Learning Approximate Preconditions for Methods in Hierarchical Plans

Okhtay Ilghami OKHTAY@CS.UMD.EDU
Department of Computer Science, University of Maryland, College Park, MD 20742-3255, USA

Héctor Muñoz-Avila MUNOZ@CSE.LEHIGH.EDU
Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015, USA

Dana S. Nau NAU@CS.UMD.EDU
Dept. of Computer Science & Institute for Systems Research, Univ. of Maryland, College Park, MD 20742, USA

David W. Aha AHA@AIC.NRL.NAVY.MIL
Navy Center for Applied Research in AI, Naval Research Laboratory (Code 5515), Washington, DC 20375, USA

Abstract

A significant challenge in developing planning systems for practical applications is the difficulty of acquiring the domain knowledge needed by such systems. One method for acquiring this knowledge is to learn it from plan traces, but this method typically requires a huge number of plan traces to converge. In this paper, we show that the problem with slow convergence can be circumvented by having the learner generate solution plans even before the planning domain is completely learned. Our empirical results show that these improvements reduce the size of the training set that is needed to find correct answers to a large percentage of planning problems in the test set.

1. Introduction

One of the biggest obstacles to the development of AI planning systems that can be used in practical applications has been the difficulty of obtaining domain-specific problem-solving knowledge for the planning algorithm to use. Such information is essential to achieve satisfactory performance, but human domain experts often do not have enough time to provide information that is sufficiently detailed and accurate.

Consequently, it is important to develop algorithms that learn the necessary domain-specific information automatically (or semi-automaically) from some user-provided training data. However, finding such data can be difficult. Therefore, such algorithms should extract as much information as possible from the training data. In this paper, we describe CaMeL++, a domain-learning algorithm that can generate plans even before it has processed enough training data to learn the domain completely. Our empirical results show that these improvements reduce the size of the training set needed to find correct answers to a large percentage of planning problems in the test set.

2. HTN Planning

In *Hierarchical Task Network (HTN)* planning, the planning system formulates a plan by decomposing *tasks* (symbolic representations of activities to be performed) into smaller and smaller subtasks until *primitive* tasks are reached that can be performed directly. The basic idea was developed in the mid-70s (Sacerdoti, 1975; Tate, 1977). The development of the formal underpinnings came much later (Erol et al., 1996). Research on HTN planning has been much more application-oriented than on most other planning approaches, and most HTN planning systems have been used in one or more application domains

An *HTN planning problem* consists of the following: the *initial state* (a symbolic representation of the state of the world at the time that the plan executor will begin executing its plan), the *initial task network* (a set of tasks to be performed, along with some constraints that must be satisfied), and a *domain description* that contains, at least, a set of *planning operators* that describe the actions the plan executor can perform di-

Appearing in *Proceedings of the 22nd International Conference on Machine Learning*, Bonn, Germany, 2005.

rectly, and a set of *methods* that describe various possible ways of decomposing tasks into subtasks. Each method may have a *precondition* that must be satisfied in the world state exactly before the application of the method so that it can be applied.

Planning is done by applying methods to nonprimitive tasks to decompose them into subtasks, and applying operators to primitive tasks to produce actions. If this is done in such a way that all of the constraints are satisfied, then the planner has found a solution plan; otherwise the planner will need to backtrack and try other methods and actions.

3. An HTN Precondition Learner Based on Candidate Elimination

The performance criterion that interests us is *planning precision*, which is the percentage of the planning problems on which the planner gives a correct answer (i.e., a correct plan if one exists, or "no plan" if no correct plan exists). Ilghami et al. (2002) describe an algorithm called CaMeL, which learns preconditions of HTN methods from training data. While CaMeL can in theory achieve 100% precision, it requires a large number of training samples to do so. In Section 4 we describe CaMeL++, an algorithm that overcomes CaMeL's problem with slow convergence. In order to describe CaMeL++ intelligibly, we must first talk about CaMeL.

