Computing Abstraction Hierarchies by Numerical Simulation*

Alan Bundy
Dept of AI
University of Edinburgh

Fausto Giunchiglia
IRST, Trento and
University of Trento

Roberto Sebastiani
DIST
University of Genoa

Toby Walsh
IRST, Trento and
DIST, University of Genoa

Abstract

We present a novel method for building ABSTRIPS-style abstraction hierarchies in planning. The aim of this method is to minimize the amount of backtracking between abstraction levels. Previous approaches have determined the criticality of operator preconditions by reasoning about plans directly. Here, we adopt a simpler and faster approach where we use numerical simulation of the planning process. We demonstrate the theoretical advantages of our approach by identifying some simple properties lacking in previous approaches but possessed by our method. We demonstrate the empirical advantages of our approach by a set of four benchmark experiments using the ABTWEAK system. We compare the quality of the abstraction hierarchies generated with those built by the ALPINE and HIGHPOINT algorithms.

Introduction

In an ABSTRIPS-style abstraction, operator preconditions are ranked according to a criticality (Sacerdoti 1973). The i-th abstract space is constructed by ignoring preconditions with rank i or less. To refine a plan at the i-th level, we need to achieve those preconditions of rank i whilst preserving (or, where necessary, re-achieving) those with greater rank. Such abstraction hierarchies can give an exponential speed-up in the time needed to build a plan (Giunchiglia & Walsh 1991; Knoblock 1990). However, if we have to backtrack between abstraction levels, abstraction can greatly increase the time to find a plan. The "downward refinement property" (Bacchus & Yang 1994) removes the need to backtrack as every abstract plan can be refined to a concrete plan. Unfortunately, relatively few abstraction hierarchies possess this property. In practice, we therefore try to build abstractions which limit the amount of backtracking but do not preclude it altogether.

Previous approaches for building ABSTRIPS-style abstractions have reasoned about plans directly. For example, in ABSTRIPS (Sacerdoti 1973) low criticalities were assigned to those preconditions which can be achievied with short plans assuming all high criticality preconditions are true. More recently, ALPINE reasoned about operators to build abstraction hierarchies which satisfy the "ordered monotonicity" property (Knoblock 1994). In (Bacchus & Yang 1994), Bacchus and Yang show that backtracking may be needed with such abstraction hierarchies. To reduce backtracking, they propose the HIGHPOINT procedure. This refines the abstraction hierarchies produced by the ALPINE procedure using estimates of the probability for successful refinement. The abstractions produced by HIGHPOINT are close to having the downward refinement property (in the terminology of (Bacchus & Yang 1994), they are "near-DRP") but may still cause backtracking.

In this paper, we offer a novel method for building abstraction hierarchies which is both fast and simple. Instead of reasoning about plans directly, we simulate the planning process *numerically*. The simplicity of this simulation allows us to impose two simple "monotonicity" conditions not guaranteed by previous methods. These conditions ensure that harder preconditions are achieved at higher levels of abstractions. This greatly limits the amount of backtracking between abstraction levels. On four benchmark examples, our method give hierarchies which offer superior performance to those generated by both the ALPINE and HIGHPOINT algorithms.

Criticality functions

Given a set of operators, Ops, we compute the criticality of the operator precondition, p by successive approximation. At the n-th iteration, the *criticality function* $C(p, n)$ returns the *numerical criticality* of p. This converges to a limiting value as we iterate n. The intuition is that the easier it is to achieve p, the smaller the numerical criticality of p should be. We collect together the limiting numerical criticalities of the same value to give the sets S_i. We then order these sets using less than, giving $S_1 < ... < S_m$. Following (Sacerdoti 1973), the *criticality* of a precondition, p is the index

*Authors are listed in alphabetical order. The last author is supported by a HCM personal fellowship. We thank Qiang Yang for assistance with ABTWEAK and HIGHPOINT.

i such that $p \in S_i$. In the i-th level of abstraction, we drop all preconditions of criticality i or less. We thereby achieve the hardest preconditions in the most abstract space.

