To appear in AAAI-96

Commitment Strategies in Hierarchical Task Network Planning*

Reiko Tsuneto† reiko@cs.umd.edu

Kutluhan Erol** kutluhan@i-a-i.com

James Hendler†‡ hendler@cs.umd.edu

Dana Nau†‡ nau@cs.umd.edu

†Dept. of Computer Science
University of Maryland
College Park, MD 20742

‡Institute for Systems Research
University of Maryland
College Park, MD 20742

**Intelligent Automation, Inc.
2 Research Place
Rockville, MD 20850

Abstract

This paper compares three commitment strategies for HTN planning: (1) a strategy that delays variable bindings as much as possible; (2) a strategy in which no non-primitive task is expanded until all variable constraints are committed; and (3) a strategy that chooses between expansion and variable instantiation based on the number of branches that will be created in the search tree. Our results show that while there exist planning domains in which the first two strategies do well, the third does well over a broader range of planning domains.

Introduction

Two of the decisions that most AI planners must make are what order to perform the steps in, and what values to use for any variables in the plan. The planner's *commitment strategy*—its strategy for when and how to make these decisions—has long been known to play a great role in the efficiency of planning.

This paper compares the relative performance of three variable commitment strategies for Hierarchical Task Network (HTN) planning:[1] the Reluctant Variable Binding Strategy (RVBS), which delays variable bindings as much as possible; the Eager Variable Instantiation Strategy (EVIS), in which no non-primitive task is expanded until all variable constraints are committed; and the Dynamic Variable Commitment Strategy (DVCS), which chooses between expansion and variable instantiation based on the number of branches that will be created in the search tree.

Our results show that there are planning domains in which EVIS does well, and planning domains where it does poorly. The same is true for RVBS. However, DVCS, which can choose between eager variable commitment and reluctant variable commitment depending on what looks best for the task at hand, does well over a broader range of planning domains.

Previous Studies of Commitment Strategies

Commitment strategies have long been acknowledged to be important in AI planning, but only recently have researchers begun to analyze them rigorously (Barret and Weld 1994; Minton *et al.* 1991; Veloso and Stone 1995; Yang and Chan 1994). The studies that we know of all deal with STRIPS-style planning.

Kambhampati et al. (1995) have compared several domain-independent partial-order planners including UA (Minton *et al.* 1991), SNLP (Barret and Weld 1994), Tweak (Chapman 1987), and UCPOP (Penberthy and Weld 1992), and several other "hybrid" planning algorithms. In their experiments, the performance was affected more by the differences in tractability refinements than by the differences in protection strategies.

If a variable has 100 possible values, instantiating it will create 100 branches in the search space—and a planner might need to backtrack on all 100 branches for other unrelated reasons. To address such problems, Yang and Chan (1994) suggested maintaining the domains of the variables instead of binding them to constant values. In their experiments, extending SNLP to use this technique improved its performance in most cases.

Based on this past work we decided to concentrate on exploring the effects of variable commitments for HTN planning, to see if it also had a strong performance effect, as indicated in these experiments on partial-order planning. Preliminary work indicated it had a large effect, and the work described in this paper is aimed at analyzing this effect and exploring what commitment strategies work best in which domains.

*This research was supported in part by grants from NSF (IRI-9306580 and EEC 94-02384), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), the ARPA I3 Initiative (N00014-94-10907) and ARPA contract DABT-95-C0037. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

[1]We concentrate on variable assignment strategies because previous work suggests that these have a great effect on the performance of planning systems (see the next section).

1. Input a planning problem
P=<*d*: goal, *tn*, *I*: initial state, *D*: domain>.
2. Initialize OPEN-LIST to contain only *d*.
3. If OPEN-LIST is empty, then
   halt and return "NO SOLUTION."
4. Pick a task network *tn* from the OPEN-LIST.
5. If *tn* is primitive, its constraint formula is TRUE,
   and *tn* has no committed-but-not-realized
   constraints, then return *tn* as the solution.
6. Pick a refinement strategy *R* for *tn* .
7. Apply *R* to *tn* and insert the resulting set of
   task networks into OPEN-LIST.
8. Go to step 3.