CaMeL is designed for domains in which the planner is given multiple methods per task, but not their preconditions. In other words, it is known in advance how to decompose tasks, but it is not known under what preconditions each of the several methods to decompose a task is applicable and should be chosen. This can happen for instance in a military operation, where the overall strategy of dividing tasks into subtasks is usually dictated by the military doctrine and therefore known, but the tactical decision of which of the available methods should be used is made based on the current situation.

CaMeL uses an extended version of Candidate Elimination (Mitchell, 1977). Candidate Elimination requires significant extension for use in an HTN planning context to handle issues like what the representational bias of the version spaces should be, or how the version spaces that represent preconditions of methods in different layers of the task hierarchy should interact with each other. Due to lack of space, we do not describe these extensions, and instead refer interested readers to (Ilghami et al., 2002).

CaMeL's training set consists of *plan traces*. Each plan trace contains a correct solution for a planning problem. Plan traces also include, at each given point in the planning process, a list of (potentially more than one) methods applicable to decompose the current task, in addition to the one applicable method that was actually selected in that particular solution to decompose the current task. These plan traces act as both positive and negative training samples: The absence of a method from the list of applicable methods at any point means that the absent method was *not* applicable given the state of the world at that particular point, hence a negative example. Note that each plan trace might be translated to more than one and potentially many positive or negative training samples for each of the methods in the domain being learned because there can be several task decompositions and examples of applications of methods in each plan trace.

4. Planning Before Full Convergence

One of CaMeL's drawbacks was that in many cases it required a huge number of plan traces to converge. Convergence occurs when the preconditions of all the task decomposition methods in a planning domain have been learned completely; i.e., when each of the version spaces representing each of those preconditions has converged to a single node. It can take a huge number of plan traces in the training set for this to happen, mainly because some methods can occur in plan traces less frequently, and certain related facts in the world state can be true (false) most of the time which makes it harder to observe what happens when those facts are false (true). For example, if the weather is good most of the time in a domain, it is harder to learn how the methods should react when it is bad; or if most of the cities in a transportation domain have airports, many plan traces are needed before even seeing what happens when a city does not have an airport.

CaMeL++ is based on the hypothesis that a high level of precision can be achieved even before full convergence has occurred. This is because *if certain facts are rare in the training data, the chances are they will also be rare in the test data (i.e., planning problems that the domain learner tries to solve after the learning phase).* Thus a planning system may do well on the test data even without having learned the facts that are rare. In this paper, we describe ways to extend CaMeL so that it uses this observation to start solving plans before all the methods have converged.

CaMeL++ replaces the all-or-nothing approach in CaMeL with a *voting scheme*. More specifically, methods can be used even if their preconditions are not fully learned (i.e., their corresponding version spaces con-

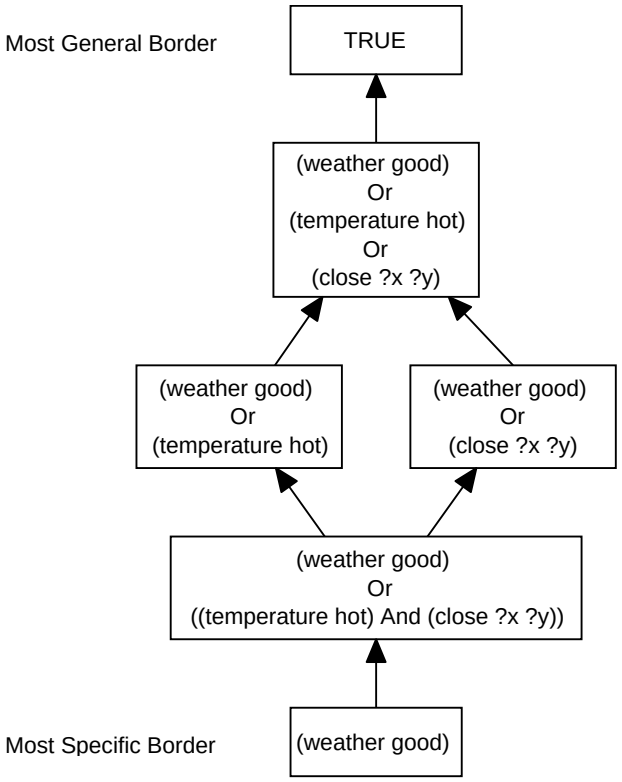

Figure 1. The version space for m_0's precondition after seeing 4 possible training samples where the weather is good.

tain more than one node). To verify if the precondition of such a method holds in a given world state, every node in the version space gets the chance to accept or reject the current state. If more than a certain threshold of nodes accept the current state (in other words, if the method's *acceptance ratio* is higher than the *acceptance threshold*), it is assumed that the method is applicable in the current state and therefore can be used to decompose the current task into its subtasks.