We impose various restrictions on criticality functions. For example, criticality functions should be *order independent.* That is, they should not depend on the order of the operators in the set Ops or the order of preconditions within an operator. Criticality functions also ought to treat symmetric preconditions symmetrically. If swapping the precondition p for the precondition q merely reorders the operators then p and q are said to be symmetric preconditions.

Definition 1 (Symmetry) *If p and q are symmetric preconditions then then $C(p,n) = C(q,n)$.*

We also demand that criticality functions treat equivalent effects equivalently. Let $Pre(op)$ be the preconditions of the operator op and $Ops(p)$ be the subset of operators which have p as primary effects. We say that a set of operators, S is *equivalent* to a set of operators, T iff $|S| = |T|$ and for any $op_1 \in S$ there is some $op_2 \in T$ with $Pre(op_1) = Pre(op_2)$ and vice versa.

Definition 2 (Precondition equivalence) *If $Ops(p)$ is equivalent to $Ops(q)$ then $C(p,n) = C(q,n)$.*

Finally, to reduce backtracking between levels, we demand that the numerical criticality of a precondition decreases with the number of operators which achieve it (operator monotonicity), and increases with the number of preconditions to operators which achieve it (precondition monotonicity).

Definition 3 (Operator monotonicity) *If $Ops(p)$ is equivalent to a subset of $Ops(q)$ then $C(p,n) \geq C(q,n)$.*

We say that a set of operators, S is *subsumed* by a set of operators, T iff $|S| = |T|$ and for any $op_1 \in S$ there is some $op_2 \in T$ with $Pre(op_1) \supseteq Pre(op_2)$. Note that if S is equivalent to T then S is subsumed by T and T is subsumed by S.

Definition 4 (Precondition monotonicity) *If $Ops(p)$ is subsumed by $Ops(q)$ then $C(p,n) \geq C(q,n)$.*

If operator monotonicity is satisfied, hard preconditions (those that are effects of few operators) will be proved in the higher abstraction levels. This will tend to minimize backtracking between abstraction levels. Similarly, if precondition monotonicity is satisfied, hard preconditions (those effects of operators with many preconditions) will be proved in the higher abstraction levels. Again this will tend to minimize the need to backtrack. Precondition and operator monotonicity both imply precondition equivalence.

ALPINE and HIGHPOINT generate abstraction hierarchies which fail to satisfy these properties and therefore cause unnecessary backtracking. Consider, for example, the manufacturing domain of (Smith & Peot 1992; Peot & Smith 1993). There are three operators which shape, drill and paint objects. For simplicity, we consider just their primary effects. The first operator has a single precondition `Object` and has `Shaped` as its effect. The second operator also has the single precondition `Object` and has `Drilled` as its effect. The third operator paints a steel object. It has `Object` and `Steel` as preconditions and has `Painted` as its effects. Precondition monotonicity ensure that the numerical criticality of `Painted` is greater or equal to that of both `Shaped` and `Drilled`. This agrees with our intuitions, as `Painted` requires an extra precondition. ALPINE, by comparison, assigns `Painted` the lowest criticality. As we will see in a later section, this can result in a large amount of backtracking.

Note that the trivial criticality function which assigns every precondition the same numerical criticality satisfies every one of these properties. This corresponds to no abstraction levels. We therefore maximize the number of abstraction levels by treating the "greater than or equal to" relations derived from the monotonicity properties as "strictly greater than" relations wherever possible. There are many non-trivial functions which satisfy these properties. However, these properties are often sufficient to rank numerical criticalities. For example, the rankings generated by our method in the next sections follow immediately from these properties.

The RESISTOR model

We propose a criticality function which satisfies the properties of the previous section based upon a model of "resistance to change". This model is analogous to that of electrical resistance. It attempts to model the difficulty of achieving preconditions. Precondition monotonicity means that operator preconditions act like resistors in series. Increasing the number of preconditions makes an operator harder to apply. Operator monotonicity, on the other hand, means that operators with the same effects act like resistors in parallel. Increasing the number of operators with the effect p reduces the difficuly of achieving p since we have parallel paths for achieving p. We shall refer to this as the RESISTOR model for computing criticalities.

