysis depends on one more assumption: that there is no backtracking across abstraction levels. That is, he assumes that the DRP holds, and obtains the same results as we do for this case, i.e., the case where $p = 1$.[7] Our analysis is a significant extension of his as it provides a model parameterized by the refinement probability p.

Even more recently Williams [35] examined the case of imperfect abstraction. Since he is not primarily interested in ABSTRIPS-style hierarchies as we are, he did not assume that the upwards solution property holds. In his analysis he uses a weighted sum of complexities: with a certain probability searching the abstract solutions will succeed, otherwise it will fail and concrete level search is performed. His analysis inspired us to treat this case with our model and resulted in Section 2.5.4. His analysis does not consider an increasing number of gap subproblems as one refines the abstract solution, so it is both quite different from our model and not applicable to ABSTRIPS hierarchies. The main weakness of his analysis, however, is that he only consider hierarchies with 2 levels: there is only a single abstract level and a concrete level. This makes his model of limited practical use as most work using hierarchical problem-solving utilizes multiple levels.

Also worth mentioning, although not as directly relevant as the works mentioned above, is the work of Karp and Pearl [13] and its subsequent generalization by McDiarmid and Provan [20]. These works address the general problem of searching a tree with branches of positive cost ≥ 0. They provide average case search complexity results for three different search algorithms all of which attempt to find a path of minimal cost to a node at level n. Initially, it might be thought that this model could be applied to our problem by letting a good path be one consisting only of zero cost branches and letting all non-refinable nodes have non-zero cost branches to their children. In this case a minimal cost path will always be a good path. Unfortunately, this will not work. The major problem is that we must consider the amount of work required to expand each node, and this varies as we move down the tree of abstract solutions. Their analysis only considers the number of nodes searched, not the cost of expanding these nodes. Since this cost varies with the depth of the node, one cannot compute it from the number of nodes expanded. Another, more minor problem is that our model considers the case where a good path is known to exist. In their model this would correspond to conditioning all of the probabilities on the existence of a zero-cost path to level n. This alters the probabilities significantly, especially when p is close to zero, and hence even their results about the number of nodes searched would be quite different in our case.

[7] A minor difference is that Knoblock ends up with a $O(k^n)$ factor instead of our $O(k^{n-1})$. This is simply due to our use of the technical device of a "universal" solution at level n. In Knoblock's numbering scheme our hierarchies would have $n-1$ levels.

3 Characterizing the DRP in ABSTRIPS Hierarchies

One of the most important uses of abstraction in AI is its application to planning, where it is one of the most effective techniques for dealing with the large search spaces involved. One of the most important forms of abstraction in planning is a mechanism due to Sacerdoti [25], called ABSTRIPS-style abstraction. In this type of abstraction one abstracts away from certain details of the concrete solution by eliminating preconditions of the planning operators. Although quite an old technique, ABSTRIPS-style abstraction remains important to current research: it is used in modern large scale planners [33], has been adapted to non-linear planning [36], and has been the subject of recent theoretical studies [31, 16].

Due to its importance it is useful to examine the special case of ABSTRIPS hierarchies in more detail. As we will demonstrate in this section it is possible to derive more information about the downward refinement property in this case. Our analysis has demonstrated that an abstraction hierarchy can be very effective if it satisfies the DRP. Hence, it is useful to characterize those ABSTRIPS hierarchies that satisfy the DRP. Such a characterization can be used to check whether a given abstraction hierarchy has the DRP, and, as we will demonstrate in Section 4, it can also be used in the construction of an algorithm that automatically constructs good abstraction hierarchies.

If the hierarchy does possess the DRP we not only have theoretical predictions of its effectiveness, but we can also use a modified, simpler, search strategy that ignores backtrack points between levels of the hierarchy. By the DRP, every abstract solution is refinable to the next level, hence we will never have to use those backtrack points.

In this section, we will provide a semantics for ABSTRIPS-style abstraction. We will then use this semantics to give a semantic definition of the DRP, suitable for this type of abstraction. Our semantic formulation of the DRP yields both a better understanding of its nature and a syntactic condition sufficient for guaranteeing its presence. To be useful, we have focused on a syntactic condition that can be tested in polynomial time, allowing a hierarchy to be checked quickly for the DRP.

3.1 A Semantics for ABSTRIPS

The Language. We restrict ourselves to planning problems that can be described with a finite quantifier-free language, $\mathcal{L}$, consisting of a collection of predicates, constants, and variables. Since the language is quantifier-free, the only use of its variables is to describe parameterized operators. Such a language can be given a semantics by a traditional first-order model, with a domain of discourse, relations over the domain, and an interpretation function mapping the symbols of the language to semantic entities. The result of such a model will be the assignment of a truth-value to every formula of the language. Abstracting away from the models we can focus on their end product: the truth-value assignments. Treating these assignments as functions from the formulas to TRUE/FALSE, we can view distinct truth-value functions as alternate realities or *possible worlds* as described by $\mathcal{L}$. We

will use $\mathcal{W}$ to denote the set of *all* possible worlds. We say that a possible world w *satisfies* a formula $\lambda \in \mathcal{L}$ (set of formulas Φ), denoted by $w \models \lambda$ ($w \models \Phi$), if it assigns the truth-value TRUE to λ (to every member of Φ).

Now consider a collection of formulas of $\mathcal{L}$, Φ. If this collection is consistent (i.e., if there is at least one world $w \in \mathcal{W}$ such that w satisfies Φ), then usually there will be many possible worlds which satisfy Φ. Let Σ_Φ denote the set of *all* worlds in $\mathcal{W}$ that satisfy Φ.[8] If we add new formulas to Φ, Σ_Φ will shrink in size, and if we remove formulas from Φ, Σ_Φ will grow in size.[9] Furthermore, if we allow arbitrary collections of formulas, it can be the case that for two distinct sets of formulas Φ and Ψ we have $\Sigma_\Phi = \Sigma_\Psi$ (in the case where Φ is logically, but not syntactically, equivalent to Ψ).

State Descriptions. In planning, $\mathcal{L}$ is used to write state descriptions: collections of formulas that are partial descriptions of the state of the world. We restrict our attention to planning problems where the state descriptions are finite collections of ground literals (a typical situation for STRIPS-style planners). Under this restriction two state descritions will be logically equivalent if and only if they are syntactically equivalent. That is, we have that for two state descriptions $S1$ and $S2$, $\Sigma_{S1} = \Sigma_{S2}$ if and only if $S1 = S2$. This follows from the fact that for every consistent collection of literals there exists a possible world that satistfies only those literals.

In addition to the literals in the state description we typically have some collection of domain-dependent integrity, or consistency, constraints that the state must satisfy. For example, in a blocks world domain a state description cannot specify that the robot is holding a block and at the same time specify that the robot's hand is empty. Such constraints are often expressed as state invariant domain axioms. However, since it is complex to reason directly with these axioms, planning systems often deal with these constraints by simply requiring that the operators preserve consistency. Hence, if the planner starts in a consistent initial state its plans (operator sequences) will never generate an inconsistency. For example, an operator that causes the robot to pick up an object will always remove hand-empty from the state description. Let $\mathcal{CW}$ denote the set of possible worlds in $\mathcal{W}$ that satisfy all of the domain constraints.

Now consider a set of ground literals of $\mathcal{L}$, S. This set will be a legal state description if it is satisfied by at least one world in $\mathcal{CW}$, i.e., if $\Sigma_S \cap \mathcal{CW} \neq \emptyset$. This insures both that S is logically consistent and that it does not violate any of the consistency constraints.

For example, consider a simple robot domain where a robot can travel between rooms (see Figure 4). The rooms are separated by doors that can be opened or closed by the robot. The predicates in our language for this domain would include type predicates, e.g., `IsDoor(Door12)`, predicates indicating the status of the robot, e.g., `Inroom(Room1)`, and

[8] Hence, Σ_Φ satisfies "All I know is Φ" in the sense of Levesque [19].

[9] This corresponds to the possible worlds notion of knowledge, where more knowledge corresponds to a smaller set of accessible worlds [12].

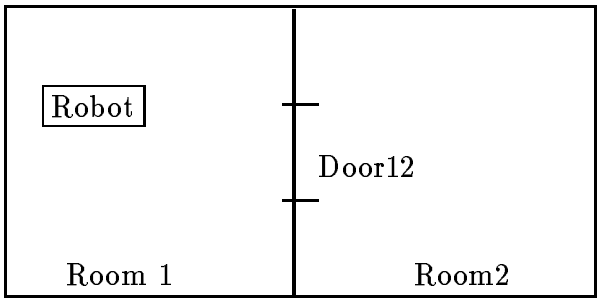

Figure 4: Part of a simple robot planning domain.

predicates indicating the status of the doors, e.g., `Open(Door12)`. If $S = \{\texttt{Inroom(Room1)}\}$, then the set of possible worlds Σ_S would include many worlds. All of these worlds will satisfy `Inroom(Room1)`, but some worlds will satisfy `Open(Door12)` while others will satisfy $\neg$`Open(Door12)`. Included in Σ_S are a number of worlds that also satisfy the consistency constraints, i.e., that are in $\mathcal{CW}$.[10]

Operators. In addition to state descriptions, planning systems contain operators that, in the case of STRIPS-style planners, are partial functions between state descriptions. A STRIPS-style operator α is defined by three sets of literals, its preconditions, $\mathsf{Pre}(\alpha)$, its add list, $\mathsf{Add}(\alpha)$, and its delete list, $\mathsf{Del}(\alpha)$. Applying an operator α to a state description S results in a new state description $\alpha(S)$ that is obtained by removing from S all the literals in $\mathsf{Del}(\alpha)$, and then adding to S the literals in $\mathsf{Add}(\alpha)$. That is, $\alpha(S) = (S - \mathsf{Del}(\alpha)) \cup \mathsf{Add}(\alpha)$, where '$-$' used here as the set difference operator.

Semantically, operators correspond to partial functions between sets of possible worlds. More precisely, consider the power set of $\mathcal{W}$, the set of possible worlds. A proper subset of this power set is the set of *describable situations* $\mathcal{DS}$. This collection of sets of worlds satisfies the condition that for each $\Sigma \in \mathcal{DS}$ there exists a legal state description S such that $\Sigma = \Sigma_S$.[11] Furthermore, as noted above, the corresponding state description S is unique. Hence, an operator α corresponds semantically to a partial function $\hat{\alpha}$ from $\mathcal{DS}$ to $\mathcal{DS}$, such that:

1. $\Sigma \in \mathcal{DS}$ is in the domain of $\hat{\alpha}$ if and only if $\Sigma \subseteq \Sigma_{\mathsf{Pre}(\alpha)}$.

[10] Σ_S will also contain worlds violating the consistency constraints. However, these worlds will not affect the correctness of plans initiated from state S. That is, as long as the operators preserve the consistency constraints we will have that for any state S' such that S' is the result of some operator application to S, $\Sigma_{S'}$ will contain at least one world in $\mathcal{CW}$, i.e., S' will also be a legal state description.

[11] Not every member of the power set of $\mathcal{W}$ can be described by a legal state description: some sets of possible worlds might require disjunctions of literals in their description, while others might contradict the consistency constraints.

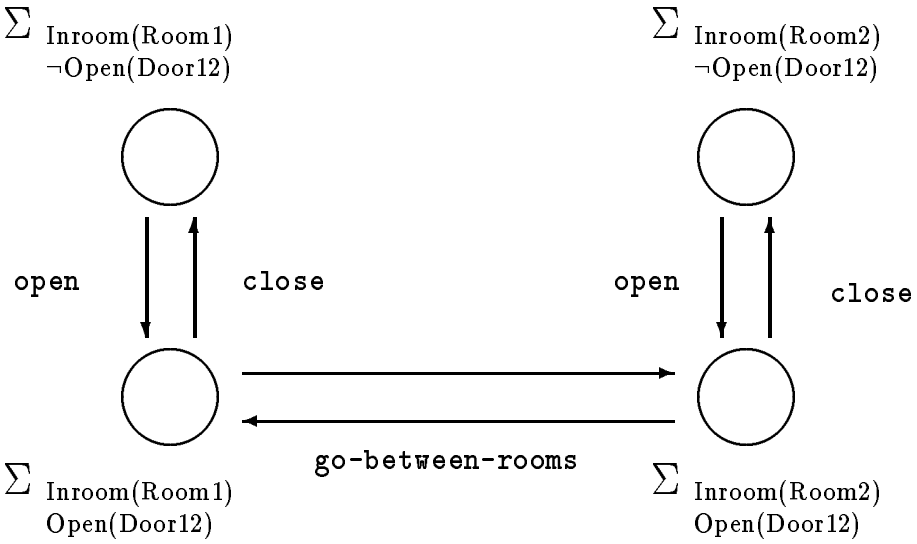

Figure 5: Semantic representation of the robot planning domain.