Imagine a domain d with a method m_0 that decomposes the task (go ?x ?y) to (walk ?x ?y) with precondition ((weather good) $\vee$ (close ?x ?y)) (Symbols with names starting with a question mark represent variables). In other words, one can go from a location to another one by walking between the two if either the weather is good or the two locations are close. Also, assume that there are three possible predicates in d: (weather good), (temperature hot), and (close ?x ?y). There are 8 different possible ways to combine these three predicates, with 6 making the application of m_0 possible and the other two making it impossible. Also, assume that in the planning problems in this domain, the weather happens to be good most of the time. This means that the four possible combinations of the above three predicates where the weather is good are the most likely to happen in a training set. Figure 1 shows the version space for the precondition of m_0 after the above four combinations (all of which happen to be positive examples) are given to CaMeL++ in a training set. Here the representational bias is that the preconditions can have at most three literals connected by logical ORs and ANDs. As can be seen, there are 6 elements in the version space, and a lot of training samples might be needed before this version space can converge to one element simply because we expect to see training samples where weather is not good infrequently. However, with the version space in Figure 1, if the acceptance threshold is set to anything above $\frac{1}{2}$ and below $\frac{5}{6}$, the version space will be able to correctly classify all the possible 8 combinations of the above three predicates as positive (i.e., m_0 can be applied) or negative (i.e., m_0 can not be applied) except one case: When (close ?x ?y) is true but the other two predicates are false, m_0 is applicable, but the version space in Figure 1 classifies it as negative since only 3 out of 6 elements in the Version Space accept this case. However, even this one case of misclassification is expected to happen rarely since the condition (weather good) is false in this situation.

In order for our voting scheme to work, the acceptance threshold should be set to a value lower than 1 (Setting it to 1 will result in the all-or-nothing approach of CaMeL). However, if it is set too low, and the training set is small (i.e., the Candidate Elimination frontier is large), this might yield too many applicable task decomposition methods for any point in the planning process. This might have two undesirable effects: First, this might slow down the planning process by increasing the branching factor of the planner's search. Second, and more importantly, this might cause the planner to produce many wrong plans when no plans exist for solving planning problems. To address this issue, we add a *beam-size* parameter to CaMeL++, which is a constant provided by the user. At any given point in the planning process, if there are more than beam-size available methods to decompose a task where the acceptance ratios of those methods is higher than the acceptance threshold, only the first beam-size methods with the highest acceptance ratios will be considered applicable. This limits the number of applicable methods at each given point in the planning process and should improve the planning precision. It should also improve the speed of plan generation in the starting phases of learning if the acceptance threshold is set too low (in which case more methods than necessary are applicable at each given point in the planning process). However, as more and more training data come in, the number of applicable methods eventually drops below the beam-size at some point. After this happens in the later stages of

learning, the effect of beam-size disappears.

There are two ways that a planner's precision can suffer: *false positives* and *false negatives*. A false positive happens when the planner returns a wrong plan for a given problem, or when the planner returns a plan when none exists that solves that problem. A false negative happens when there exist plan(s) that can solve a given problem, but the planner fails to find one. CaMeL never produces false positives (i.e., it is a sound planner)[1] since it uses a method only if it has provably learned its precondition, but this results in slow convergence and therefore higher number of false negatives before convergence. CaMeL++'s voting scheme reduces the number of false negatives (the lower the acceptance threshold, the lower the number of false negatives) but makes CaMeL++ prone to false positives. The beam-size then reduces the number of false positives (the lower the beam size, the lower the number of false positives).