To simplify the presentation, we introduce the notion of the numerical criticality of an operator. This represents the difficulty of applying an operator. As with serial resistors, the numerical criticality of an operator increases with the number of preconditions. We define the numerical criticality of an operator as the sum of the numerical criticalities of its preconditions. As with electrical resistors in parallel, multiple operators with the same effect reduce the difficulty of achieving that effect. We therefore define the numerical criticality of a precondition as the parallel sum of the numerical criticalities of the operators with this precondition as effects.

We interpret $C(p,n)$ as the difficulty of achieving p

with a maximum depth of n operator applications. In the base case, $n = 0$. That is, no operator applications are used. The difficulty of a precondition is the constant a_0. This models the constant time database lookup in the initial state. We therefore define,

$$C(p, 0) \;=\; a_0. \tag{1}$$

Note that a_0 always factors out of the final numerical criticalities. In the step case, a precondition can either be achieved using n operator applications or by being true in the initial state. The difficulty of a precondition is thus the parallel sum of the difficulty in the initial state and of the difficulty of any operators of which it is an effect. That is,

$$\frac{1}{C(p,n)} \;=\; \frac{1}{C(p,0)} + \sum_{op \in Ops(p)} \frac{1}{C(op,n)}. \tag{2}$$

Finally, applying an operator at depth n is as difficult as the serial sum of the difficulties of its preconditions at depth $n-1$. That is,

$$C(op, n) \;=\; \sum_{p \in Pre(op)} C(p, n-1). \tag{3}$$

The recursive nature of these definitions naturally leads to an iterative procedure for computing numerical criticalities.

Unsupervised preconditions are those that cannot be changed by any operators. Previous methods have conventionally given them the maximum criticality. By Equation (1), unsupervised preconditions are assigned the numerical criticality a_0 at $n = 0$. By Equation (2), their numerical criticality remains at a_0 for all subsequent n. Since a_0 is the largest numerical criticality possible, unsupervised preconditions are assigned the maximum criticality as required.

An Example

To illustrate the RESISTOR model for computing criticalities numerically, we use the computer hardware domain of (Bacchus & Yang 1994). This domain has four operators which print files, turn on devices, plug devices into power outlets, and transfer files onto computers. In Table 1, we give the numerical criticalities computed by the RESISTOR model for the different preconditions in this domain. The unsupervised preconditions are `CableCanReach`, `Functional`, `IsComputer`, `IsOutlet`, and `IsPrinter`. At $n = 4$, the numerical criticalities reach their limiting values. It can easily be proved that if, as at $n = 4$, the numerical criticalities remain stable for one iteration, then they will not change subsequently.

We group these numerical criticalities together, and order them using the less than relation. `Loaded` is assigned the lowest criticality of 0, `PowerOn` is given a criticality of 1, `PluggedIn` is assigned a criticality of 2, `Printed` is given a criticality of 3 and the unsupervised preconditions are given the highest criticality of 4. This is in line with our intuitions for this domain. The unsupervised preconditions cannot be changed so must be achieved in the most abstract space. The next hardest precondition to achieve is `Printed` since we must have a computer and printer turned on, and the file to print loaded on the computer. As we must plug in a device before turning it on, `PluggedIn` is assigned a greater numerical criticality than `PowerOn`. Finally, as loading a file onto a computer is less important than getting computers and printers plugged in and turned on, `Loaded` is given the lowest numerical criticality. In a later section, we demonstrate that this abstraction hierarchy is significantly better than that produced by the ALPINE algorithm, and offers slightly superior performance to the hierarchy generated by the HIGHPOINT procedure on larger problems.