2. $\hat{\alpha}(\Sigma) = \Sigma_{\alpha(S)}$, where S is the state description corresponding to Σ.

That is, the semantics of an operator is determined by mapping its syntactic effects into the semantic domain.

In general, a planning system will contain operator templates instead of operators. These templates contain variables which specify parameterized versions of the actual operators. The operators themselves are generated by instantiating the variables in the template. We will often refer to these operator templates as if they were actual operators. In these cases we are implicitly referring to all of the template's instantiations.

Consider again our simple robot domain. The operator templates for this domain are described in Table 3. Semantically, the instantiations of these operators map between collections of possible worlds. These collections of worlds are specified by their corresponding state description. In Figure 5, we represent these sets of possible worlds as nodes in a graph, and the operators become arcs connecting these nodes.

Planning Problems and Plans. A plan Π is a sequence of operators $\alpha_1, \ldots, \alpha_n$. A planning problem is a pair of state descriptions $\langle I, G \rangle$, where I is the initial state, and G is the goal state. If we apply the plan Π to the initial state I the operators in Π will define a sequence of state descriptions $S_0, \ldots, S_n$ resulting from the application of the operators: $S_0 \stackrel{\alpha_1}{\mapsto} S_1 \stackrel{\alpha_2}{\mapsto} \ldots \stackrel{\alpha_n}{\mapsto} S_n$, where $S_0 = I$. A plan Π is a *solution* to a planning problem $\langle I, G \rangle$, or is *correct* with respect to $\langle I, G \rangle$, if the sequence of state descriptions generated by applying Π to I satisfies two conditions:

Preconditions	*Adds*	*Deletes*
`go-between-rooms(z,x,y)`		
`IsDoor(z), Connects(z,x,y), Inroom(x), Open(z)`	`Inroom(y)`	`Inroom(x)`
`open(z)`		
`IsDoor(z), ¬Open(z)`	`Open(z)`	`¬Open(z)`
`close(z)`		
`IsDoor(z), Open(z)`	`¬Open(z)`	`Open(z)`

Table 3: Operator templates in the simple robot domain.

1. $\mathsf{Pre}(\alpha_i) \subseteq S_{i-1}$,
2. $G \subseteq S_n$.

That is, the plan is correct if the preconditions of each operator are satisfied in the state to which it is applied, and the final state satisfies the goal G.

Semantically, a planning problem corresponds to a pair of sets from $\mathcal{DS}$, $\langle \Sigma_I, \Sigma_G \rangle$. A correct plan Π for $\langle I, G \rangle$ corresponds to a semantic solution $\hat{\Pi}$ consisting of a sequence of functions $\hat{\alpha}_1, \ldots, \hat{\alpha}_n$ that traverse through $\mathcal{DS}$ such that Σ_I is in the domain of $\hat{\alpha}_1$, each intermediate set of possible worlds, $\Sigma_{S_{i-1}}$, is in the domain of the function next applied, $\hat{\alpha}_i$, and the final set of possible worlds Σ_{S_n} is a subset of Σ_G. It is not difficult to see that every syntactic solution has a corresponding semantic solution and vice versa.

For example, in our simple robot domain if the planning problem is

$$\langle \{\texttt{Inroom(Room1)}, \neg\texttt{Open(Door12)}\}, \{\texttt{Inroom(Room2)}, \neg\texttt{Open(Door12)}\} \rangle,$$

then a possible solution is the sequence

$$\texttt{open(Door12)} \mapsto \texttt{go-between-rooms(Door12,Room1,Room2)}, \mapsto \texttt{close(Door12)}.$$

Semantically this solution corresponds to a path leading from the upper left node in Figure 5 to the upper right node.

3.2 Abstraction

In ABSTRIPS-style abstraction abstract operators are generated by eliminating preconditions using the following scheme.[12] Every predicate symbol P in the language $\mathcal{L}$ is assigned one of a

[12] More elaborate schemes are possible, but we will not consider them here.

Criticality	Predicates
2	`IsDoor, Connects`
1	`Inroom`
0	`Open`

Table 4: A criticality assignment.

finite number of integer criticality values $crit(P)$. The number of levels of abstraction is equal to the number of distinct criticality values. In particular, let there be $n+1$ different criticality values corresponding to the integers $\{0, 1, \ldots, n\}$, where the highest level of abstraction is n and level 0 corresponds to the concrete level where no abstraction occurs.

Abstract Operators and Plans. Given these abstraction levels we can define a syntactic abstraction operator **Abs** which maps a set of literals S to a new set of literals $\mathsf{Abs}(i, S)$, its i-th level abstraction, where $0 \leq i \leq n$. $\mathsf{Abs}(i, S)$ is defined by removing from S all literals with criticality value less than i, where every literal has the criticality value of its predicate symbol. We can extend **Abs** so that it can be applied to an operator α to yield an abstract operator $\mathsf{Abs}(i, \alpha)$. The abstract operator has the same add and delete lists as α but it has a abstracted precondition list. That is, $\mathsf{Add}(\mathsf{Abs}(i, \alpha)) = \mathsf{Add}(\alpha)$ and $\mathsf{Del}(\mathsf{Abs}(i, \alpha)) = \mathsf{Del}(\alpha)$, but $\mathsf{Pre}(\mathsf{Abs}(i, \alpha)) = \mathsf{Abs}(i, \mathsf{Pre}(\alpha))$. Extending **Abs** further we can apply it to plans. If Π is a plan, then $\mathsf{Abs}(i, \Pi)$ is an i-th level abstract plan where every operator $\alpha \in \Pi$ has been replaced by its i-th level abstraction $\mathsf{Abs}(i, \alpha)$. Note that the ordering of the operators has not been disturbed.

Finally, we say that an i-th level abstract plan $\mathsf{Abs}(i, \Pi)$ is a i-th level *abstract solution* to the planning problem $\langle I, G \rangle$ if it is correct with respect to the problem with an abstracted goal. That is, if $\mathsf{Abs}(i, \Pi)$ is a solution to $\langle I, \mathsf{Abs}(i, G) \rangle$. As before, a 0-th level solution is called a *concrete* solution.[13]

For example, in our simple robot domain one possible criticality assignment is shown in Table 4. The templates for the level 1 abstract operators are shown in Table 5. A solution to the planning problem

$$\langle \{\texttt{Inroom(Room1)}, \neg\texttt{Open(Door12)}\}, \{\texttt{Inroom(Room2)}, \neg\texttt{Open(Door12)}\} \rangle$$

at abstraction level one is simply $\mathsf{Abs}(1, \texttt{go-between-rooms(Door12, Room1, Room2)})$.

[13]This method of abstraction contains some arbitrary choices. For example, only the precondition list of the operators is abstracted, not their Add or Del lists. Similarly, only the goal is abstracted, not the initial state. Other choices are possible, but the choices we have made are the traditional ones, following Sacerdoti [25], and they allow our results to be more readily related to previous work. It is an interesting open question whether or not other choices have a significant impact on the efficiency of abstraction.

Preconditions	*Adds*	*Deletes*
Abs(1,go-between-rooms(z,x,y)		
IsDoor(z), Connects(z,x,y), Inroom(x)	Inroom(y)	Inroom(x)
Abs(1,open(z))		
IsDoor(z)	Open(z)	¬Open(z)
Abs(1,close(z))		
IsDoor(z)	¬Open(z)	Open(z)

Table 5: Sample abstract operator templates in the simple robot domain.

It should be noted that as a consequence of our notation $\mathbf{Abs}(0, S) = S$, $\mathbf{Abs}(i, S) \subseteq \mathbf{Abs}(i-1, S)$, and $\mathbf{Abs}(i, \mathbf{Abs}(j, S)) = \mathbf{Abs}(j, S)$ if $j \geq i$. Further, if Π is a solution for $\langle I, G\rangle$ then $\mathbf{Abs}(i, \Pi)$ will be a solution to $\langle I, \mathbf{Abs}(i, G)\rangle$, i.e., an i-level abstract solution. However, if $\mathbf{Abs}(i, \Pi)$ is an abstract solution, Π will probably not be a solution as some of its operators might have unsatisfied lower level preconditions.

Semantically, this style of abstraction has an easy description. The abstraction of a state description will correspond to a larger set of possible worlds, i.e., $\Sigma_S \subseteq \Sigma_{\mathbf{Abs}(i,S)}$. The semantic function corresponding to the abstract operator has the same definition as the concrete level function, it simply has a (potentially) larger domain. In particular, its domain now includes all $\Sigma \in \mathcal{DS}$ such that $\Sigma \subseteq \Sigma_{\mathbf{Abs}(i,\mathbf{Pre}(\alpha))}$ which is a superset of $\Sigma_{\mathbf{Pre}(\alpha)}$.

We can extend our abstraction operator **Abs** so that it can be applied to semantic entities. In particular, we define $\mathbf{Abs}(i, \hat{\alpha})$, the i-th level abstraction of a semantic operator $\hat{\alpha}$, to be the semantic counterpart of the i-th level abstraction of the corresponding syntactic operator α. That is, let $\beta = \mathbf{Abs}(i, \alpha)$, then $\mathbf{Abs}(i, \hat{\alpha}) = \hat{\beta}$. Similarly, the i-th level abstraction of a collection of possible worlds $\Sigma \in \mathcal{DS}$, i.e., $\mathbf{Abs}(i, \Sigma)$, is simply $\Sigma_{\mathbf{Abs}(i,S)}$, where S is the set of literals that defines Σ. In other words, we define abstraction on the semantic entities by first performing abstraction on their syntactic counterparts and then mapping the abstract syntactic entities to semantics ones. Since abstraction preserves the property that the state descriptions are collections of literals, we continue to have a one-to-one correspondence between abstract solutions (plans) and sequences of semantic function applications, where the semantic functions now correspond to abstract versions of the operators.

3.3 The Downward Refinement Property in ABSTRIPS

Let us fix an arbitrary planning problem $\langle I, G\rangle$. In this section our discussion will be about plans, at various levels of abstraction, that are intended as solutions to this fixed

problem. For simplicity, we will augment every such plan with special initial and terminal operators. The initial operator α_0, has no preconditions and has I as its add list. The terminal operator α_{k+1} has G as its preconditions and an empty add list. Both have empty delete lists. Semantically, $\hat{\alpha}_0$ is a function whose domain is all of $\mathcal{DS}$ and whose range is Σ_I; $\hat{\alpha}_{k+1}$ is an identity function whose domain is $\{\Sigma | \Sigma \in \mathcal{DS} \wedge \Sigma \subseteq \Sigma_G\}$. Hence, a plan will have the form $\stackrel{\alpha_0}{\mapsto} S_0 \stackrel{\alpha_1}{\mapsto} \cdots \stackrel{\alpha_k}{\mapsto} S_k \stackrel{\alpha_{k+1}}{\mapsto} S_{k+1}$, where $S_0 = I$ and $S_k = S_{k+1}$. The advantage of this convention is that all of the states in the sequence of states defined by a non-augmented plan are sandwiched by a pair of operators in the augmented plan. Note also that α_0 is unaffected by abstraction, as it has no preconditions, but α_{k+1} is affected, as it has the goal as its precondition and the goal is altered by abstraction.

Semantically, a plan will correspond to a semantic solution $\hat{\Pi}$ to the planning problem $\langle \Sigma_I, \Sigma_G \rangle$. Using the convention of augmented plans, this solution is a sequence of function applications with a corresponding sequence of semantic states: $\stackrel{\hat{\alpha}_0}{\mapsto} \Sigma_{S_0} \stackrel{\hat{\alpha}_1}{\mapsto} \ldots \stackrel{\hat{\alpha}_k}{\mapsto} \Sigma_{S_k} \stackrel{\hat{\alpha}_{k+1}}{\mapsto} \Sigma_{S_{k+1}}$, where $\Sigma_{S_0} = \Sigma_I$, $\Sigma_{S_k} \subseteq \Sigma_G$, $\Sigma_{S_k} = \Sigma_{S_{k+1}}$, and each $\Sigma_{S_i} \in \text{Domain}(\hat{\alpha}_{i+1})$.

3.3.1 Refinement in ABSTRIPS

We already know what the DRP is: every abstract solution can be refined. Hence, to offer a formal definition of the DRP we need only pin down the notion of what is to count as a refinement in an ABSTRIPS hierarchy. This turns out to be a fairly subtle notion. To understand the problem, consider the refinement strategy that at each level simply ignores the abstract plan produced at the previous level and plans completely from scratch at that level. Such a strategy will produce a sequence of abstract plans, one at each level, finally planning from scratch at the concrete level producing a concrete plan. However, there is clearly no benefit in using abstraction in this case, as the planning at the concrete level could have been done immediately.