5. Empirical Evaluation

As shown in (Ilghami et al., 2002), CaMeL sometimes needs a high number of plan traces to achieve full convergence. In this section, we show that the CaMeL++ algorithm requires fewer training examples than CaMeL to solve a similar percentage of planning problems in the test set. To accomplish this, CaMeL++ tries to solve problems before full convergence is achieved.

We used two different domains in our experiments: an HTN implementation of the well-known blocks world domain, and a simplified and abstracted version for planning a Noncombatant Evacuation Operation (NEO). NEOs are conducted to assist the U.S. Department of State with evacuating noncombatants, nonessential military personnel, selected host-nation citizens, and third country nationals whose lives are in danger from locations in a host foreign nation to an appropriate safe haven. The HTN implementation of blocks world we used has 6 operators and 11 methods and the NEO domain has 4 operators and 20 methods.

5.1. Simulating a Human Expert

NEO domains are usually complicated, requiring many samples to learn each method. It is difficult to obtain training samples for these kinds of domains. Even if we had access to real world NEO training samples, they would need to be classified by human experts and the concepts learned by CaMeL++ would need to be tested by human experts to assess their correctness. This would be expensive and time-consuming.

To overcome this problem, we used the same approach that was used in (Ilghami et al., 2002): To *simulate* a human expert. We used a correct hierarchical planner to generate planning traces for random planning problems on an HTN domain. Then we gave these plan traces to CaMeL++ as its training set and observed its behavior on the test set, another set of randomly generated problems.

The hierarchical planner we used is a slightly modified version of SHOP (Nau et al., 1999). In SHOP, if more than one method is applicable in some situation, the method that appears first in the SHOP knowledge base is *always* chosen. Since in our framework there is no ordering on the set of methods, we changed this behavior so that SHOP chooses one of the applicable methods randomly at each point. We also changed the output of SHOP from a simple plan to a plan trace.

5.2. Generating the Random Problems

For the blocks world, we generated sets of random problems with 300 blocks. We generated planning problems block by block, putting each new block randomly and uniformly onto an existing clear block or the table. All of the blocks in each problem have a known initial state and an associated goal statement.

For problem generation in the NEO domain, every possible state atom was assigned a random variable, indicating whether it should be present in the initial world state (e.g., should there be an airport in a specific city), or what value its corresponding state atom should have (e.g., should hostility level be *hostile*, *neutral*, or *permissive*). For all of these random variables we used a uniform distribution. There are a total of 7 such variables in this domain.

5.3. Results

In all the experiments reported in this Subsection, the size of the test set is 1000 for each domain for each run, and each reported result is the average of the results of 20 different runs. We have not used error bars in our graphs because each graph represents results for several different settings and therefore using error bars in the graphs makes them unreadable. However, the readers should know that in all of the reported results in this Subsection (which include planner precisions, running times, and percentages of correct plans and false negatives and positives) the result of each of the 20 different runs has been less than 10% different from the reported average.

[1] Assuming that training data is noise-free and each method's precondition is present in the Version Space.