Figure 1: High-level Refinement-Search in UMCP

HTN Planning and UMCP

The most recent and most comprehensive effort at providing a general description of HTN planning is Erol's UMCP algorithm (Erol 1995). Since UMCP provides the basis for our work, it is summarized below.

One way to solve HTN planning problems is to generate all possible expansions of the input task network to primitive task networks, then generate all possible variable assignments and total orderings of those primitive task networks, and finally output those whose constraint formulae evaluate to true. However, it is better to try to prune large chunks of the search space by eliminating in advance some of the variable bindings, orderings or methods that would lead to dead-ends. To accomplish this UMCP uses a branch-and-bound approach.

A task network can be thought of as an implicit representation for the set of solutions consistent with that task network. UMCP works by refining a task network into a set of task networks, whose sets of solutions together make up the set of solutions for the original task network. Those task networks whose set of solutions are determined to be empty are filtered out. In this aspect, UMCP nicely fits into the general refinement search framework described in (Kambhampati *et al*. 1995).

Figure 1 contains a sketch of the high-level search algorithm in UMCP. Search is implemented by keeping an OPEN-LIST of task networks in the search space that are to be explored. Depth-first, breadth-first, best-first and various other search techniques can be employed by altering how task networks are inserted and selected from the OPEN-LIST. Step 5 checks whether *tn* is a solution node; if all tasks in *tn* are primitive, the constraint formula is the atom TRUE, and the list of constraints that have been committed to be made true but not yet made true is empty, then all task orderings and variable assignments consistent with the auxiliary data structures associated with *tn* solve the original problem. Those plans can be easily enumerated. If *tn* is *not* a solution node, then it is refined by some refinement strategy *R*, and the resulting task networks are inserted back into the OPEN-LIST.

Three types of refinement strategies used in UMCP are task reduction, constraint refinement, and user-specific critics. Task reduction involves retrieving the set of methods associated with a non-primitive task in *tn*, expanding *tn* by applying each method to the chosen task and returning the resulting set of task networks. Constraint refinement involves selecting a group of constraints and making them true by adding ordering or variable binding restrictions to the task network. User-specific critics are domain-dependent strategies that a user can specify to improve the planner's performance.

Commitment Strategies in HTN Planning

In many planners, the commitment strategy is built into the search algorithm and cannot be modified by the user. For example, Tate's Nonlin system (Tate 1977) planner (and numerous planners based thereon) expanded tasks in a breadth-first manner: variables were instantiated by constants immediately after they were introduced to the plan if they unified with constants, and all constraints were applied before the next task expansion.

More recent HTN-style planners (e.g., O-Plan (Currie and Tate 1991)) use more sophisticated commitment strategies. The O-Plan system uses a number of criteria to decide when an entry in its agenda (list of things to be done) is ready to run. The criteria involve knowledge of how the plan is evolving and how potential interactions can be avoided.

In addition to the default automatic commitment strategies supplied by the system, planners like UMCP, O-Plan and SIPE-2 allow users to interact with the planning process to control commitments interactively. In the current implementation of UMCP, the system suggests the next process to the user at each decision point. The user can confirm the process suggested by the system, or can choose any other process applicable to the task network that the system is currently working on.

Below, we discuss some of the considerations that go into choosing a commitment strategy for HTN planning.

- *Expand first or refine constraints first?* This is analogous to the question of when to make commitments in STRIPS-style planning. A "least commitment" strategy would postpone constraint refinements until the planner gets a primitive task network. Since some state constraints and ordering constraints might not be fully realized while the task network has non-primitive tasks, this approach will eliminate the redundancy of working on the same constraints multiple times. On the other hand, earlier constraint refinement helps prune the search space. This is especially important if the planner is doing depth-first search as the search might keep failing for the same reason.