We want our notion of refinement to be a process in which the abstract solution is used as a skeleton around which the lower level solution is constructed. As our discussion in Section 2.5.5 indicates, it is only when we can utilize the intermediate states provided by the abstract solution that the hierarchical approach is effective.

The problem of finding a reasonable definition of refinement has been studied by Knoblock et al. [16], and we adopt their approach, but adapt it to our semantics.

Monotonic Refinement. Knoblock et at. [16] developed the notion of a *monotonic refinement*. A monotonic refinement is a refinement of an abstract solution that is a "good" refinement in the sense that it uses as much of the work done at the previous level as possible.

When we refine an abstract solution to the next level we want to use that solution as a skeleton for the solution at the next level. This implies that the new, refined, solution should contain as a subsequence all of the operators of the abstract plan. Of course, since we are at a lower level of abstraction we do have to replace the operators in the abstract plan by their lower level, less abstract, versions, but other than this change all of the operators and

their sequencing should be preserved in the refined solution. In moving to the lower level, the operators will have additional preconditions. Hence, the refined solution must contain additional operators whose purpose is to achieve these preconditions. However, to insure that the refinement is using as much of the work done at the previous level as possible, these new operators should be added for the sole purpose of achieving the lower level preconditions. This condition can be ensured if we require that higher level conditions, i.e., literals with higher criticality, achieved by the abstract plan are not violated by the added operators.

These intutions lead to the following definition given in terms of *semantic* entities:

Definition 3.1 [Monotonic Refinement] Let $\hat{\Pi}_1$ and $\hat{\Pi}_2$ be abstract semantic solutions, with $\hat{\Pi}_1$ being an i-th level solution and $\hat{\Pi}_2$ an $i-1$-th level solution. $\hat{\Pi}_2$ is a *monotonic refinement* of $\hat{\Pi}_1$ if the following two conditions hold.

1. $\hat{\Pi}_1$ is a subsequence of $\mathsf{Abs}(i, \hat{\Pi}_2)$.

2. Let $\mathsf{Abs}(i, \hat{\alpha}_j)$ and $\mathsf{Abs}(i, \hat{\alpha}_{j+1})$ be any pair of adjacent operators in $\hat{\Pi}_1$,[14] and let Σ be the semantic state between these two operators in $\hat{\Pi}_1$, i.e., $\overset{\mathsf{Abs}(i,\hat{\alpha}_j)}{\mapsto} \Sigma \overset{\mathsf{Abs}(i,\hat{\alpha}_{j+1})}{\mapsto}$ is part of the sequence of operator applications in $\hat{\Pi}_1$.

   Now by condition (1) there are two corresponding operators $\mathsf{Abs}(i-1, \hat{\alpha}_j)$ and $\mathsf{Abs}(i-1, \hat{\alpha}_{j+1})$ in $\hat{\Pi}_2$ such that the first preceeds the second. Let Σ' be any semantic state in $\hat{\Pi}_2$ lying between these two operators. The second condition is then stated as the requirement that $\Sigma' \subseteq \mathsf{Abs}(i, \Sigma)$.

The first condition is simply the requirement that a monotonic refinement contain the abstract solution, modulo the fact that in the refinement all of the operators are at a lower level of abstraction.

The second condition ensures that the operators added to the refinement, $\hat{\Pi}_2$, do not interfere with higher level conditions already achieved by the abstract plan, $\hat{\Pi}_1$. Consider Σ, the semantic state between two operators in the abstract plan. The state description corresponding to Σ contains all of the literals achieved by $\hat{\Pi}_1$ up to this point. Since we are not abstracting the add lists nor the initial state this state description might contain low level literals (i.e., literals with criticality less than i). Although, we do not want a monotonic refinement to violate the high level achievements of the abstract plan, we do not object to changes in the low level achievements. Hence, we use $\mathsf{Abs}(i, \Sigma)$ in our condition instead of Σ. The state description corresponding to $\mathsf{Abs}(i, \Sigma)$ contains only the literals of criticality greater than or equal to i that have been achieved by the abstract plan up to this point. Consider now Σ' a state in $\hat{\Pi}_2$ that lies between the operators inherited from the abstract solution. By requiring that $\Sigma' \subseteq \mathsf{Abs}(i, \Sigma)$, we are requiring that all of the literals in the state description corresponding to $\mathsf{Abs}(i, \Sigma)$ be contained in the state description corresponding

[14] $\hat{\Pi}_1$ being an i-th level solution contains i-th level abstract operators, i.e., operators of the form $\mathsf{Abs}(i, \hat{\alpha})$.

to Σ'.[15] That is, we are requiring that all of the high level achievements of the abstract plan be perserved by the refinement.

Knoblock et al. [16] gave a syntactic definition of monotonic refinement. It is based on the idea that a monotonic refinement has the property that if it is moved back to the higher level of abstraction and then redundant operators are removed one recovers the abstract plan. This requires a careful definition of operator redundancy. Our semantic definition of a monotonic refinement is at least as strong as Knoblock et al.'s definition. In particular, it can be shown that if $\hat{\Pi}_2$ is a monotonic refinement of $\hat{\Pi}_1$ in our sense, and if neither contains redundant operators in Knoblock et al.'s sense, then the syntactic versions of $\hat{\Pi}_1$ and $\hat{\Pi}_2$ satisfy Knoblock et al.'s definition of monotonic refinement.

The Downward Refinement Property.

Definition 3.2 [DRP] An ABSTRIPS hierarchy has the *downward refinement property* if and only if for every i-th level abstract solution $\hat{\Pi}$, there is a monotonic refinement of $\hat{\Pi}$ at level $i-1$, for $0 < i \leq n$. ■

In other words, given our definition of a refinement, the DRP has a straightforward definition. Now we can provide a sufficient condition for the DRP.

Theorem 3.3 *Let $\hat{\Pi}_1$ be any i-th level abstract solution to any planning problem that has a concrete solution. Let $\mathsf{Abs}(i, \hat{\alpha}_j)$ and $\mathsf{Abs}(i, \hat{\alpha}_{j+1})$ be any pair of adjacent operators in $\hat{\Pi}_1$, and let Σ be the semantic state between these two operators in $\hat{\Pi}_1$.*

Suppose that for every semantic state Σ_1 such that $\Sigma_1 \in \mathrm{Range}(\mathsf{Abs}(i-1, \hat{\alpha}_j))$ and $\Sigma_1 \subseteq \mathsf{Abs}(i, \Sigma)$, there exists a state Σ_2, such that

1. *$\Sigma_2 \subseteq \mathsf{Abs}(i, \Sigma)$, $\Sigma_2 \in \mathrm{Domain}(\mathsf{Abs}(i-1, \hat{\alpha}_{j+1}))$, and*
2. *a solution $\hat{\Pi}_2$ consisting of $i-1$ level operators exists for the problem $\langle \Sigma_1, \Sigma_2 \rangle$, such that for every state Σ' in $\hat{\Pi}_2$, $\Sigma' \subseteq \mathsf{Abs}(i, \Sigma)$.*

Then the DRP is satisfied by the hierarchy. ■

A proof of this theorem is provided in the appendix. Intuitively, when we refine an abstract solution we see that the semantic state that is the result of an operator application may no longer be in the domain of the subsequent operator, due to as yet unsatisfied low level preconditions. The theorem states that the hierarchy has the property that a sequence of new operators can be found that will reconnect the two operators (see Figure 6). Furthermore, this sequence has the property that it does not affect any higher level conditions; i.e., at the i-th level nothing is changed.

[15] Note that more literals in the state description means less worlds in the set of world corresponding to that state description. That is, the set inclusions for the semantic entities go the opposite direction of the set inclusions for the syntactic entities.

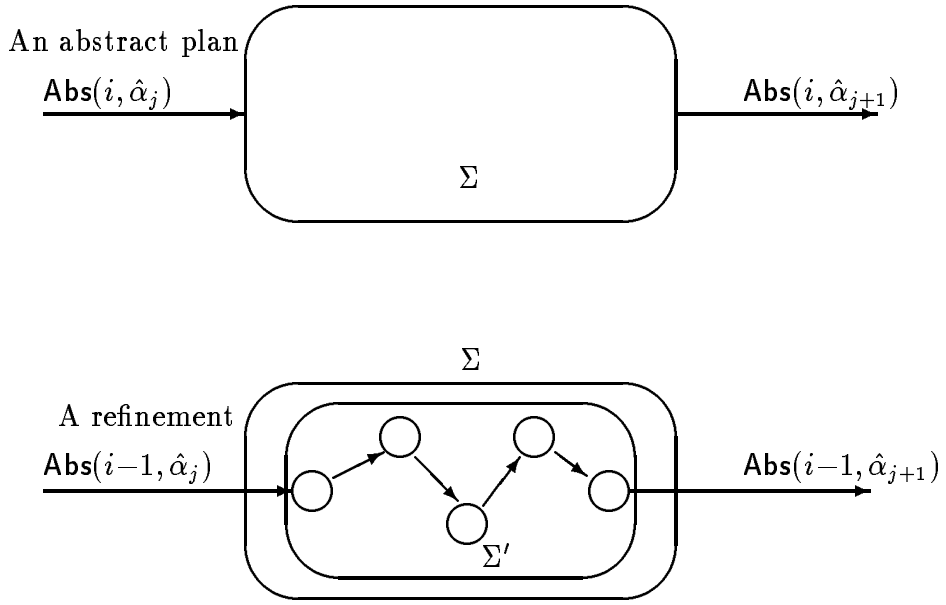

Figure 6: A plan segment and its refinement.

From this semantic condition we can derive a number of syntactic tests that are sufficient to detect if a hierarchy possesses the DRP. A number of different tests are described in our previous work [3]. Here, however, we will just present one of the more general ones. It is a test used by our hierarchy generation algorithm HIGHPOINT, described in Section 4.

The syntactic test basically checks condition 2 of the above semantic condition for each pair of operators that could possibly appear in sequence in a plan. Note that, as pointed out above, we are actually referring to every pair of instantiated operators; different instantiations generate distinct operators. There is a difficult in testing Condition 2 syntactically, however; the condition simply specifies that a solution $\hat{\Pi}_2$ exists for the problem $\langle \Sigma_1, \Sigma_2 \rangle$, it does not specify how long this solution is. Clearly, it is not computationally tractable to check for the existence of any solution. Instead, we realize our test for this condition by testing for solutions of length k. This means that we only need $O(1)^k$ work to perform this test for each pair of operators. We keep k fixed and small so that the test can be run efficiently. An alternate way of realizing the test is to simply search for a solution under a fixed time bound. Both of these approaches mean that the syntactic test is only a sufficient, not necessary test for the DRP.

Definition 3.4 [k-ary Necessary Connectivity] Let $\mathcal{O}_{\geq i}$ be the set of operators whose add and delete lists contain at least one literal with criticality value greater than or equal to i. If every pair of operators α_1, α_2 from $\mathcal{O}_{\geq i}$ passes the *k-ary connectivity test*, then levels i and $i-1$ are k-ary connected. If every pair of levels i, $i-1$ in the hierarchy is k-ary connected

then we say that the hierarchy satisfies the condition of k-ary necessary connectivity. ■

Definition 3.5 [k-ary Connectivity Test] This test is run on a pair of operators α_1 and α_2 and is also dependent on the level of abstraction i. The test is in the form of an implication. Hence, a pair of operators will pass the test if they fails to satisfy the first condition, i.e., the antecedent. For the antecedent the conditions are

1. $\mathsf{Abs}(i, \alpha_1(\mathsf{Pre}(\alpha_1))) \cup \mathsf{Abs}(i, \mathsf{Pre}(\alpha_2))$ does not contain both a literal and its negation, where $\alpha_1(\mathsf{Pre}(\alpha_1))$ is the application of α_1 to the state consisting solely of its own preconditions, and

2. $\mathsf{Abs}(i{-}1, \mathsf{Pre}(\alpha_2)) \neq \mathsf{Abs}(i, \mathsf{Pre}(\alpha_2))$.

By these restrictions α_1 and α_2 are operators that could appear in sequence in an i-th level plan, i.e., the weakest postcondition state of α_1 does not contradict the preconditions of α_2 at level i; and α_2 depends on preconditions at level $i{-}1$ so that there is the possibility of a problem when refining to this level.