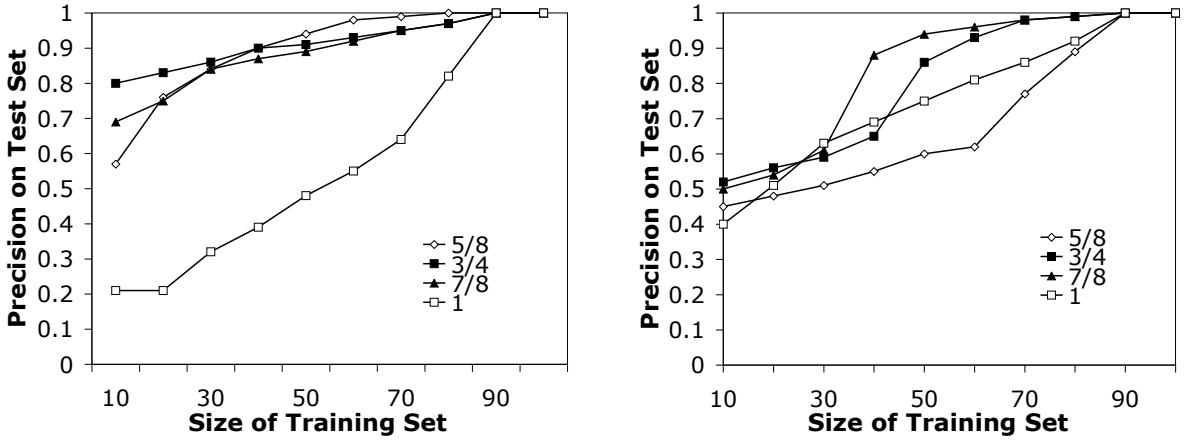

Figure 2. Planner precision given the training set size for different values of acceptance threshold in Blocks World (left) and NEO Domain (right).

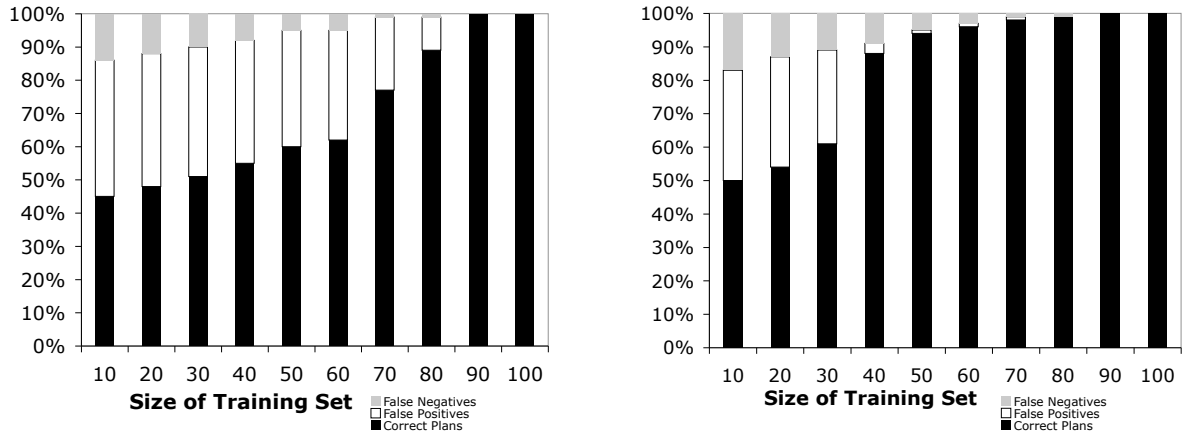

Figure 3. Error analysis for NEO Domain with acceptance thresholds of 5/8 (left) and 7/8 (right).

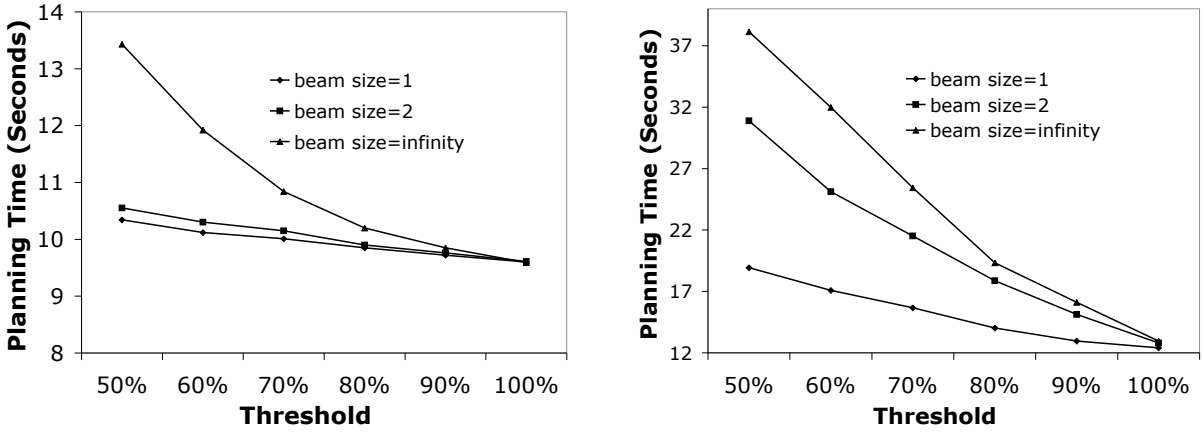

Figure 4. Planning time given acceptance threshold for different values of beam size in Blocks World (left) and NEO Domain (right).