X	$C(X,n)/a_0$				
	$n=0$	$n=1$	$n=2$	$n=3$	$n=\infty$
unsupervised	1.000	1.000	1.000	1.000	1.000
`Printed`	1.000	0.833	0.800	0.795	0.795
`PluggedIn`	1.000	0.667	0.667	0.667	0.667
`PowerOn`	1.000	0.667	0.625	0.625	0.625
`Loaded`	1.000	0.667	0.625	0.619	0.619

Table 1: Numerical criticalities for the computer hardware domain

Theoretical results

The RESISTOR criticality function is bounded in $[0, a_0]$ and monotonically decreasing. It is therefore convergent. Indeed, it typically converges very quickly. In the experiments in the next section, the change in numerical criticalities appears to decrease by at least a constant factor at each iteration. To explore this analytically, we developed a simple model in which each operator has m preconditions and each precondition can be achieved by l distinct operators. This gives an and-or search tree in which m is the and-branching and l is the or-branching. Under these assumptions, the numerical criticality of a precondition converges rapidly, with the difference between successive iterations being $O((l/m)^n)$ for $l < m$, $O(1/n^2)$ for $l = m$, and $O((m/l)^n)$ for $l > m$.

The RESISTOR criticality function is trivially order independent and symmetric. It also treats equivalent preconditions equivalently.

Theorem 1 *$C(p, n)$ is precondition equivalent*

Proof: By induction on n. In the base case, by Equation (1), all preconditions are assigned the same numerical criticality, a_0. Equivalent preconditions therefore have the same numerical criticality. In the step case, we unfold with Equations (2) and (3) and appeal to the induction hypothesis.

The RESISTOR criticality function also satisfies both the monotonicity properties.

Theorem 2 *$C(p,n)$ is operator and precondition monotonic.*

Proof: (Sketch) We define a notion of monotonicity which combines both operator and precondition monotonicity. The proof then uses induction on n. The base case follows immediately from Equation (1) since all preconditions have the same criticality. In the step case, using Equations (2) and (3) and some simple inequality reasoning, we can again appeal to the induction hypothesis.

For reasons of space, full proofs for all these theorems appear in an associated technical report.

Empirical results

To demonstrate the empirical advantages of the RESISTOR model, we ran a set of four benchmark experiments using the ABTWEAK system (Yang, Tenenberg, & Woods 1996), a state-of-the-art non-linear planner combining Abstrips-style abstractions (Sacerdoti 1973) with Tweak-style partial-order planning (Chapman 1987). In each experiment, we compared the quality of the abstraction hierarchies generated by the RESISTOR model with those built by the ALPINE and HIGHPOINT algorithms (Knoblock 1994; Bacchus & Yang 1994). These are two of the best available procedures for generating abstraction hierarchies.

The four experiments use standard benchmark problems taken from the literature. The first domain appears in (Knoblock 1994) and (Yang, Tenenberg, & Woods 1996). The next three are presented in (Bacchus & Yang 1994). We either repeated exactly the same experiments (for example, in the manufacturing domain), or we run them in a more exhaustive manner (for example, in the robot-box domain). We used two different measurements to evaluate ABTWEAK's performance with the different abstraction hierarchies: CPU time and the number of nodes expanded. The later is often a more reliable measurement of performance. All experiments were on a SUN Sparc 10 workstation with 32Mbytes RAM running compiled `Allegro CL 4.2` under the `Solaris 2` operating system[1].

Tower of Hanoi

The goal is to move a pile of three disks of different sizes from one peg to another using a third intermediate peg. At no time is a larger disk allowed to sit on a smaller one. The representation consists of an unsupervised type predicate `Is-peg(peg)`, and three predicates `On-small(peg)`, `On-medium(peg)` and `On-large(peg)`. ALPINE, HIGHPOINT and RESISTOR all produced the same abstraction hierarchy in which preconditions are abstracted according to their size. Thus, in the most abstract space, we just consider the large disk. In the next level of abstraction, we consider both the medium and large disks. And in the ground space, we consider all the disks. ALPINE generates this hierarchy in 0.01s, RESISTOR in 0.06s, and HIGHPOINT in 7.79s. In (Knoblock 1990), Knoblock shows that such a hierarchy reduces a breadth first search from exponential to linear. Similar abstraction levels are generated for towers with more disks giving, as here, an exponential reduction in search.