If the α_1 and α_2 fail this condition, the test returns success; this pair of operators will not cause a problem during refinement. If the antecedent test is satisfied the operator pair must then satisfy the consequent test, which is as follows. There must exist an sequence of k, $i{-}1$ abstract operators $\beta_1, \ldots, \beta_k$ such that:

1. None of the β_i add or delete literals with criticality higher than $i{-}1$, and

2. The sequence of operators β_i is a solution to the problem

$$\langle \mathsf{Abs}(i{-}1, \alpha_1(\mathsf{Pre}(\alpha_1))), \mathsf{Abs}(i{-}1, \mathsf{Pre}(\alpha_2)) \rangle .$$

If the consequent test succeeds, then we return success, otherwise failure. ■

Intuitively, k-ary necessary connectivity is saying the following: for every pair of operators, if they might be sequenced in a plan at level i (the antecedent test), there must exist a sequence of $i{-}1$ abstract operators β_i that can correctly solve any gap subproblem that might result from the refinement of α_1 and α_2 to the next lower level, $i{-}1$ (the consequent test). Any gap subproblem at abstraction level $i{-}1$ would have to start from a state that must include all the conditions established in the weakest post-condition state of α_1. This weakest state is specified by the application of α_1 to its own preconditions, i.e., $\alpha_1(\mathsf{Pre}(\alpha_1))$, and we simply have to take its $i{-}1$ abstraction. Similarly, the gap subproblem would have to end in a state that achieves the $i{-}1$ preconditions of operators α_2. Condition 1 ensures, furthermore, that none of these added operators alter anything at higher levels of abstraction, so that we satisfy all of the criteria of Theorem 3.3. This argument can easily be formalized to yield:

Theorem 3.6 *k-ary Necessary connectivity is sufficient to guarantee the DRP.*

As we increase the parameter k we increase the complexity of performing this test (the complexity increases exponentially with k as we are searching through the space of operator sequences of length k).

Although we can keep each individual run of the k-ary connectivity test at a reasonable level of complexity, by keeping k small, we still have to test every pair of instantiated operators to determine necessary connectivity. This might be quite a large number, although various techniques can be applied to minimize the number of tests we need. For example, the operator variables are subject to type constraints so not all instantiations are legal; similarly, if two constants have exactly the same properties we do not have to use both in our instantiations. However, we will not delve into such details here. Our main use of necessary connectivity will be to estimate refinement probabilities by subjecting random pairs of operators to the connectivity test, Section 4.2.

3.4 Related Work

With the syntactic condition developed in the previous section, we are in a position to design an algorithm for automatically generating abstraction hierarchies that come close to satisfying the DRP. Such an algorithm, taking as input a description of a domain along with a set of operators, generates a set of criticality assignments that defines an abstraction hierarchy. But before we discuss this algorithm we review some existing theories and algorithms that have been proposed for this purpose. We also discuss the general characteristics that we would like our system to have in order to guarantee the quality of the hierarchies it generates.

3.4.1 ABSTRIPS

In the past, many abstract planning systems have relied on the user to provide an abstraction hierarchy [34, 30, 36]. One of the first systems that semi-automatically generated its own abstraction hierarchies was ABSTRIPS[25]. In addition to the domain specification ABSTRIPS also requires a user-defined partial order on the literals in the domain. It then tries to assign criticality values to the literals that are consistent with the given partial order. For each literal l, ABSTRIPS searches for a short plan that achieves l from a state where all literals before l in the partial order hold. If such a plan is found, then l is considered a detail, and is assigned a low criticality value. Otherwise, l will be assigned a high criticality value. This algorithm can be considered as a method for judging the quality of an abstraction hierarchy: a hierarchy is "good" according to ABSTRIPS, if for every low level literal l, there is a short plan to achieve it from a state where all high level literals are true. PABLO [6], a successor of ABSTRIPS, can also be viewed in this way.

At a first glance, our syntactic necessary connectivity condition is similar to ABSTRIPS since they both depend on finding short plans to achieve low level literals. However, there are some significant differences.

First, our condition specifies that all low-level precondition literals of an operator must be *simultaneously* achievable, whereas ABSTRIPS only requires that each literal be achievable individually. As pointed out by Knoblock [17], in domains where interactions often occur, the existence of an individual plan for each literal does not ensure the existence of a plan for the simultaneous achievement of all of the literals.

Second, our necessary connectivity condition specifies that the low-level plan for achieving the preconditions of an operator should not violate any abstract conditions achieved at higher levels. This is in accordance with our notion of monotonic refinement. In contrast, ABSTRIPS does not specify any restriction on the plan that achieves the low level literals. This means that when ABSTRIPS searches for a refinement of an abstract plan it does not restrict itself to searching for monotonic refinements. This can significantly increase the length of the refinement, and can increase the cost of finding it. Also, during refinement the plan to achieve a low level literal might undo work accomplished at the higher level.

In general, although ABSTRIPS attempts to order the literals in such a way as to make its abstract plans easily refinable, it does not take into account all of the factors that affect refinability. Its techniques are mainly heuristic, and were developed without a formal analysis of the problem.

It can also be noted that ABSTRIPS only partially automates the abstraction process; the quality of its hierarchies depends heavily on the user supplied partial ordered of the literals. The system we have developed, described in the next section, is based on the results of our theoretical analysis, and is completely automatic.

3.4.2 ALPINE

Another successor of ABSTRIPS is Knoblock's ALPINE system [14], which is related to the earlier system LAWALY [27]. ALPINE automatically generates an abstraction hierarchy that has the ordered monotonicity property (OM). The OM property is designed to separate a problem into parts that can be solved first and then held invariant while the rest of the problem is solved. Conditions defining the parts that can be solved first are then placed at a higher level of abstraction than the rest. More precisely, the Ordered Monotonicity Property states that [14, 16]

> *every* refinement of an abstract plan leaves *all* high-level literals unchanged.

This property can be guaranteed by a set of syntactic conditions that relate the operator schemas to the literals in the domain language. These syntactic conditions can then be used in the design of an algorithm that generates abstraction hierarchies possessing the OM property [16]. Experiments reported in [14] demonstrate that in several domains, planning with the abstraction hierarchy generated by ALPINE clearly improves planning efficiency.

Comparing the OM property with the DRP, we note that OM is stronger in some aspects, but weaker in others. In particular, the OM property is stronger than the DRP in that it requires that a refinement leave intact *all* higher-level literals, even those that are not part of

the abstract plan being refined. Furthermore, this restriction must hold for *every* refinement. In many cases, the OM property is so strong that it can only be satisfied by trivial hierarchies, i.e., the hierarchy often collapses to a single level. In contrast, the DRP only requires that for each abstract plan, there exists at least one monotonic refinement; other refinements need not be monotonic. Furthermore, the monotonic refinement can affect higher-level literals, just as long as it does not affect the higher-level literals appearing in the particular abstract plan being refined.

The OM property is also weaker than the DRP, in that it does not guarantee that an abstract plan can be refined. Therefore, the OM property does not eliminate the need to backtrack across abstraction levels. In fact, an OM hierarchy may have very low refinement probabilities. Excessive backtracking causes a significant degradation in performance, since in such cases the hierarchy typically has refinement probabilities that fall in the worst region. This observation is confirmed by our experimental results presented in Section 5. This problem of excessive backtracking in OM hierarchies has also been independently observed by Smith and Peot [28].

In conclusion, the DRP and the OM property address orthogonal issues in the design of good hierarchies. Our theoretical analysis in Section 2 indicates that there are three major criteria relevant to the design of good abstraction hierarchies:

1. Regularity.

2. A lack of subproblem interactions.

3. Minimal backtracking across abstraction levels.[16]

These three criteria can be considered as dimensions along which we can measure the quality of an abstraction hierarchy. While the DRP guarantees the third condition, the OM property contributes to the satisfaction of the first criterion.[17]

Specifically, the regularity property requires that the hierarchy divide the problem into approximately equally sized and relatively short subproblems. If the OM property is not satisfied, then a low-level operator might clobber higher-level conditions. To repair these high-level conditions, more operators may need to be inserted at the current level. This process can cause the lengths of the solutions to the subproblems to increase, which defeats

[16]Our analysis also indicated that if deadend detection is very cheap, then the hierarchy will demonstrate good behavior. This occurs when the refinement probabilities are *very* low, $< 1/B$. However, as we have pointed out, the refinement probabilities are hardly ever this low in practice. Hence, in practice, a good hierarchy will have very high refinement probabilities, in which case there will be minimal backtracking.

[17]Our claim that these two properties are complementary is further justified by the experimental results presented by Knoblock [14]. Knoblock showed that for the Tower of Hanoi domain, an extended STRIPS domain, and a machine-shop scheduling domain, hierarchies that satisfy the OM property enable a dramatic improvement in search efficiency. Our analysis has shown that all three hierarchies tested also satisfy the k-ary necessary connectivity condition (Definition 3.4), thus they all have the DRP as well. This contributes significantly to the efficiency of these hierarchies.

the regularity condition. Although the OM property does not guarantee regularity, it does aid in its achievement.

3.4.3 Near DRP Hierarchies

Since the DRP and the OM property contribute in a complementary manner to the effectiveness of a hierarchy. we would like our hierarchy to possess both properties. However, in practice it may be too strong to require that every abstract plan have a monotonic refinement. In that case, we can relax this requirement and consider instead hierarchies that are close to having the DRP. In particular, we can say that a hierarchy is *near-DRP* if it satisfies the following conditions:

1. Let the refinement probability for level i be p_i, $i = 0, 1, \ldots, n$. Let θ be a user-defined threshold, where $0.5 \leq \theta \leq 1$. We require that $p_i \geq \theta$. That is, the refinement probabilities must be no less than a given threshold.

2. $p_j \geq p_i$ for all $j > i$. That is, the refinement probabilities must be monotonically increasing as we move down the levels of abstraction.

Both of these conditions are motivated by our analysis. As discussed in Section 2 sufficently high refinement probabilities will result in less search, and typically increasing refinement probabilities are more effective, Section 2.5.3.

4 HIGHPOINT

A good hierarchy should have the ability to avoid interactions with higher level achievements, and it should ensure that for every abstract plan a low level refinement exists with high probability. In this section we present an algorithm, HIGHPOINT, that automatically constructs abstraction hierarchies possessing both of these properties.

The algorithm HIGHPOINT is presented in Table 4. Informally, HIGHPOINT functions as follows. It takes as input the set of operators, initial states, and goal states of the domain. First it invokes ALPINE [14] to generate a partially ordered graph that represents a set of OM hierarchies for the given domain. Each node n_i in the graph represents a set of predicates that should be assigned the same criticality in an abstraction hierarchy. An arc from a node n_i to n_j denotes that plans for achieving subgoals whose predicates are in the set n_j will not affect any predicates in the set n_i. That is, if we have an abstract plan involving the achievement of literals whose predicates are in n_i, then we can refine that plan to achieve literals in n_j without affecting any of the n_i literals. The graph has the property that every total order of the nodes that extends the partial order, represents a hierarchy with the Ordered Monotonic Property. Next, steps 2-5 of the algorithm assigns a estimated refinement probability to every pair of nodes n_i and n_j such that $n_i > n_j$ is allowed by the partial order defined by the graph. Finally, step 6 processes the nodes in the graph using the additional information

Input: A set of operators O, initial states *Inits* and goal states *Goals*.
Output: A criticality assignment to the predicates, such that the abstraction hierarchy satisfies the OM property and is close to being near-DRP.

Algorithm HIGHPOINT(**O,Inits,Goals**)
1. $graph$:= `ALPINE(O, Inits, Goals)`;
2. **for** every pair of nodes n_i and n_j in $graph$, such that
3. there is no path from n_j to n_i in $graph$, **do**
4. `prob`(n_i, n_j) := `Find-Probability`(n_i, n_j, O)
5. **endfor**
6. $graph$:= `Collapse-Nodes`$(graph, prob)$;
7. $hierarchy$:= `Augmented-Top-Sort`$(graph, prob)$;
8. **return**(hierarchy);

Table 6: The HIGHPOINT algorithm for creating a hierarchy.

provided by the estimated refinement probabilities. The procedure `Augmented-Top-Sort` returns a criticality assignment to each predicate, such that the resultant hierarchy has the OM proprety and is close to being near-DRP. We cannot guarantee near-DRP for two reasons. First, we are only able to obtain estimates of the refinement probabilities, so the true refinement probabilities might fail to satisfy the near-DRP property. Second, the OM property forces the placement of some literals above others. Since it does not consider refinement probabilities, the placement of literals forced by OM might result in the violation of near-DRP. HIGHPOINT attempts to find the best hierarchy, with respect to being near-DRP, among those hierarchies that satisfy the OM property. The hierarchy is then returned as the output of the algorithm.

Below, we explain in detail each of the major components of HIGHPOINT, with the help of an extension of the simple robot domain used in Section 3. As before there is a robot and a number of connected rooms between which the robot can move. Between any two rooms there may be a door, which can be open or closed. In addition, there are also a number of boxes, which the robot can either pull or carry from one location to another. Figure 7 shows one configuration of the domain. The operators in this domain are presented in Table 7. There is one additional operator not shown in the table, `carry-thru-door(b,d,r2,r2)`; this operator is identical to `pull-thru-door` except that the box, `b`, must be `Loaded` instead of `Attached`.