Figure 2 shows CaMeL++'s precision with different values for the acceptance threshold given different training set sizes. As expected, in both domains, precision increases with training set size. However, the effects of choosing different acceptance thresholds in these domains are very different.

In blocks world, lowering the threshold from 1 (i.e., ordinary CaMeL) to $\frac{5}{8}$ greatly increases the speed of learning for precision. This is because all blocks world planning problems have solutions, and therefore there are fewer false positives compared to other domains in general. Because of this, it pays to relax the acceptance threshold in this domain; even without full evi-

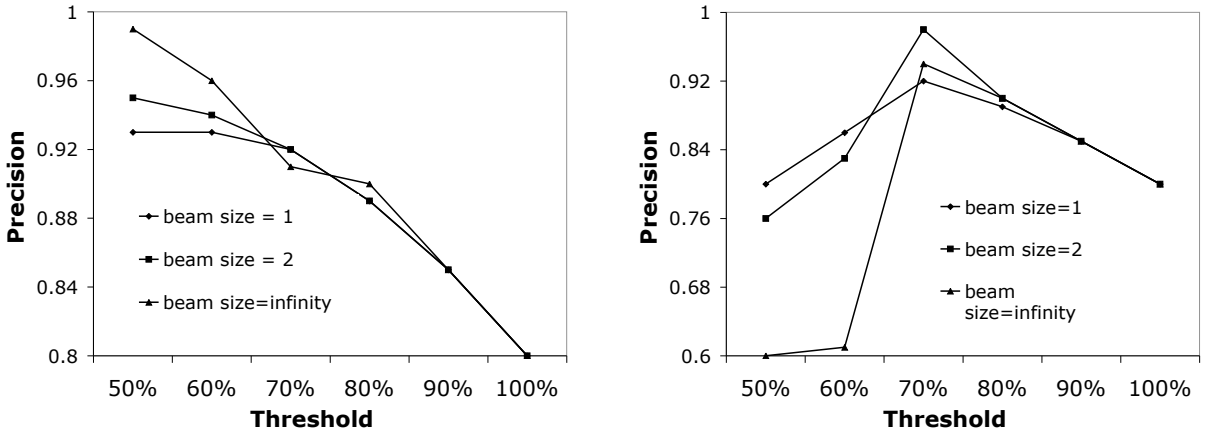

Figure 5. Planner precision given acceptance threshold for different values of beam size in Blocks World (left) and NEO Domain (right).

dence that a method is applicable, it helps to assume that it is indeed applicable and use it to find and return a plan. This explains why, in the blocks world, learning speed lags when the acceptance threshold is 1. It is simply too harsh an assumption for this domain. However, as can be seen, the lowest acceptance threshold is not necessarily the best one when the training set is small. Although every planning problem in the blocks world has a solution, every plan returned by CaMeL++ is not necessarily correct. That is, when the acceptance threshold is set too low, the planner might incorrectly assume that a method is applicable and use that method to construct and return an incorrect plan. This happens in the blocks world when the acceptance threshold is just $\frac{5}{8}$, and the number of training examples is small. Thus, CaMeL++ performs better when the acceptance threshold is set to $\frac{3}{4}$ or $\frac{7}{8}$ in the early phases of learning.