Criticality	Precondition
3	`Is-peg`
2	`On-large`
1	`On-medium`
0	`On-small`

Table 2: Criticalities generated by all methods for Tower of Hanoi domain.

Robot-box domain

This domain comes from (Bacchus & Yang 1994) and is a variant of the well-known ABSTRIPS robot domain (Sacerdoti 1973). The robot can either carry or pull boxes between one of six rooms. The doors connecting rooms may be either open or closed. Closed doors may be either openable or not openable. A typical configuration is given in Figure 1.

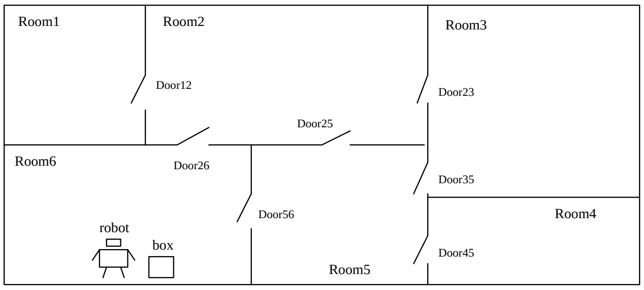

Figure 1: The robot-box domain.

For this domain, neither ALPINE or HIGHPOINT return criticalities which are order independent. The lowest three preconditions can be permuted by reordering the operators. This is because ALPINE constructs a partial order on preconditions which is then topologically sorted. To compare results, we used the ordering of operators which generates the same abstraction hierarchy as in (Bacchus & Yang 1994).

We ran experiments with both "easy" and "hard" problems. In the first set of experiments, all doors are openable. HIGHPOINT then constructs the same abstraction hierarchy as ALPINE. The criticalities are given in Table 3. ALPINE took 0.01s, HIGHPOINT 22.32s and RESISTOR 0.18s to generate these hierarchies. We ran ABTWEAK on all 30 possible goals

[1] Code used in these experiments is available from the authors' FTP site.

of moving between different rooms using these criticalities. Table 4 shows that while RESISTOR performs marginally better than ALPINE/HIGHPOINT, the differences between the hierarchies are not significant as backtracking is never needed.

ALPINE/HIGHPOINT	
4	Connects Is-Box Is-Door Is-Room Openable
3	Box-In-Room
2	Attached
1	Loaded
0	Open

RESISTOR	
3	Connects Is-Box Is-Door Is-Room Openable
2	Box-In-Room
1	Open
0	Attached Loaded

Table 3: Criticalities for the "easy" robot-box domain.

plan len	CPU times (secs)		nodes expanded		samples
	A/H	RES	A/H	RES	
3	0.66	0.62	28.86	27.86	14
6	4.24	4.07	143.64	142.64	14
8	12.95	12.55	379.50	378.50	2

Table 4: Mean performance on the "easy" robot-box domain in which doors can be unlocked.

In the harder set of experiments, certain doors are locked. HIGHPOINT increases the criticality of `Open` so that it is above `Attached` and `Loaded`. This reduces the probability of the robot meeting a locked door and thus the amount of backtracking. All other criticalities remain the same. ALPINE and RESISTOR return the same criticalities as before. We ran four sets of experiments. In each, `door25` and one of `door23`, `door26`, `door35` and `door56` are locked. In each case, there is just one unique path connecting any pair of rooms. For each set of experiments, we ran ABTWEAK on all 30 possible goals. In 8 out of the 120 problems, ABTWEAK exceeded the cut off bound of 2000 nodes using the HIGHPOINT and RESISTOR abstraction hierarchies. Using the ALPINE hierarchy, an additional problem also failed. The results are given in Table 5.

On this harder domain, the RESISTOR hierarchy performs slightly better than the HIGHPOINT hierarchy. Both perform significantly better than the ALPINE hierarchy as there is less backtracking caused by meeting locked doors. The poor mean performance of the ALPINE hierarchy was, in fact, entirely due to a small number of problems where ABTWEAK backtracked extensively.