Our representation for this domain includes the following predicates: `BoxInroom(b,r)` representing that box `b` is in room `r`; `Attached(b)` representing that box `b` is attached to the robot; `Loaded(b)` representing that box `b` is loaded onto the robot, `Open(d)` representing that door `d` is open. In addition, there are also a number of type predicates (e.g.,

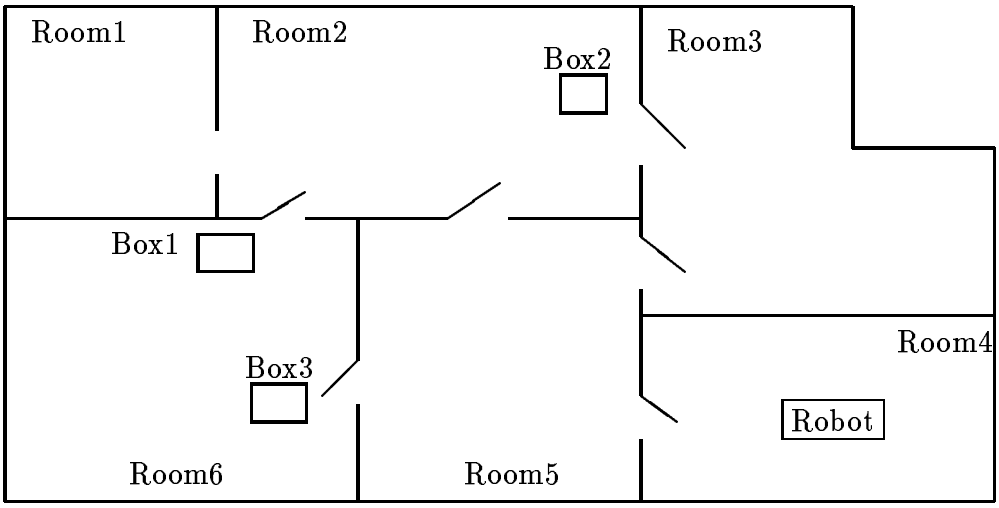

Figure 7: Robot-Box Planning Domain.

`IsDoor(Door12), Openable(Door12), Connects(Door12, Room1, Room2)`).

4.1 ALPINE

ALPINE is an abstraction hierarchy generation algorithm designed and implemented by Knoblock [14]. Given the operator definitions for a given domain, ALPINE constructs a partially ordered graph of the literals. Each node in the graph denotes a set of literals that are to be assigned the same criticality value in the final hierarchy. If a node n_j precedes n_i in the graph, i.e., if there is a path from n_j to n_i in the graph, then we cannot place n_i above n_j in the final hierarchy; the criticality value of n_j must be greater than or equal to the criticality value of n_i. The algorithm ensures that every total order supported by the graph will yield a hierarchy that has the ordered-monotonic property, whereby every refinement of an abstract plan leaves all the higher level literals unchanged.

The core of the ALPINE algorithm is the following restriction:

Definition 4.1 [Ordered Restriction (Knoblock)] Let O be the set of operators in a domain. Let P_α be the preconditions of α that can be either added or deleted by some operator. Then a criticality assignment satisfies the ordered restriction if $\forall \alpha \in O$, $\forall p \in P_\alpha$, and $\forall e_1, e_2 \in \mathsf{Add}(\alpha)$,

(1) $crit(e_1) = crit(e_2)$, and
(2) $crit(e_1) \geq crit(p)$.

That is, all the adds of an operator are required to have the same criticality, and that criticality must be at least as great as the operator's changeable preconditions. Knoblock has

Preconditions	*Adds*	*Deletes*
`pull-thru-door(b,d,r1,r2)`		
`IsDoor(d), IsBox(b), IsRoom(r1), IsRoom(r2) Connects(d,r1,r2) Attached(b) BoxInroom(b,r1) Open(d)`	`BoxInroom(b,r2)`	`BoxInroom(b,r1)`
`attach-box(b)`		
`IsBox(b), ¬Attached(b)`	`Attached(b)`	`¬Attached(b)`
`load-box(b)`		
`IsBox(b), ¬Loaded(b)`	`Loaded(b)`	`¬Loaded(b)`
`open(d)`		
`IsDoor(d), Openable(d), ¬Open(d)`	`Open(d)`	`¬Open(d)`

Table 7: Operators for the robot-box domain.

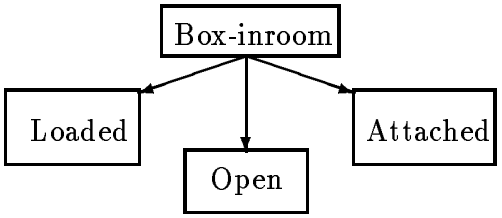

Figure 8: Robot-box domain graph generated by ALPINE.

shown that if the assignment of criticality values satisfies this restriction then the hierarchy will satisfy the OM property.

The ALPINE algorithm implements this restriction, generating a graph representing partially ordered collections of literals. The ALPINE system then uses this graph to compute a particular total order and resulting hierarchy, using a collection of heuristics to pick the total ordering. The algorithm depends only on the operator definitions, not on the goals and initial situations. However, Knoblock has shown how to modify this algorithm so that it can generate problem specific hierarchies that take into account a particular goal and initial state. This often results in a finer grained hierarchy [14].

We illustrate the algorithm via the robot-box example. When applied to the operators in this domain, ALPINE generates the graph shown in Figure 8. It is clear that there are

six possible total orders that could result from this graph, corresponding to the predicate `BoxInroom` followed by permutations of `Open, Attached, Loaded`.[18] Each total order corresponds to a different hierarchy. For example, for the total order,

`BoxInroom, Attached, Loaded, Open`,

the corresponding hierarchy is shown in Table 8.

Criticality	Predicates
4	`IsDoor, Openable, Connects, IsBox, IsRoom`
3	`BoxInroom`
2	`Attached`
1	`Loaded`
0	`Open`

Table 8: Robot-box domain hierarchy generated by ALPINE.

4.2 Refinement Probabilities

If there is a path from n_j to n_i in the ALPINE graph, we cannot place the literals in n_i at a higher level of abstraction than the literals in n_j, without violating the OM property. However, for all pairs n_i and n_j that do not have this constraint we have a choice; we can, if we wish, place n_i above n_j in the final hierarchy. The next step of the HIGHPOINT algorithm is to determine the merit of such a placement. It does this by estimating the refinement probability that would exist between the levels n_i and n_j if n_i was in fact placed above n_j. If node n_i was placed before n_j, then all of the predicates in n_i would be placed at a higher level of abstraction than the predicates in n_j.

For all pairs of nodes n_i and n_j such that it is possible to place n_i above n_j, HIGHPOINT calls `Find-Probability`(n_i, n_j) to estimate the probability of refinement going from predicates in n_i to predicates in n_j. Since we do not know the final total ordering of the nodes, `Find-Probability` generates its estimate using two simplifications. First, it assumes that n_i is directly above n_j in the abstraction ordering, i.e., that the predicates in n_i are at level i and the predicates in n_j are at level $i-1$, for some i. Second, it ignores the effect of any level that might lie above n_i. In the final hierarchy the true refinement probability of moving a solution from the level of n_i to the level of n_j will depend on both of these factors, i.e., the intervening levels between n_i and n_j as well as the levels above n_i. We use these simplifications to keep HIGHPOINT computationally tractable. In particular, as a result of these

[18] The type predicates, like `Openable`, are not shown on this graph. Since these predicates are not affected by any of the operators they are always placed at the highest level by ALPINE.

simplifications `Find-Probability` only has to examine $O(n^2)$ pairs of nodes in a graph containing n nodes. Of course, these simplifications also mean that `Find-Probability` provides only a rough estimate of the true refinement probabilities. Nevertheless, we have found that in practice (see Section 5) the estimates provided by `Find-Probability` are quite effective in guiding the construction of a good hierarchy, probably because the final performance of the hierarchy shows some degree of tolerance to minor variations in the node ordering.

In its operation `Find-Probability` first locates the sets of operators O_i and O_j that achieve literals in n_i and n_j respectively. It generates its estimate of the refinement probability from n_i to n_j by determining how often pairs of operators in O_i pass the connectivity test, Definition 3.5, where we consider the predicates in n_i to be at level i and the predicates in n_j to be at level $i-1$.

Consider a plan at level n_i. Such a plan will typically have to achieve literals in n_i, hence it will contain operators from O_i. To refine this plan to the level of n_j, which we are assuming is right below, we will have to solve the gaps generated by pairs of operators from O_i using operators from O_j. To estimate how often these gaps can be solved we check to see how often pairs of operators in O_i pass the connectivity test, as if they do their gap problems can always be solved. In particular, according to Definition 3.5 this means testing how often the planning problem $\langle \mathsf{Abs}(i-1, \alpha_1(\mathsf{Pre}(\alpha_1))), \mathsf{Abs}(i-1, \mathsf{Pre}(\alpha_2))\rangle$ can be solved using operators that affect only literals at level $i-1$ and below, where α_1 and α_2 are members of O_i. Since we are ignoring all other nodes, we ignore all preconditions of α_1 and α_2 that are outside of n_i and n_j, except for type preconditions which are always at the top of the hierarchy. Furthermore, since we are considering level i to be the predicates in n_i and level $i-1$ to be the predicates in n_j we can compute the $i-1$ abstractions, i.e., $\mathsf{Abs}(i-1, \ldots)$, by simply adding any predicates from n_j back into the sets being abstracted. Finally, we can try to solve these problems using any operators in O_j which do not affect predicates in n_i, effectively ignoring the other predicates (nodes) affected by these operators.

For operators in O_i, `Generate-Random-Problems` chooses at random a pair of instantiated operators that passes the antecedent condition of the connectivity test; this condition weeds out the operator pairs that cannot appear in sequence in an n_i level abstract plan and it also eliminates those operator pairs that have no n_j level preconditions. It then computes the problem shown above which is specified by the consequent condition of the connectivity test. A whole collection of such pairs of operators are generated and their resulting problems are returned as the set "random-probs". `Find-Probability` then calls a planner ABTWEAK, [36], which tries to solve these problems. In line 5 of the `Find-Probability` algorithm, the ABTWEAK parameter O_j/O_i specifies the operators that can be used to solve the problem. This is the set of operators O_j with those operators that affect literals contained in n_i removed, i.e., all operators also in O_i have been removed using a set difference operation. Each problem is solved under a fixed solution length bound, this implements the "k-ary" limit part of the connectivity test. `Find-Probability` accumulates a count of the number of times a solution is found.

The frequency of success serves as an estimate of the refinement probability, as we know

Input: Operators O, predicate sets n_i, n_j.
Output: A refinement probability value.

Procedure Find-Probability(n_i, n_j, O)
1. O_i := Find-Operators(O, n_i);
2. O_j := Find-Operators(O, n_j);
3. random-probs := `Generate-Random-Problems`(O_i, n_i, n_j);
4. **for** every random problem $\langle initial, goal \rangle$ in random-probs **do**
5. **if** ABTWEAK$(initial, goal, O_j/O_i,$ solution-length-bound$) = Success$ **then**
6. success-count := success-count + 1
7. **endif**
8. **endfor**
9. **if** |random-probs| = 0 **then**
10. prob := 1
11. **else** prob := success-count/|random-probs|
12. **endif**
13. **return**(prob);

Table 9: Algorithm Find-Probability.

that if two operators pass the test then the gap subproblem they generate during refinement can be solved. If `Generate-Random-Problems` fails to find any random problems, then it has failed to find any operators with level n_j preconditions among the set of operators that can be sequenced. Hence, it is unlikely that any difficulties will be encountered during refinement, and we estimate the refinement probabilities as being 1.

We now illustrate the procedure using our robot domain. Let n_i be {`BoxInroom`}, and n_j be {`Open`}. The operator sets corresponding to the nodes are O_i = {`pull-thru-door`, `carry-thru-door`}, and O_j = {`open`}. Both of the operators in O_i can be sequenced in any manner as long as the room the first operator takes us into is identical to the room the second operator moves us out of; this constraint is enforced at the n_i level via the `BoxInroom` precondition of these operators. Additionally, we ignore the precondition of `Attached` (`Loaded` in the case of `carry-thru-door`) as this precondition is in node other than n_i or n_j and it is not a type predicate.