- *Which non-primitive tasks to expand?* This corresponds roughly to goal selection in STRIPS-style planning. One can do depth-first expansion (i.e., expand the most recently generated task first), breadth-first expansion, or any other systematic expansion method. If two tasks in a task network are known to be independent, it will be more efficient if the planner solves one task first in depth-first way and then deals with the other, so that it only has to backtrack over expansions of one task at a time.

- *Instantiate variables or maintain various constraints like CSP?* Yang and Chan (1994) argued the advantage of using deferred variable commitments. To delay variable bindings, they presented a CSP-like variable maintenance method that lets the planner postpone variable instantiation until absolutely necessary. UMCP uses a similar technique; refining variable or state constraints in UMCP either trims possible value lists or records variable distinctions.

  While Yang and Chan's argument applies to UMCP, there are certain domains which can use early variable instantiation handily. For instance, the n-puzzle is a highly complex domain since moving a tile to a desired location involves moving other tiles and thus might ruin other effects we want to preserve. It is easy to prune the search if the planner instantiates variables into constants because then it can detect redundant moves.

- *How to handle constraints in disjunctive formulas?* Refining disjunctive constraint formulas means making definite decisions at the point of search. Since formula simplification sometimes eliminates some constraints in the formula, there might be no disjunctions after some expansions and other refinements. However, if the planner has the right heuristics for the domain, early refinements of disjunctions have the same benefits as eager commitment.

In general, which commitment strategy is best can depend both on the problem domain and on the particular planning problem being solved in that domain. However, the following argument suggests that certain kinds of commitment strategies should be likely to do well across a wide variety of problem domains:

In the search tree for an HTN planner such as UMCP, each node represents a partial plan, and each edge represents a refinement made by the planner. If the search is systematic and does not prune nodes that lead to valid plans, then there should be the same number of *solution* nodes in the tree regardless of what commitment strategy we use. Suppose one commitment strategy does most of its branching near the top of the tree, and another does most of its branching near the bottom of the tree. If both trees have roughly the same height, then the second tree should usually have fewer nodes than the first tree (e.g., see Figure 2).

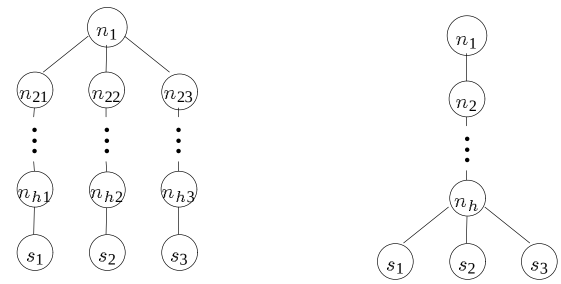

Figure 2: Two search trees with the same height *h* and the same set of solutions $\{s_1, s_2, s_3\}$. The one with the branch at the top of the tree has 3*h*+1 nodes. The one that with the branch at the bottom has *h*+3 nodes.

The above argument is not conclusive—for depending on how good the planner is at pruning unpromising solutions from the search space, the size and shape of its search tree is determined more by the set of *candidate* solutions (which is a superset of actual solutions) than the set of solutions itself. However, the *intuition* seems sound that a commitment strategy that tries to minimize the branching factor will do well. We hypothesized, and our experiments show, that we could exploit this feature as discussed in the next section.

Experiments

Although the argument at the end of the previous section is not conclusive, it suggests that a planner will do better if its commitment strategy keeps the branching factor near the top of the tree is as small as possible. One way to do this is to choose, at each node of the tree, the expansion or refinement option that yields the smallest number of alternatives. To test this hypothesis, we created an implementation of such a "dynamic commitment" strategy and compared it experimentally with implementations of a "least commitment" strategy and an "eager commitment" strategy. More specifically, the commitment strategies are as follows:

- **Eager Variable Instantiation Strategy (EVIS)** This is an HTN version of the eager variable commitment strategy described earlier. Don't expand any non-primitive task until all variable constraints are committed. Instantiate variables into constants whenever necessary to resolve constraints.