Computer Hardware

We return to the computer hardware domain of (Bacchus & Yang 1994) discussed in an earlier section. The

plan len	CPU times (secs)			samples
	ALP	HIGH	RES	
3	0.91	0.74	0.82	40
6	6.20	5.33	5.32	40
8	39.42	29.03	28.99	24
10	82.58	69.64	67.45	7/8/8

plan len	nodes expanded			samples
	ALP	HIGH	RES	
3	28.30	28.30	27.30	40
6	162.92	160.32	159.32	40
8	775.08	752.67	751.67	24
10	1729.78	1654.87	1653.87	7/8/8

Table 5: Mean performance on the "hard" robot-box domain in which two doors are locked.

task is to print a file in an environment where there are a number of computers and printers. Computers and printers may not be turned on, may not be functional, or located near to a power outlet. As in (Bacchus & Yang 1994), we ran experiments in a domain in which at the initial situation just one computer and printer are within reach of a power outlet. The criticalities generated by the different methods are given in Table 3. ALPINE took 0.01s, HIGHPOINT 15.26s and RESISTOR 0.12s to generate these hierarchies. As in (Bacchus & Yang 1994), we ran ABTWEAK on 30 different problems involving between 1 and 3 files to print, and with between 1 and 10 computers, using a time limit of 1800 seconds. The results are given in Figures 2 to 4.

ALPINE	

4	`Cable-Can-Reach`
	`Functional`
	`Is-Computer`
	`Is-Printer`
	`Is-Outlet`

3	`Printed`
2	`Loaded`
1	`Power-On`
0	`Plugged-In`

HIGHPOINT	

3	`Cable-Can-Reach`
	`Functional`
	`Is-Computer`
	`Is-Printer`
	`Is-Outlet`

2	`Printed`
1	`Loaded`
0	`Power-On` `Plugged-In`

RESISTOR	

4	`Cable-Can-Reach`
	`Functional`
	`Is-Computer`
	`Is-Printer`
	`Is-Outlet`

3	`Printed`
2	`Plugged-In`
1	`Power-On`
0	`Loaded`

Table 6: Criticalities for the computer hardware domain.

ALPINE performs poorly in this domain, again due

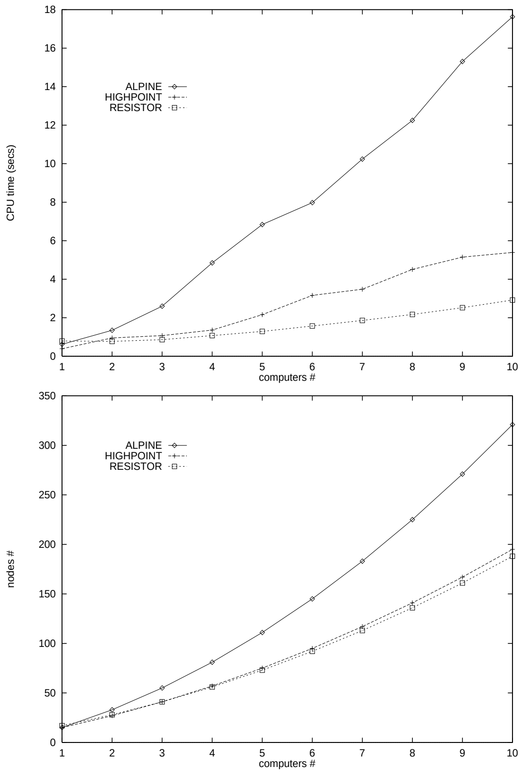

Figure 2: CPU time and nodes explored, 1 file to print.