So of those operators that can be sequenced, i.e., that pass the antecedent test, we will generate random problems that are instantiations of the following template arising from the

connectivity test:

$$\left\langle \begin{array}{ll} \texttt{IsDoor(d1),IsBox(b)} & \texttt{IsDoor(d2),IsBox(b)} \\ \texttt{IsRoom(r1),IsRoom(r2)} & \texttt{IsRoom(r2),IsRoom(r3)} \\ \texttt{Connects(d1,r1,r2)} & , \texttt{Connects(d2,r2,r3)} \\ \texttt{BoxInroom(b,r2)} & \texttt{BoxInroom(b,r2)} \\ \texttt{Open(d1)} & \texttt{Open(d2)} \end{array} \right\rangle .$$

That is, the problem is to achieve the preconditions of the second operator in the state that results from applying the first operator to its precondition set, as specified by the connectivity test, for various instantiations of the operators. It is easy to see that this problem reduces to achieving `Open(d2)` for different instantiations of `d2`. A door can be opened with the operator `open(d)` without affecting any higher level literals, as long as it is `Openable`. So we see that the number of solvable random problems will correspond approximately to the proportion of doors that are `Openable` in the domain. Hence, the refinement probability returned by `Find-Probability` will depend on the probability of a door being openable. This agrees with intuition. If we place `Open` at a lower level of abstraction the abstract level will be free to develop a plan ignoring the status of the connecting doors. If most doors can be opened this will generally not be a problem, but if most doors cannot be opened, most of the routes chosen at the abstract level will fail. That is, most of the abstract plans will not be refinable.

The result of running `Find-Probability` on all eligible pairs of nodes is a matrix of estimated refinement probabilities. In one of our tests in which half of the doors were openable we obtained the matrix of values shown in Table 10.

Predicates	`BoxInroom`	`Open`	`Attached`	`Loaded`
`BoxInroom`	0	0.5	1	1
`Open`	0	0	1	1
`Attached`	0	1	0	1
`Loaded`	0	1	1	0

Table 10: A matrix of refinement probabilities.

4.3 Collapsing Nodes with Low Refinement Probabilities

Using the refinement probabilities the procedure `Collapsing-Nodes` processes the ALPINE-graph. In particular, based on the threshold θ used in the near-DRP condition, Section 3.4.3, it decides if two nodes should be collapsed into one.[19]

[19]Given our analytic results which indicate that asymptotically more levels are better, it may seem strange that we would wish to collapse levels. However, the collapsing of levels is supported by our exact results, as

If the refinement probability for both orderings of these nodes is below θ, then no hierarchy in which these nodes are on separate levels will be close to being near-DRP. That is, if $\mathsf{prob}(n_i, n_j) < \theta$ and $\mathsf{prob}(n_j, n_i) < \theta$, as found by the `Find-Probability` routine, then we collapse these two nodes into one $n_{ij} = (n_i \cup n_j)$. This means that when we assign criticalities using a total order produced from this graph the literals in n_i and n_j will be given identical criticalities. Therefore, to ensure we satisfy the constraints imposed by the original partial order we must also collapse all nodes that lie on any path between n_i and n_j into the new node n_{ij}. To collapse the nodes the graph is modified by substituting all collapsed nodes by the new node n_{ij} and then all in-edges of the collapsed nodes become in-edges of the new node and similarly for the out-edges. The refinement probabilities to and from all the remaining nodes must be recomputed for the new node. To avoid doing this computation, we choose instead to use the average of the original probabilities as estimates for the new ones, so that for every other node n, we let

$$\mathsf{prob}(n, n_{ij}) = \text{Average}\{\mathsf{prob}(n, n_i) \mid \ n_i \text{ has been collapsed into } n_{ij}\},$$

and similarly for $\mathsf{prob}(n_{ij}, n)$ for nodes n connected via out-edges. The collapsing process continues until no more nodes can be further collapsed.

In robot and box domain the two nodes containing `BoxInroom` and `Open` will be collapsed to one level whenever the threshold θ is greater than the proportion of openable doors. In this case the refinement probability from `BoxInroom` to `Open` falls below θ, and the opposite ordering of these nodes is impossible.

4.4 Augmented Topological Sort of Abstraction Graph

After collapsing nodes with low refinement probabilities, the procedure `Augmented-Top-Sort` computes a total order of the nodes in the resulting graph, using both the partial order relation and refinement probabilities as guides to compute the order. The procedure is a simple modification of a standard topological sort algorithm (see, e.g., [1]). The standard topological sort works by first placing all nodes with no in-edges in a queue. Nodes are removed from this queue, and when a node is removed from the queue all of its out-edges are removed from the graph. If any of its successors now have no in-edges they are added to the end of the queue. The topological order is the order in which the nodes are removed from the queue. Our augmented version simply orders the collection of nodes that are placed on the queue during any one step. That is, at each stage we add those nodes with no in-edges to the queue, but we add them to the queue so that the sequence of `Find-Probability` estimates between these nodes is ascending. For example, if at some state we are to add the nodes n_1, n_2, and n_3 to the queue we would choose, e.g., the ordering n_3, n_1, n_2 if $\mathsf{prob}(n_3, n_1) \leq \mathsf{prob}(n_1, n_2)$.

---

we will discuss in Section 5.4.

To illustrate the procedure, consider again the simple robot example. The graph generated by ALPINE now has associated refinement probabilities, shown in Table 10. The augmented topological sort algorithm will place `Open` right below the `BoxInroom` level, resulting in the hierarchy shown in Table 11.

Criticality	Predicates
4	`IsDoor, Openable`
3	`BoxInroom`
2	`Open`
1	`Attached`
0	`Loaded`

Table 11: Robot box domain hierarchy generated by HIGHPOINT.

5 Experimental Results

In the last section, we have described the HIGHPOINT algorithm for generating abstraction hierarchies which augments the ALPINE algorithm by taking into account the refinement probabilities between abstraction levels. To compare their performance, we have conducted a set of experiments in three different experimental domains, with a time bound of 30 CPU minutes. For the refinement probability estimate computation, ABTWEAK is called with a solution length limit of five steps. The common features of the domains are that we are able to change the refinement probability at each level individually, by changing the mixture of objects with different properties. For example, in the robot box domain we can change the refinement probability between `BoxInroom` and `Open` by changing the proportion of openable doors. All systems and domains were implemented in Allegro Common LISP on a Sun4/Sparc Station. Below, we describe our test results.

5.1 Box Domain

Our first test was run on the robot-box domain used in our previous examples. In our realization of this domain we place two doors between every pair of adjacent rooms. Each door may or may not be openable. The robot's task is to move boxes between the room, by either carrying or pulling them.

For domain instances where every door is openable, both ALPINE and HIGHPOINT generate the same abstraction hierarchy, shown in Table 8. Therefore the costs of actual problem-solving using the ALPINE and HIGHPOINT hierarchies are the same. The only difference is that HIGHPOINT requires extra time to determine the refinement probabilities. For the

Solution Length	HIGHPOINT(seconds)	ALPINE (seconds)
0	0.083	0.083
3	0.55	0.484
6	4.15	6.55
9	11.167	17.234
12	97.017	419.017
15	229.3	903.25

Table 12: CPU time comparison between HIGHPOINT and ALPINE in the box domain.

domain where every door is openable, HIGHPOINT takes 27 CPU seconds to generate the hierarchy, while ALPINE took only 0.05 CPU seconds. However, to solve each problem, both systems take 4-5 seconds on average. Thus, as the number of problems grows, HIGHPOINT's initial cost is amortized away.

We then changed the domain so that not all the doors were openable. In this case not every plan that ignores the status of the doors will be refinable, and HIGHPOINT will place `Open` higher up in the hierarchy, so that this condition can be tested earlier on, before more resources are allocated to solving the `Loaded` and `Attached` preconditions. In our test we set more than 50% of the doors to be openable, so HIGHPOINT did not collapse any levels. The hierarchy it generated is shown in Table 11. HIGHPOINT takes 75 seconds to generate this hierarchy, after checking 20 randomly generated problems. In contrast, ALPINE is not able to alter its hierarchy in response to a change in the number of openable doors, so it generates the same hierarchy as before.

To compare the qualities of the hierarchies generated by HIGHPOINT and ALPINE, we first ran ABTWEAK on six problems of varying sizes. ABTWEAK solves each problem by first using the hierarchy generated by HIGHPOINT, and then by using the one by ALPINE. The CPU time costs for both hierarchies, which do not include the time for generating the hierarchies, are shown in table 12. The table demonstrates that as the planning problems get more complex, HIGHPOINT is increasingly more efficient than ALPINE. When the initial costs for generating the hierarchies is taken into account, HIGHPOINT might require a number of problems of small size before it can recover the cost of its more expensive hierarchy generation algorithm. However, for problems of large size, HIGHPOINT outperforms ALPINE by such a large margin that it can recover its intial cost of hierarchy generation in a single problem, yielding an immediate improvement in net problem solving costs.

As a further test we ran 42 test problems of equal length using the hierarchies generated by HIGHPOINT and ALPINE, respectively. Figure 9 compares the accumulated CPU time of both systems over the same 42 problems. The time required by the algorithms to generate their abstraction hierarchies is also included in the values plotted. It is clear from the

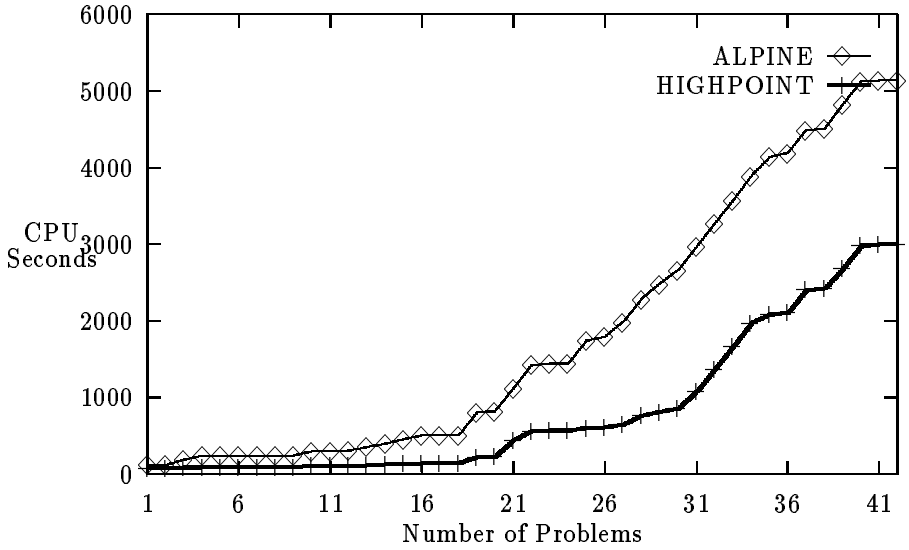

Figure 9: Robot box domain tests.

figure that the HIGHPOINT hierarchy is soon able to recover from its initial cost, and after 2 problems of this size the net cost was lower than problem solving with the ALPINE hierarchy. In this domain the HIGHPOINT hierarchy is able to solve problems of this size almost twice as fast as the ALPINE hierarchy, and as the previous table has shown, as the problems become longer so does HIGHPOINT's factor of improvement.

5.2 Computer Hardware Domain

In the second domain we have a simple computer configuration. The task is to print a file. The file can be transferred to any of a number of different computers and then sent to a printer, where each computer can send the file to any printer. The computers and printers may or may not be turned on, and they must both be turned on for the file to be printed. If the power for one of the devices is not on it can be plugged in and then turned on. Once it is plugged in, the device must be functional for it to be powered up. Similarly, for it to be plugged in the device must be located with the reach of a power outlet. Table 13 shows the operators for this domain.

For this domain, ALPINE generates the graph shown in Figure 10. If every computer is functional and can be connected to an electric outlet, then every total order of the graph results in an abstraction hierarchy with the DRP. In this case ALPINE generates the hierarchy shown in Table 14. This takes 0.02 seconds. HIGHPOINT takes 40.7 seconds to check that no changes need to be made as all the refinement probabilities are estimated to be 1.

Preconditions	*Adds*	*Deletes*
`print(f,c,p)`		
`IsComputer(c)`, `IsPrinter(p)`, `PowerOn(c)`, `PowerOn(p)`, `Loaded(f,c)`	`Printed(f)`	
`turn-on(d)`		
`PluggedIn(d)`, `Functional(d)`	`PowerOn(d)`	
`plug-in(d,o)`		
`is-outlet(o)` `CableCanReach(d,o)`	`plugged-in(d)`	
`transfer-file(f,c)`		
`IsComputer(c)` `PowerOn(c)`	`Loaded(f,c)`	

Table 13: The operators of the computer domain.