- **Reluctant Variable Binding Strategy (RVBS)** This is basically the opposite strategy. Delay instantiating variables as much as possible. Expand all tasks before making any variable binding constraints.

Method for toptask()		Method 1 for ctask(v1 v2)		Method 10 for ctask(v1 v2)
Expansion: (ctask v1 v2)		Expansion: (ptask1 v1 v2)	...	Expansion: (ptask10 v1 v2)
Constraints: v1$\neq$v2, (obj v1), (obj v2)		Constraint: (type v2 t1)		Constraint: (type v2 t10)

Figure 3: Methods for Domain A

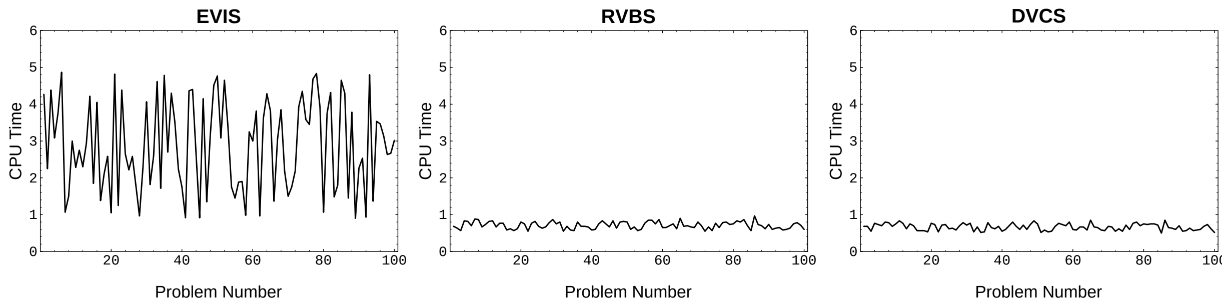

Figure 4: CPU time (in seconds) in Domain A

- **Dynamic Variable Commitment Strategy (DVCS)** This strategy attempts to minimize the branching factor as discussed earlier. Suppose T is the task network at the current node in the search space. For each variable x in T, let $v(x)$ be the number of possible values for v; and for each task t in T, let $m(t)$ be the number of methods that unify with t. Let $V = \min\{v(x) : x \text{ is a variable in } T\}$; and let $M = \min\{m(t) : t \text{ is a task in } T\}$. If $V < M$, then choose to instantiate the variable x for which $v(x)$ is smallest. If $M \leq V$, then choose to expand the task t for which $m(t)$ is smallest. Although this decision criterion may seem more complicated than EVIS and RVBS, the overhead involved in computing it is negligible.

  When $M = V$, we favor expansions over instantiations because further refinements might constrain the possible value set but not limit the number of methods. Unless the task network is pruned, expansion will eventually take place with same number of methods. On the other hand, it is possible to instantiate a variable with less number of possible values if the instantiation is delayed.

We compared the EVIS, RVBS, and DVCS commitment strategies by using them in the UMCP planner on randomly chosen problems in three different planning domains. The three planning domains—and our experimental results in those domains—are described below.

The experiments were run using Allegro Common Lisp on a SUN Sparc station, and running UMCP with a depth-first search strategy. For each problem and each commitment strategy, we counted both the CPU time and the number of nodes (i.e., the number of task networks) generated. Since both measurements gave similar results, below we will only discuss the CPU time.