The situation is quite different in the NEO domain. The results when the acceptance threshold is set to either $\frac{5}{8}$ or 1 are worse than when it is set to values between them. The reason for this is that in NEO domain many of the randomly generated planning problems do not have any solutions, which causes a high number of false positives when the acceptance threshold is too low. On the other hand, there is a huge number of false negatives when the acceptance threshold is too high, especially for small training sets. When the acceptance threshold is $\frac{3}{4}$ or $\frac{7}{8}$, increasing the training set size beyond 30 and 40 greatly increases precision. This is mainly because, as more negative examples are seen, the percentage of false positives rapidly drops. At this point these values for the acceptance threshold start to outperform the other two (more extreme) values of $\frac{5}{8}$ and 1. This phenomenon is depicted in Figure 3. This Figure represents the proportion of correct plans (including "no plan" when there is none), false positives, and false negatives for acceptance thresholds of $\frac{5}{8}$ and $\frac{7}{8}$ respectively for different training set sizes in the NEO domain. As can be seen, in the former case, the majority of errors are false positives (because the threshold is too low and lax) while in the latter case the rate of false positives drops rapidly and is soon overtaken by the number of false negatives (which is higher than when the threshold is $\frac{5}{8}$).

This last observation led us to the hypothesis that the use of a beam size should have a positive effect on CaMeL++'s precision in the NEO domain where there are many false positives (especially when the acceptance threshold is set too low) and a slightly negative effect on its precision for the blocks world domain (because some of the methods that would have resulted in a correct plan will be discarded, which increases the number of false negatives). Using a beam size should also decrease the average time required by the planner to derive solutions for test problems because of the reduced branching factor in the planner's search space.

Figures 4 and 5 show our experimental tests of the above hypotheses. In these experiments, the size of the training set was 79 for the blocks world and 60 for the NEO domain. These were the sizes for which ordinary CaMeL (i.e., acceptance threshold of 1 and infinite beam size) achieved an 80% precision.

Figure 4 shows the average running time (taken on our set of 1000 planning problems in the test data) on a planning problem in the blocks and NEO domains for beam sizes of 1, 2, and infinite. (These experiments were conducted on a Sun Ultra 10 machine with a 440 MHz SUNW UltraSPARC-IIi CPU and 128 megabytes of RAM.) As can be seen, limiting the beam size modestly decreases the planning time, suggesting a decrease in the number of methods considered on average with lower beam sizes.

Figure 5 shows the precision of CaMeL++ with different beam sizes in the two domains. As expected, setting a beam size decreases CaMeL++'s precision for blocks world problems independently of the acceptance threshold setting. On the other hand, setting a beam size for the NEO domain increases CaMeL++'s precision in cases where the acceptance threshold is lower. The highest precision is achieved with a moderate acceptance threshold of 70% and a beam size of 2. This suggests that, for some domains, combining our voting scheme with an appropriate beam size can increase a planner's precision when there has not been enough training data for CaMeL++ to fully converge.

6. Related Work

CaMeL++ improves CaMeL, which utilizes version spaces to learn method preconditions. There are, however, other learning techniques, such as Inductive Logic Programming (ILP) and Explanation-Based Learning (EBL) that have been used before to learn control and/or domain knowledge. Reddy and Tadepalli (1997) introduce X-Learn, a system that uses a generalize-and-test algorithm based on ILP to learn goal-decomposition rules. These (potentially recursive) rules are 3-tuples that tell the planner how to decompose a goal into a sequence of subgoals in a given world state, and therefore are functionally very similar to methods in our HTN domains. X-learn's training data consists of solutions to the planning problems ordered in an increasing order of difficulty (authors refer to this training set as an *exercise set*, as opposed to an *example set* which is a set of random training samples without any particular order). The simple-to-hard order of samples in the training set is based on the observation that simple planning problems are often subproblems of harder problems and therefore learning how to solve simpler problems will potentially be useful in solving more difficult ones. Ruby and Kibler (1993) introduce another system that utilizes the above observation to learn recurring subplans. ILP techniques are also used to learn control rules for BlackBox, a planner that formulates planning problems as constraint satisfaction problems (Huang et al., 2000). The main difference between the above systems and CaMeL++ is that these systems learn control knowledge as opposed to CaMeL++ that learns domain knowledge. Another difference is that these systems use learning algorithms other than Candidate Elimination.