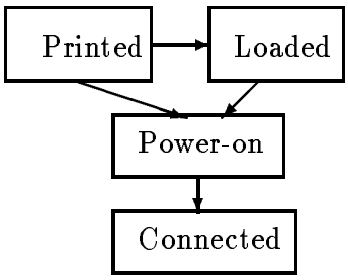

Figure 10: Computer domain graph generated by ALPINE.

Criticality	Predicates
4	`IsComputer, IsPrinter, Functional, CableCanReach`
3	`Printed`
2	`Loaded`
1	`PowerOn`
0	`PluggedIn`

Table 14: Computer domain hierarchy generated by ALPINE.

However, suppose that not all of the cable runs have been completed, so that most of the devices cannot yet be connected to an power outlet. Then the refinement probability will be very low between levels 1 and 0 in the hierarchy of Table 14. Of course ALPINE cannot alter the hierarchy it generates, but HIGHPOINT recognizes the problem and collapses these two levels. The resultant hierarchy, generated by HIGHPOINT using a threshold value of 0.5, is shown in Table 15. To generate this hierarchy, it took HIGHPOINT 64.5 CPU seconds.

Criticality	Predicates
3	`IsComputer, IsPrinter, Functional, CableCanReach`
2	`Printed`
1	`Loaded`
0	`PowerOn, PluggedIn`

Table 15: Computer domain hierarchy generated by HIGHPOINT.

We first compared the two hierarchies on five planning problems of increasing solution lengths. All test problems involve printing several files by choosing a functional computer and printer. The result of the comparison is shown in table 16, where each datum represents the amount of CPU time required to solve the problem using each hierarchy. The time required for generating the hierarchy is not included in the table. A CPU time limit of 30 minutes was imposed on both sets of tests, and as can be seen from the table, HIGHPOINT is able to solve longer problems within the time limit. Also, the data again demonstrates that HIGHPOINT's advantage over ALPINE increases as the problem becomes more complex.

Figure 11 further compares the performance of the two hierarchies. A total of 45 sample problems, all requiring solutions of length six, were tested in a domain where only a small percentage of the computers could be connected to a power outlet. ALPINE and HIGHPOINT

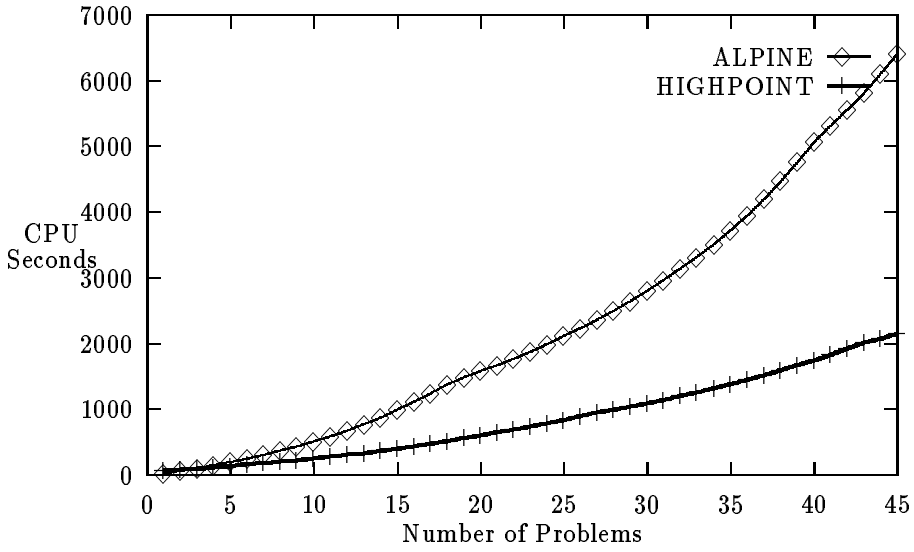

Figure 11: Computer domain tests.

Solution Length	HIGHPOINT (seconds)	ALPINE (seconds)
6	1.6	3.2
8	11.6	57.6
10	66.4	1800.0
12	365.6	
14	1800.0	

Table 16: CPU time comparison between HIGHPOINT and ALPINE in the computer hardware domain.

Preconditions	*Adds*	*Deletes*
`shape(x)`		
`Object(x)`	`Shaped(x)`	`Drilled(x)` `Painted(x)`
`drill(x)`		
`Object(x)`	`Drilled(x)`	`Painted(x)`
`paint(x)`		
`Object(x), Steel(x)`	`Painted(x)`	

Table 17: The operators of the manufacturing domain.

were applied to generate their hierarchies, and then ABTWEAK was used to solve the 45 sample problems using the different hierarchies. The graph shows accumulated CPU time as a performance measure, including the initial time for hierarchy generation. Again, the initial cost of HIGHPOINT is quickly recovered as its hierarchy solved problems of this size more than 3 times faster than the ALPINE hierarchy.

5.3 A Manufacturing Domain

In our last test we used a domain suggested by Smith and Peot [28]. The task is to transform a piece of stock into a desired form, by drilling, shaping and painting. The operators of the domain are listed in Table 17. The goal of the domain is to solve the planning problem

$$\left\langle \texttt{Object(s)} \;,\; \begin{array}{l}\texttt{Shaped(x)}\\ \texttt{Drilled(x)}\\ \texttt{Painted(x)}\end{array} \right\rangle,$$

for some object x in the domain.

Criticality	Predicates
3	`Object, Steel`
2	`Shaped`
1	`Drilled`
0	`Painted`

Table 18: Transportation domain hierarchy generated by ALPINE.

Criticality	Predicates
1	`Object, Steel`
0	`Shaped, Drilled, Painted`

Table 19: Manufacturing domain hierarchy generated by HIGHPOINT.

Stocks	HIGHPOINT (seconds)	ALPINE (seconds)
100	25.2	34.3
110	28.9	39.3
120	33.5	44.0
130	37.0	65.0
140	40.0	54.8
150	46.7	60.5
160	67.3	65.6
170	55.2	72.2
180	60.3	78.6
190	65.1	99.5

Table 20: CPU time comparison between HIGHPOINT and ALPINE in the manufacturing domain.

Smith and Peot used this example to illustrate a shortcoming of ALPINE. Suppose that in the initial state there are 100 pieces of stock and only one of them is made of steel. ALPINE would not recognize this specific situation and generate a hierarchy as shown in Table 18. With this hierarchy an abstract planner will first satisfy the goals involving predicates `Shaped` and `Drilled`, and during this process an object will be arbitrarily selected to substitute for the variable x. However, when we try to achieve the `Painted` goal, the `Steel(x)` precondition of operator `Paint(x)` will most likely be unsatisfied, because only one out of 100 pieces of stocks is made of steel. Hence, backtracking will occur up through the hierarchy for the selection of another object, and many partial solutions will have to be abandoned.

In contrast, HIGHPOINT's hierarchy does not suffer from this problem because in the situation where steel objects are rare the refinement probabilities from any higher level to the `Painted` level will be low. The HIGHPOINT algorithm, will collapse the three predicates `Shaped, Drilled` and `Painted` to a single level, as this is the only way the OM property can be maintained. In the case where there are 100 pieces of stock with only a single steel one, it took HIGHPOINT 16.7 seconds to generate the hierarchy shown in Table 19.

The empirical comparison in CPU time of HIGHPOINT and ALPINE is shown in Table 20,

where the CPU times do not include the initialization costs of generating the hierarchies. Each row in the table corresponds to different quantities of stock in the initial state, where in each case there is only piece made of steel. It can be seen from the table that HIGHPOINT outperforms ALPINE as the quantity of stock grows larger.

Here we would also note that the solution offered by HIGHPOINT is different from that suggested by Smith and Peot. Instead of collapsing the part of the hierarchy where the refinement probability is low, they suggested that the abstraction level corresponding to `Painted` should be placed at the highest level. The reason for doing this is that there are far fewer solutions for the `Painted` condition than the others. While ordering the predicates in this manner makes sense for this specific example, in general it is difficult to obtain a count for the number of solutions required to implement this strategy. In addition, by violating the order suggested by ALPINE, one runs a risk of having the abstract plans interfered with by their low level refinements. The result could be additional planning required during refinement to repair violated higher level achievements.

5.4 Discussion of the Experimental Results

We have shown that HIGHPOINT is able to offer a significant improvement over ALPINE, which serves to demonstrate the validity of our approach, and the importance of the DRP property. In particular, HIGHPOINT has the following advantages over ALPINE:

1. As demonstrated by tests in the robot-box domain, when ALPINE generates a partially ordered graph of predicate sets, for hierarchy generation, its selection of a total order does not depend on refinement probabilities. Thus, it is possible that it may generate a hierarchy in which the refinement probabilities are poorly configured, e.g., where they are not increasing as we move down the hierarchy. In contrast, HIGHPOINT is able to select a more intelligent total order by examining these probabilities.

2. When the refinement probability is low, HIGHPOINT recognizes the need to collapse two or more levels. This property is displayed in both the computer hardware domain and in the transportation domain.

3. Although HIGHPOINT requires more time to generate its hierarchy it is clear that over a number of problem solving instances, and for problems with lengthy solutions, this cost is quickly paid off.

The first criteria that HIGHPOINT uses, that of high and non-decreasing refinement probabilities is directly supported by our analysis. However, its behavior of collapsing levels seems on the surface to contradict our results.

Our analysis predicts that asymptotically we are better off with more levels, as complexity is reduced as the number of levels increases. However, the asymptotic results ignore the

constant factors. One important feature of our analysis is that we are able to give exact results. In particular, our results provide the more accurate expression

$$\mathsf{GoodTreeWork}(n) = O(b^k k^{n-1}) + O(b^k k^{n-1}(pB)^{n-2}\gamma(p)) \qquad 1/B < p < 1, \qquad (17)$$

(Equation 12). Hence, we see that there is a constant overhead factor $O((pB)^{n-2})$ once the DRP fails. This constant factor is exponential in the number of levels. For practical problems B can be quite large, as the number of alternate refinements of an abstract solution will often be large. This means that in reducing the number of levels we can often improve performance for the size of problems that we are dealing with. That is, in practice we are still not able to solve problems so large that the full force of the asymptotic results come into play (where the factor $O((pB)^{n-2})$ fades in significance). This is why exact complexity results, as we have given, are much more useful than asymptotic results. However, analysis is not sufficient. It was our experimental results which indicated that the constant factors were still dominant for the size of problems we were dealing with.

HIGHPOINT can easily be improved, but since this was not the focus of our work we were content with a simple implementation. Two obvious ways HIGHPOINT can be improved is if it used our analytic forms directly to estimate the amount of work to solve a problem on candidate hierarchies. This would involve a more through search through the space of candidate hierarchies; for the searched candidates the analytical forms predicting their behavior could be evaluated using the information gathered by `Find-Probability`. This would give a more accurate evaluation of the hierarchy's worth than the approximations we used. Another improvement would be to extend HIGHPOINT to handle individual planning problems rather than try to construct a single hierarchy for all problems in the domain. Knoblock has demonstrated that problem-specific hierarchies can often display superior performance [14].

Another source of improvement is to obtain a better understanding of the threshold value θ. In this work, the value is set to 0.5 for near-DRP hierarchies. However, there might be other better values depending on the domain of planning. It might be worthwhile to examine the impact of the threshold values on the quality of the hierarchies.

6 Conclusions

In this paper we have presented an analytic study of the benefits of abstraction with and without the DRP, and have used the analysis in a practical manner to design an improved hierarchy generation algorithm. Our experimental tests have demonstrated the validity of the approach.

Our analytical results indicate that the main benefit in abstraction is the ability of the hierarchy to break up the problem into smaller and independent subproblems. If the hierarchy is successful in doing this then *asymptotically* abstraction will always be a gain, even if we have to search through many alternate abstract plans.

However, except in the unusual case where we can generate a variable number of levels in the hierarchy, hierarchical problem solving is not a magic solution. It decreases the size of the exponent, but the complexity of problem solving remains exponential. This is why our practical experiments do not always show the effects of the asymptotics. Since the underlying complexity is exponential it is not always possible to solve large enough problems for the asymptotic benefits to manifest themselves. This is why abstraction does not always work in practice, and this is why collapsing levels of the hierarchy, as done by HIGHPOINT, is of practical benefit. Furthermore, this phenomia underscores the importance of providing exact complexity results whenever possible. In particular, the exact complexity results we have provided are able to explain the benefit of collapsing levels of the hierarchy: there is a constant factor that is exponential in the number of abstract levels, and this constant factor can often dominate in problems of practical size.

In conclusion, the main contributions of this work are as follows:

1. A model of abstraction has been developed, and based on that model we have provided an analysis of a number of different features of hierarchical problem-solving under different conditions of refinability.

2. A syntactic condition has been provided for testing if an ABSTRIPS-hierarchy possesses the DRP. The condition can also be used to estimate refinement probabilities.