Domain A

In Domain A the goal is to find a way to accomplish a 0-ary task (toptask). As shown in Figure 3, (toptask) expands into a 2-ary task (ctask v1 v2), where v1 and v2 are variables; and there are ten different methods for expanding (ctask v1 v2). The initial state is the set

$$\{\text{(obj obj1), (obj obj2), } \cdots \text{, (obj obj10), (type } o\ t\text{)}\},$$

where $o \in \{\text{obj1}, \ldots, \text{obj10}\}$ and $t \in \{\text{t1}, \ldots, \text{t10}\}$. Different planning problems are specified by choosing different values for o and t. Since the initial state has exactly one type literal, there is only one successful way to bind the variable v2 and expand the task (ctask v1 v2). The planning problem is to find the way that works.

We compared EVIS, RVBS, and DVCS in Domain A by running them on a suite of 100 randomly generated problems. Figure 4 shows the performance of UMCP with the three commitment strategies. There is exactly one solution for each problem. For each problem, RVBS and DVCS always find this solution after creating 14 task networks. Depending on the problem, EVIS creates between 24 and 114 task networks. UMCP's average CPU times were 2.88 seconds using EVIS, 0.71 seconds using RVBS, and 0.66 seconds using DVCS.

EVIS has more trouble than RVBS and DVCS because it instantiates the variable v2 before expanding the task ctask, and this tends to bind v2 to an object that does not meet the constraint found in the methods of ctask. On the other hand, RVBS does not instantiate v2 until after enforcing the constraint (type v2 t) so it does not make an instantiation of v2 which eventually fails. DVCS chooses to expand ctask before the instantiation of v2 since the values of V and M are the same (10), and thus performs identically to RVBS.

Method for (toptask)
Expansion: (ctask1 v1 v2)
Constraints: (obj v1), (obj v2)

Method 1 for (ctask1 v1 v2)
Expansion: (ctask2 t v1 v2 v3)
Constraints: (type v1 t1), (type v2 t1), (type v3 t1), v1≠v2≠v3

· · ·

Method 3 for (ctask1 v1 v2)
Expansion: (ctask2 t v1 v2 v3)
Constraints: (type v1 t3), (type v2 t3), (type v3 t3), v1≠v2≠v3

Method 1 for (ctask2 t v1 v2 v3)
Expansion: (ctask3 t v1 v2 v3)
Constraints: none

· · ·

Method 4 for (ctask2 t v1 v2 v3)
Expansion: (ctask3 t v1 v2 v3)
Constraints: none

Method 1 for (ctask3 t v1 v2 v3)
Expansion: (ptask t1 v1 v2 v3)
Constraints: none

· · ·

Method 4 for (ctask3 t v1 v2 v3)
Expansion: (ptask t3 v1 v2 v3)
Constraints: none

Figure 5: Methods for Domain B

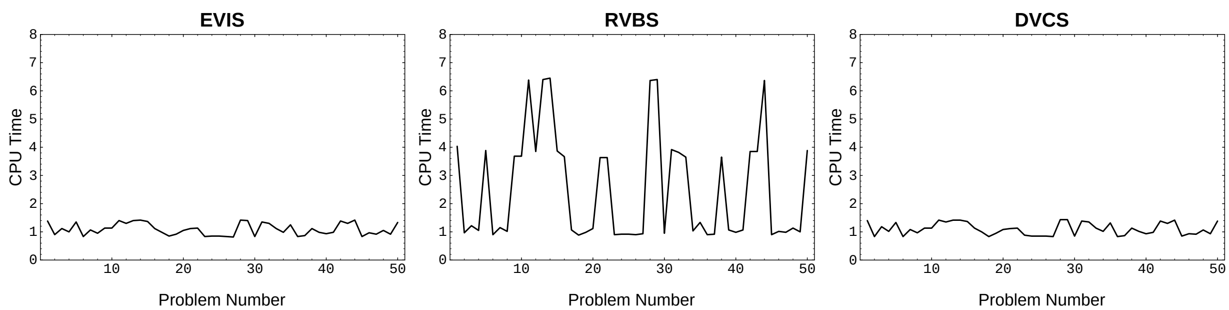

Figure 6: CPU time (in seconds) in Domain B

Domain B

Domain B is basically an encoding of the well known arc-consistency problem (Kumar 1992). As in Domain A, the goal is to accomplish toptask; but the methods are different. As shown in Figure 5, toptask expands into ctask1, ctask1 expands into ctask2, and ctask2 expands into ctask3. The methods for ctask1 specify that v1, v2 and v3 must have different values but the same type. ctask2 and ctask3 each have four identical methods, which increases the branching factor when UMCP does task expansion. The initial state is the set