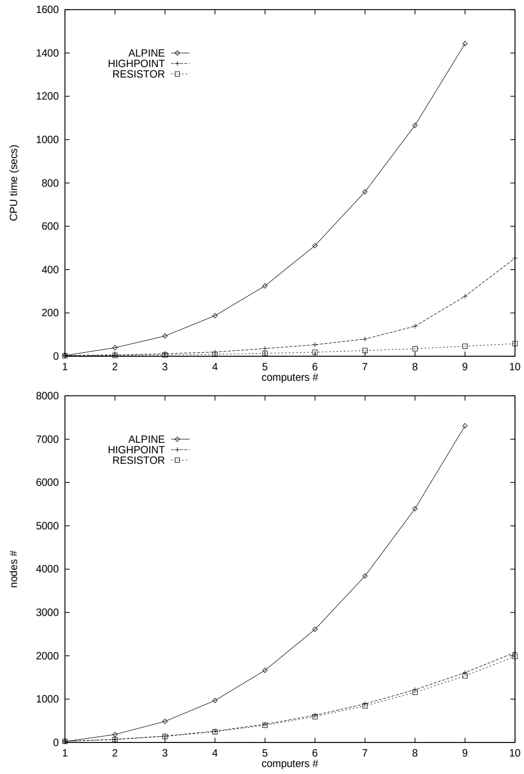

Figure 3: CPU time and nodes explored, 2 files to print.

to backtracking when devices are not plugged-in. RESISTOR and HIGHPOINT both require much less backtracking. The RESISTOR hierarchy gives slightly better performance, most noticeably on the larger problems.

Manufacturing

We return to the manufacturing domain of (Smith & Peot 1992; Peot & Smith 1993). The goal is to shape, drill and paint an object from stock. Recall that only steel objects can be painted. We assume that just one out of the large number of objects in stock are made from steel. The criticalities generated by the different methods are given in Table 7. ALPINE took 0.01s, HIGHPOINT 13.33s and RESISTOR 0.68s to generate these hierarchies.

ALPINE	
3	`Object` `Steel`
2	`Shaped`
1	`Drilled`
0	`Painted`

HIGHPOINT	
1	`Object` `Steel`
0	`Painted` `Drilled` `Shaped`

RESISTOR	
2	`Object` `Steel`
1	`Painted`
0	`Shaped` `Drilled`

Table 7: Criticalities for the manufacturing domain.

ALPINE's abstraction hierarchy violates the precondition monotonicity property as the `Painted` precondition should not be lower than either the `Shaped` or `Drilled` preconditions. HIGHPOINT compensates for the low probability of an object from stock being paintable by collapsing together the bottom three levels of ALPINE's abstraction hierarchy. This reduces the need to backtrack but gives just one level of abstraction.

RESISTOR is able to generate an additional level of abstraction. The `Shaped` and `Drilled` preconditions are equivalent and are placed at the bottom of the abstraction hierarchy. The `Painted` precondition appears above them as the operator for achieving it has an additional unsupervised precondition. The RESISTOR hierarchy is in line with the suggestions of Smith and Peot in (Smith & Peot 1992).

As in (Bacchus & Yang 1994), we ran ABTWEAK on problems with between 100 and 200 objects in stock. Results are plotted in Figure 5. The RESISTOR hierarchy results in less backtracking than the ALPINE hierarchy, and performs significantly better than the HIGHPOINT hierarchy due to the additional level of abstraction.

Conclusions

We have proposed a novel method for building ABSTRIPS-style abstractions automatically. The aim of this method is to minimize the amount of backtrack-