3. An algorithm has been provided that automatically constructs an abstraction hierarchy for a given domain. The hierarchies constructed have both the ordered monotonicity property and the near-DRP.

4. Empirical tests have been presented that support our analytical results, and confirm the utility of our algorithm.

Acknowledgments. Thanks to Craig Knoblock and Josh Tenenberg for helpful comments and questions. During a presentation of this work Tom Bylander asked if abstraction aids in detecting the nonexistence of a solution, this inspired Section 2.5.2. Finally, we wish to thank the anonymous referees for very useful comments.

A GoodTreeWork(i)—The simplification

In section 2.4.4 we presented a simplified form for GoodTreeWork(i) which we derive here. Starting with the form:

$$\begin{aligned}\mathsf{GoodTreeWork}(i) \;=\; & \mathsf{NodeWork}(i) \\ & + \sum_{m=1}^{B} \frac{b(m,B,1-F(i-1))}{1-b(0,B,1-F(i-1))} \Big(\big(\frac{B-m}{m+1} \big) \mathsf{BadTreeWork}(i-1) \Big) \\ & + \mathsf{GoodTreeWork}(i-1).\end{aligned}$$

We can take $\mathsf{BadTreeWork}(i-1)$ outside of the summation to obtain

$$\begin{aligned}\mathsf{GoodTreeWork}(i) &= \mathsf{BadTreeWork}(i-1)\sum_{m=1}^{B}\frac{b(m,B,1-F(i-1))}{1-b(0,B,1-F(i-1))}\Big(\frac{B-m}{m+1}\Big)\\ &\quad+\mathsf{NodeWork}(i)+\mathsf{GoodTreeWork}(i-1).\end{aligned}$$

Since $b(m,B,1-F(i-1))=\binom{B}{m}[1-F(i-1)]^m F(i-1)^{B-m}$, we can simplify the remaining sum.

$$\begin{aligned}&\sum_{m=1}^{B}\frac{b(m,B,1-F(i-1))}{1-b(0,B,1-F(i-1))}\Big(\frac{B-m}{m+1}\Big)\\ &= \frac{1}{1-b(0,B,1-F(i-1))}\sum_{m=1}^{B}\frac{B!}{m!(B-m)!}\Big(\frac{B-m}{m+1}\Big)(1-F(i-1))^m F(i-1)^{B-m}\\ &= \frac{F(i-1)}{(1-b(0,B,1-F(i-1)))(1-F(i-1))}\\ &\qquad\sum_{m=1}^{B}\frac{B!}{(m+1)!(B-m-1)!}(1-F(i-1))^{m+1}F(i-1)^{B-m-1}\\ &= \frac{F(i-1)}{(1-b(0,B,1-F(i-1)))(1-F(i-1))}\sum_{m=1}^{B}b(m+1,B,1-F(i-1))\\ &= \frac{F(i-1)}{(1-F(i-1))(1-b(0,B,1-F(i-1)))}\sum_{m=2}^{B}b(m,B,1-F(i-1)),\end{aligned}$$

where the last is obtained by noting that $b(B+1,B,1-F(i-1))=0$ and then changing the index of summation. Let

$$\Gamma(i-1)=\frac{F(i-1)}{(1-F(i-1))(1-b(0,B,1-F(i-1)))}\sum_{m=2}^{B}b(m,B,1-F(i-1)).$$

We can write $\Gamma(i)$ in the closed form

$$\Gamma(i)=\frac{F(i)}{(1-F(i))(1-F(i)^B)}\Big(1-B(1-F(i))F(i)^{B-1}-F(i)^B\Big),$$

by expanding the binomials and noting that they sum to 1.

No work needs to be done refining a concrete solution, so $\mathsf{GoodTreeWork}(0)=0$. Therefore, the recurrence for $\mathsf{GoodTreeWork}(i)$ becomes

$$\mathsf{GoodTreeWork}(i)=\sum_{j=1}^{i}\mathsf{NodeWork}(j)+\sum_{j=1}^{i-1}\mathsf{BadTreeWork}(j)\Gamma(j)$$

Using our expression for $\mathsf{NodeWork}(i)$, Eq. 1, the first term becomes

$$\sum_{j=1}^{i} k^{n-j} b^k = k^n b^k \sum_{j=1}^{i} (1/k)^j$$

Since the last is a geometric sum with closed form, we obtain, after some simplification

$$\sum_{j=1}^{i} \mathsf{NodeWork}(j) = k^{n-i} b^k \Big(\frac{k^i - 1}{k-1}\Big).$$

Thus,

$$\mathsf{GoodTreeWork}(i) = k^{n-i} b^k \Big(\frac{k^i - 1}{k-1}\Big) + \sum_{j=1}^{i-1} \mathsf{BadTreeWork}(j)\Gamma(j), \tag{18}$$

as claimed.

B Asymptotics for $\mathsf{BadTreeWork}(i)$

$$\sum_{j=1}^{n-1} \mathsf{BadTreeWork}(j) = \begin{cases} O(b^k k^{n-1}) & p < 1/B \\ O(b^k k^{n-1} n) & p = 1/B \\ O(b^k k^{n-1} (pB)^{n-2}) & p > 1/B. \end{cases}$$

Proof: Eq. 6 gives the following expression for $\mathsf{BadTreeWork}(i)$:

$$\mathsf{BadTreeWork}(i) = b^k k^{n-i} \frac{(pBk)^i - 1}{pBk - 1}.$$

This can be rewritten as

$$\mathsf{BadTreeWork}(i) = b^k k^{n-i} \sum_{m=0}^{i-1} (pBk)^m$$

Hence,

$$\begin{aligned} &\sum_{i=1}^{n-1} \mathsf{BadTreeWork}(i) \\ &\quad = b^k \sum_{i=1}^{n-1} k^{n-i} \Big(\sum_{m=0}^{i-1} (pBk)^m\Big) \\ &\quad = b^k (k^{n-1} + k^{n-2}(1 + pBk) + \cdots + k(1 + pBk + \cdots + (pBk)^{n-2})) \\ &\quad = b^k (k^{n-1}(1 + pB + \cdots + (pB)^{n-2})) + \text{lower order terms.} \end{aligned}$$

When $pB < 1$, the sum $(1 + pB + \cdots + (pB)^{n-2})$ is bounded above by the constant $1/(1 - pB)$. Hence, we get

$$\sum_{i=1}^{n-1} \mathsf{BadTreeWork}(i) = O(b^k k^{n-1}) \qquad pB < 1.$$

When $pB = 1$, this sum becomes $O(n)$. Hence, we get

$$\sum_{i=1}^{n-1} \mathsf{BadTreeWork}(i) = O(b^k k^{n-1} n) \qquad pB = 1.$$

When $pB > 1$, this sum becomes $O(pB^{n-2})$. Hence, we get

$$\sum_{i=1}^{n-1} \mathsf{BadTreeWork}(i) = O(b^k k^{n-1} (pB)^{n-2}) \qquad pB > 1.$$

■

C Semantic Conditions for the DRP

Theorem 3.3. Let $\hat{\Pi}_1$ be any i-th level abstract solution to any planning problem that has a concrete solution. Let $\mathsf{Abs}(i, \hat{\alpha}_j)$ and $\mathsf{Abs}(i, \hat{\alpha}_{j+1})$ be any pair of adjacent operators in $\hat{\Pi}_1$, and let Σ be the semantic state between these two operators in $\hat{\Pi}_1$.

Suppose that for every semantic state Σ_1 such that $\Sigma_1 \in \mathrm{Range}(\mathsf{Abs}(i-1, \hat{\alpha}_j))$ and $\Sigma_1 \subseteq \mathsf{Abs}(i, \Sigma)$, there exists a state Σ_2, such that

1. $\Sigma_2 \subseteq \mathsf{Abs}(i, \Sigma)$, $\Sigma_2 \in \mathrm{Domain}(\mathsf{Abs}(i-1, \hat{\alpha}_{j+1}))$, and
2. a solution $\hat{\Pi}_2$ consisting of $i-1$ level operators exists for the problem $\langle \Sigma_1, \Sigma_2 \rangle$, such that for every state Σ' in $\hat{\Pi}_2$, $\Sigma' \subseteq \mathsf{Abs}(i, \Sigma)$.

Then the DRP is satisfied by the hierarchy.

Proof: Let $\hat{\Pi}_1$ be any i-th level solution, and let $\hat{\Pi}'$ be a $i-1$-th level plan, formed from $\hat{\Pi}_1$ by inserting a $i-1$-th level subplan $\hat{\Pi}_j$ between every pair of operators $\mathsf{Abs}(i, \hat{\alpha}_j)$ and $\mathsf{Abs}(i, \hat{\alpha}_{j+1})$, $j = 0, \ldots, k-1$, in $\hat{\Pi}_1$, where $\hat{\Pi}_j$ is the subplan whose existence is specified in the condition 2. Then the resultant plan $\hat{\Pi}'$ is correct, since from this condition, $\hat{\Pi}_j$ does not violate the correctness of the i-th level conditions. Furthermore, from Definition 3.1, $\hat{\Pi}_2$ is also a monotonic refinement of plan $\hat{\Pi}_1$. Therefore, for every abstract solution, there exists a monotonic refinement of that solution, and the DRP is satisfied. ■

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, London, 1983.

[2] K. B. Athreya and P. E. Ney. *Branching Processes.* Springer-Verlag, New York, 1972.

[3] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 286–292, 1991.

[4] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem solving. In *Proceedings of the AAAI National Conference*, 1992.

[5] Peter Cheeseman, Bob Kanefsky, and Willian M. Taylor. Where the really hard problems are. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 331–337, 1991.

[6] Jens Christensen. A hierarchical planner that generates its own abstraction hieararchies. In *Proceedings of the AAAI National Conference*, pages 1004–1009, Boston, MA., 1990.

[7] William Feller. *An Introduction to Probability Theory and Its Applications: Volume 1.* John Wiley and Sons, New York, 1968.

[8] Eugene Fink. Justified plans and ordered hierarchies. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, N2L 3G2, Canada, 1992. available as Technical Report CS–92–42.

[9] Eugene Fink and Qiang Yang. Formalizing plan justifications. In *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI92)*, pages 9–14, Vancouver, BC, 1992.

[10] Fausto Giunchigilia and Toby Walsh. Abstract theorem proving. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 372–377, Detroit, MI, 1989.

[11] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, London, 1989.

[12] Joseph Y. Halpern and Yorham Moses. A guide to the modal logics of knowledge and belief. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 480–490, 1985.

[13] R. M. Karp and J. Pearl. Searching for an optimal oath in a tree with random costs. *Artificial Intelligence*, 21:99–116, 1983.

[14] Craig Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Technical Report CMU–CS–91–120.

[15] Craig Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the AAAI National Conference*, pages 686–691, 1991.

[16] Craig Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the AAAI National Conference*, pages 692–697, 1991.

[17] Craig A. Knoblock. An analysis of ABSTRIPS. In *Artificial Intelligence Planning Systems: Proceedings of the first international conference*, 1992.

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

[19] Hector J. Levesque. All I Know: A study in autoepistemic logic. *Artificial Intelligence*, 42:255–287, 1990.

[20] C. J. H. McDiarmid and G. M. A. Provan. An expected-cost analysis of backtracking and non-bactracking algorithms. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 172–177, 1991.

[21] Marvin Minsky. Steps towards artificial intelligence. In Edward A. Feigenbaum, editor, *Computers and Thought*, pages 406–450. McGraw-Hill, New York, 1963.

[22] Allen Newell, J. C. Shaw, and Herbert A. Simon. The processes of creative thinking. In *COmtemporary Approaches to Creative Thinking*, pages 63–119. Altherton Press, New York, 1962.

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

[24] D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

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

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

[27] L. Siklossy and J. Dreussi. An efficienct robot planner which generates its own procedures. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 423–430, 1973.

[28] D. E. Smith and M. A. Poet. A critical look at Knoblock's hierarchy mechanism. In *Artificial Intelligence Planning Systems: Proceedings of the first international conference*, 1992.

[29] Mark Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[30] Austin Tate. Generating project networks. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 888–893, 1977.

[31] J. D. Tenenberg. Abstraction in planning. In *Reasoning about Plans*. Morgan Kaufmann, San Mateo, California, 1991.

[32] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 681–687, Detroit, MI, 1989.

[33] David Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

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

[35] Colin Williams. Imperfect abstraction. In *Proceedings of the AAAI Workshop on Approximation and Abstraction of Computational Theories (mimeographed)*, 1992.

[36] Qiang Yang and Josh D. Tenenberg. Abtweak: Abstracting a nonlinear, least commitment planner. In *Proceedings of the AAAI National Conference*, pages 204–209, 1990.