$$\{\text{(obj obj1), (obj obj2)}, \cdots, \text{(obj obj7)}, \text{(type obj1 } t_1\text{), (type obj2 } t_2\text{)}, \cdots, \text{(type obj7 } t_7\text{)}\},$$

where each t_i is one of t1, . . ., t3. Different planning problems in this domain are specified by choosing different values for each of the t_i. The problem is to find three different objects which share the same object type.

In Domain B, we created a suite of 50 problems by randomly assigning types to each object obj_i in the initial state. Each problem had at least one solution. The results are shown in Figure 6. EVIS and DVCS created same number of task networks for each test problem, and incurred about the same amount of CPU time: with them, UMCP averaged 1.09 seconds and 1.10 seconds, respectively. RVBS never did better than EVIS or DVCS, and usually did much worse. On the average, UMCP's CPU time with RVBS was 2.54 seconds.

The reason for these results is that when EVIS instantiates variables v1, v2 and v3 before expanding the task ctask2, EVIS can prune the task networks which cannot satisfy the constraints imposed in the methods for ctask1. On the other hand, RVBS does not instantiate variables until they are fully expanded into primitive task networks. Thus RVBS generates task networks that would not be generated by EVIS.

Domain C

As shown in Figure 7, Domain C contains tasks and methods similar to those from both Domains A and B. Solving the problem involves combining methods similar to those in Domain A with methods similar to those in Domain B—but the order in which these methods should be used depends on whether the goal is toptaska or toptaskb. The initial state contains the atoms

$$\text{(obj obj1), (obj obj2)}, \cdots, \text{(obj obj10)},$$

and also fifteen atoms of the form (type o t) where type $\in \{\text{type1}, \text{type2}\}$; $o \in \{\text{obj1}, \ldots, \text{obj10}\}$; and $t \in \{\text{t1}, \ldots, \text{t3}\}$. Different planning problems are specified by choosing different values for o and t, as well as by choosing either toptaska or toptaskb as the goal.

In Domain C, we created a suite of 100 problems by randomly selecting the goal tasks and initial states. Of these problems, 44 problems had the goal task toptaska and 56 problems had the goal task toptaskb. Seven of

Method for (toptaska)
Expansion: (ctaska1 v1 v2)
Constraints: (obj v1), (obj v2)

Method 1 for (ctaska1 v1 v2)
Expansion: (ctaska2 t1 v1 v2 v3)
Constraints: (type1 v1 t1), (type1 v2 t1), (type1 v3 t1), $v1 \neq v2 \neq v3$

. . .

Method 3 for (ctaska1 v1 v2)
Expansion: (ctaska2 t3 v1 v2 v3)
Constraints: (type1 v1 t3), (type1 v2 t3), (type1 v3 t3), $v1 \neq v2 \neq v3$

Method 1 for (ctaska2 t v1 v2 v3)
Expansion: (ctaska3 t1 v1 v2 v3)
Constraints: none

. . .

Method 4 for (ctaska2 t v1 v2 v3)
Expansion: (ctaska3 t3 v1 v2 v3)
Constraints: none

Method 1 for (ctaska3 t v1 v2 v3)
Expansion: (ptaska t v1 v2 v3)
Constraints: (type2 v1 t1)

. . .

Method 3 for (ctaska3 t v1 v2 v3)
Expansion: (ptaska t v1 v2 v3)
Constraints: (type2 v1 t3)

Method for (toptaskb)
Expansion: (ctaskb1 v1 v2)
Constraints: (obj v1), (obj v2), $v1 \neq v2$

Method 1 for (ctaskb1 v1 v2)
Expansion: (ctaskb2 t1 v1 v2)
Constraints: (type1 v2 t1)

. . .