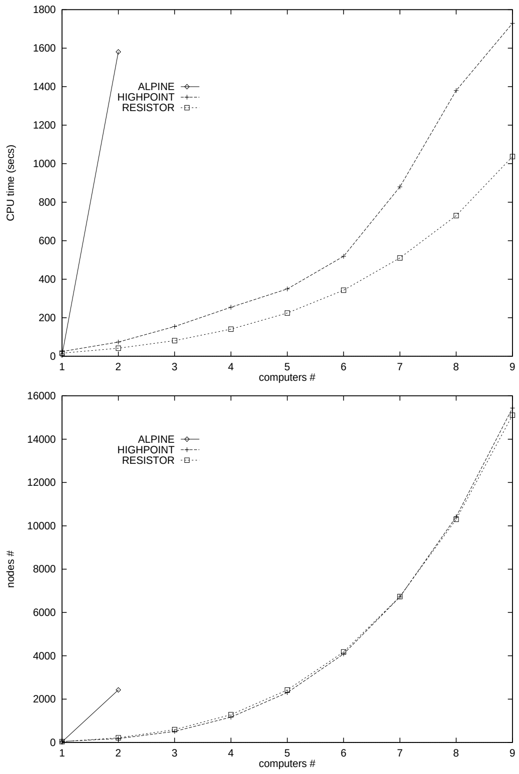

Figure 4: CPU time and nodes explored, 3 files to print.

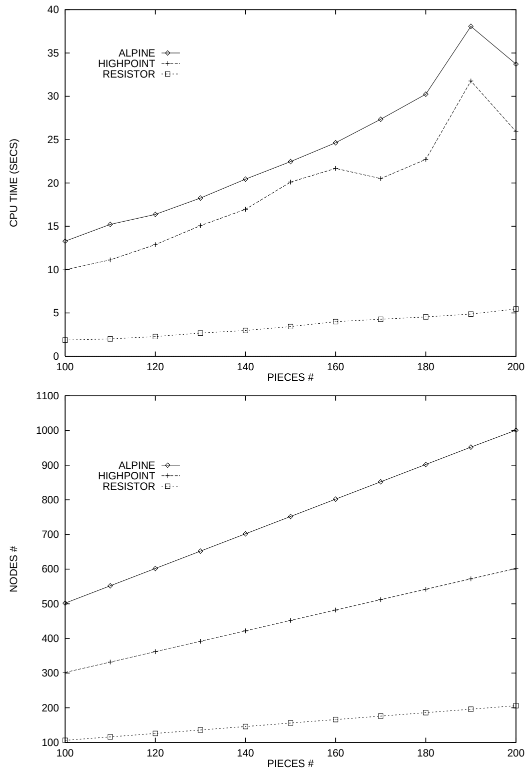

Figure 5: CPU time and nodes explored for the manufacturing domain.

ing between abstraction levels. Unlike previous approaches which reasoned about plans directly, we simulate the planning process numerically. Our model is based upon an analogy with electrical resistance. It is both fast and simple. The simplicity of our approach allows us to guarantee various theoretical properties hold lacking in previous approaches. In particular, the abstraction hierarchies constructed by our method satisfy two simple “monotonicity” properties. These ensure that the harder preconditions are achieved in the higher abstract levels. These monotonicity properties limit the amount of backtracking required between abstraction levels. We have compared our method with those in the ALPINE and HIGHPOINT procedures. Using a comprehensive set of experiments, we have demonstrated that the hierarchies constructed are better than those generated by ALPINE and HIGHPOINT. In addition, our method builds these hierarchies rapidly.

References

Bacchus, F., and Yang, Q. 1994. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence* 71:43–100.

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32:333–377.

Giunchiglia, F., and Walsh, T. 1991. Using abstraction. In *Proc. of the 8th Conference of the Society for the Study of Artificial Intelligence and Simulation of Behaviour*. Also IRST-Technical Report 9010-08 and DAI Research Paper 515, University of Edinburgh.

Knoblock, C. A. 1990. Abstracting the Tower of Hanoi. In *Working Notes of AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, 13–23. AAAI.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68:243–302.

Peot, M., and Smith, D. 1993. Threat-Removal Strategies for Partial-Order Planning. In *Proceedings AAAI-93*.

Sacerdoti, E. 1973. Planning in a Hierarchy of Abstraction Spaces. In *Proceedings of the 3rd International Joint conference on Artificial Intelligence.*

Smith, D. E., and Peot, M. A. 1992. A Critical Look at Knoblock's Hierarchy Mechanism. In *Proc. 1st International conference Artificial Intelligence planning systems (AIPS-92)*, 307–308.

Yang, Q.; Tenenberg, J. D.; and Woods, S. 1996. On the Implementation and Evaluation of `AbTweak`. *Computational Intelligence* 12.