Method 3 for (ctaskb1 v1 v2)
Expansion: (ctaskb2 t3 v1 v2)
Constraints: (type1 v2 t3)

Method 1 for (ctaskb2 tp v1 v2)
Expansion: (ctaskb3 tp t1 v1 v2 v3)
Constraints: (obj v3), (type2 v1 t1), (type2 v2 t1), (type2 v3 t1), $v1 \neq v2 \neq v3$

. . .

Method 3 for (ctaskb2 tp v1 v2)
Expansion: (ctaskb3 tp t3 v1 v2 v3)
Constraints: (obj v3), (type2 v1 t3), (type2 v2 t3), (type2 v3 t3), $v1 \neq v2 \neq v3$

Method 1 for (ctaskb3 tp1 tp2 v1 v2 v3)
Expansion: (ptaskb t1 v1 v2 v3)
Constraints: none

. . .

Method 4 for (ctaskb3 tp1 tp2 v1 v2 v3)
Expansion: (ptaskb t3 v1 v2 v3)
Constraints: none

Figure 7: Methods for Domain C

the 100 problems had no solutions. As shown in Figure 8, DVCS had the best performance overall. UMCP's average CPU times were 2.15 seconds using EVIS, 1.83 seconds using RVBS, and 1.38 seconds using DVCS.

To test whether or not the differences shown in Figure 8 were statistically significant, we did a paired sample t-test. Let μ_D be UMCP's mean CPU time using DVCS and μ_R be UMCP's mean CPU time using RVBS. The null hypothesis H_0 is that $\mu_R - \mu_D = 0$ (or H_0: $\mu_R = \mu_D$); the alternative hypothesis H_1 is that $\mu_R - \mu_D > 0$. The t statistic computed from the results is 5.569. This is greater than the value 2.626 of the t-distribution with probability 0.995 where the degrees of freedom = 100. Thus we can reject H_0 and say that the difference of the means is significant. Similarly, we can say the difference of the mean CPU time for DVCS and the mean CPU time for EVIS is significant with the t statistic 8.155.

The reason why DVCS outperformed EVIS and RVBS is that even while solving a single planning problem, which commitment strategy is best can vary from task to task—and DVCS can select between the EVIS and RVBS strategies on the fly.

Conclusions and Future Work

We have discussed the impact of using appropriate commitment strategies in HTN planning. We believe that the choice of commitment strategies should depend on the problem domain and the particular problem. This paper is a first step to see how different commitment strategies affect the performance of HTN planning on different domains, and to explore whether variable commitment strategies have a significant effect on performance.

We have presented three variable commitment strategies, EVIS, RVBS and DVCS and examined their performance on three domains using the HTN planner UMCP. The results suggest while there is a domain where EVIS does well and a domain where RVBS does well, the dynamic strategy DVCS is a better choice overall. While DVCS does not always do better than both EVIS and RVBS, it cannot do worse than both of them.

In our experiments, when any of the commitment strategies selected variable instantiation, the variable which had the smallest number of possible values was chosen to be instantiated. This technique is known to work well in constraint satisfaction problems (Kumar 1992). However, another heuristic for choosing variable instantiation, more specific to HTN planning, is to instantiate those variables first that participate in the

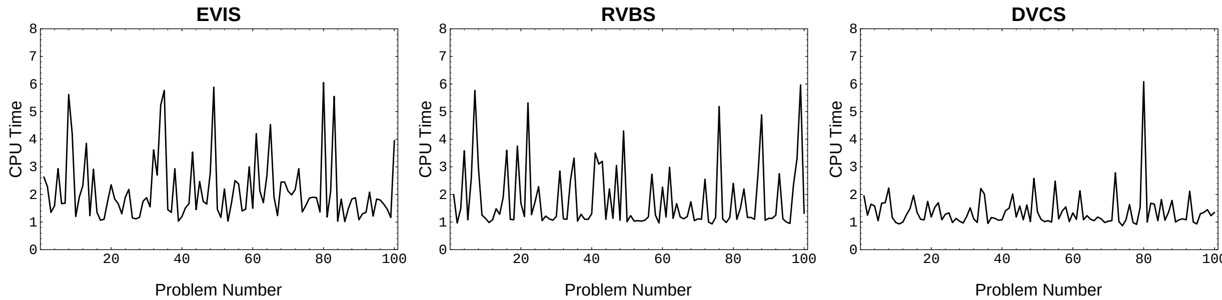

Figure 8: CPU time (in seconds) in Domain C

highest number of pending, but not yet bound, constraints. We intend to try this heuristic in the future.

Based on the results in this paper, it would seem that DVCS is a good variable binding strategy. We thus intend to explore how this commitment strategy performs in other planning problems such as simple domains like Blocks World, the test suites for UCPOP (Penberthy and Weld 1992), and more complex ones such as UM Translog (Andrews *et al.* 1995).

In addition, the DVCS approach of trying to minimize the branching factor can be extended for step-ordering commitments as well. While, as we described, we suspect that the variable commitment strategies will have a greater overall effect on planning efficiency, we hope that an approach to DVCS will also be effective for the introduction of ordering constraints. In general, we believe that dynamic commitment strategies perform better than static commitment strategies unless enough domain information is provided beforehand so that the user can foretell a static strategy would perform satisfactory, and we wish to test this out.

Although this paper discussed only domain-independent commitment strategies, a commitment strategy could also be highly domain specific. However, writing a good domain-specific commitment strategy requires much knowledge about the domain and the planning system. One of our goals is to build a methodology which can automatically extract the domain knowledge useful for efficient commitment strategies.

In particular, we hope to use AI learning techniques to develop domain-specific commitment strategies. Case-based reasoning (Veloso 1994) and explanation-based learning (Ihrig and Kambhampati 1995) are already used to learn search control for various planners, and we hope to extend this work to HTN planning. We also intend to explore the adjustment of dynamic heuristics such as DVCS based on feedback from experience in the domain.

References

S. Andrews, B. Kettler, K. Erol, and J. Hendler. UM translog: A planning domain for the development and benchmarking of planning systems. Technical report, CS-TR-3487, University of Maryland, 1995.

A. Barret and D. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence 67(1)*, pp. 71–112, 1994.

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

K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence 52*, pp. 49–86, 1991.

K. Erol. HTN planning: Formalization, analysis, and implementation. Ph.D. dissertation, Computer Science Dept., University of Maryland, 1995.

E. Fink and M. Veloso. Prodigy planning algorithm. Technical report, CMU-CS-94-123, Carnegie Mellon University, Pittsburgh, PA, 1994.

L. Ihrig and S. Kambhampati. Integrating replay with EBL to improve planning performance. ASE-CSE-TR 94-003, Arizona State University, 1995.

S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence 76*, pp. 167–238, 1995.

V. Kumar. Algorithms for constraint -satisfaction problems: A survey. *AI Magazine*, pp. 32–44, 1992.

S. Minton, J. Bresina, and M. Drummond. Commitment strategy in planning: A comparative analysis. In *IJCAI-91*, pp. 259–265, 1991.

J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. *Proceedings of KR-92*, 1992.

A. Tate. Generating project networks. In *IJCAI-77*, pp. 888–893.

M. Veloso and P. Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research 3*, pp. 25–52, 1995.

M. Veloso. Flexible strategy learning: Analogical replay of problem solving episodes. In *AAAI-94*, pp. 595–600, 1994.

Q. Yang and A. Chan. Delaying variable binding commitments in planning. In *AIPS-94*, pp. 182–187, 1994.