Garland et al. (2001) use *programming by demonstration* to build a system in which a domain expert performs a task by executing actions and then reviews and annotates a log of these actions. This information is then used to learn hierarchical task models. KnoMic (van Lent & Laird, 1999) is a learning-by-observation system that extracts knowledge from observations of an expert performing a task and generalizes this knowledge to a hierarchy of rules. An agent uses these rules to perform the same task. Langley and Rogers (2004) describes how ICARUS, a cognitive architecture that stores its knowledge of the world in two hierarchical categories of *concept memory* and *skill memory*, can learn these hierarchies by observing problem solving in sample domains. Although CaMeL++ and these systems both involve hierarchies, they differ in the expected input and the learning task. In CaMeL++, the structure of the hierarchy is known in advance and the learning task is to identify under what conditions different hierarchies are applicable, while these systems learn the hierarchies themselves.

Another way to formulate the problem of learning how to plan is to learn a function, called a *generalized policy*, that operates over all instances in the domain and maps states and goals into actions. The planner then uses this function to solve planning problems. Khardon (1999) and Martin and Geffner (2000) use this formulation and utilize ILP techniques to learn this function given solution plans to planning problems in the training set. This approach is similar to that of CaMeL++ in that it learns domain knowledge rather than control knowledge. The difference between these approaches and that of CaMeL++ is the input available to them. CaMeL++ has access to the hierarchy structure in the beginning and builds upon this available information by learning the method preconditions from the training data, while the only information that these systems have access to is the training data (i.e., they have no domain knowledge to begin with).

7. Conclusion and Future Work

In this paper, we described CaMeL++, an algorithm for learning preconditions for HTN methods that enables the planner to start planning before the method preconditions are fully learned. Our empirical results show that, by doing so, the planner can start solving planning problems with a smaller number of training examples than is required to learn the preconditions completely. These results also suggest that this speed-up comes at an insignificant cost of few incorrect plans. We also described how the characteristics of different planning domains might affect the usefulness of our suggested extensions.

For future work, we will explore more sophisticated voting schemes, and observe the effect such schemes might have on CaMeL++'s precision, learning rate,

and planning time. Furthermore, we want to observe the correlation between the beam size and the required size of the training set more closely in more general cases. We hope that our observation will result in ways to automatically infer, given the domain definition and also by observing the planning process, what the best values are for the parameters of our extensions (i.e., the acceptance threshold and the beam size).

Two of the biggest potential drawbacks of Candidate Elimination are its high sensitivity to noise, and sometimes the difficulty of representing the most general and most specific borders of the version space. Hirsh et al. have worked on extending the original Candidate Elimination algorithm to address these issues (1997; 2004). We intend to incorporate these extensions into our system and evaluate their effects on the performance of CaMeL++.

We also want to consider learning paradigms other than version spaces as replacements for Candidate Elimination currently used in CaMeL and CaMeL++ and compare the performance of different learning algorithms in the context of HTN domain learning.

Another topic for further investigation is *active learning*, i.e., enabling the learner to intelligently choose the next training sample it will be given, rather than just providing it with a randomly-generated training set. This way, the learner can ask for training samples that will help it gain more information more quickly and therefore it might be possible to converge with fewer training samples.

Acknowledgments

This work was supported in part by the following grants/contracts: NRL N00173-04-1-G033, NSF IIS0412812, AFOSR FA9550-05-1-0298, and DARPA's REAL initiative. The opinions expressed in this paper are those of authors and do not necessarily reflect the opinions of the funders.

References

Erol, K., Hendler, J., & Nau, D. S. (1996). Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence*, *18*, 69–93.

Garland, A., Ryall, K., & Rich, C. (2001). Learning hierarchical task models by defining and refining examples. *Proceedings of the 1st Int'l Conference on Knowledge Capture* (pp. 44–51).

Hirsh, H., Mishra, N., & Pitt, L. (1997). Version spaces without boundary sets. *Proceedings of the 14th Nat'l Conference on Artificial Intelligence* (pp. 491–496).

Hirsh, H., Mishra, N., & Pitt, L. (2004). Version spaces and the consistency problem. *Artificial Intelligence*, *156*, 115–138.

Huang, Y.-C., Selman, B., & Kautz, H. A. (2000). Learning declarative control rules for constraint-based planning. *Proceedings of the 17th Int'l Conference on Machine Learning* (pp. 415–422).

Ilghami, O., Nau, D., Muñoz-Avila, H., & Aha, D. (2002). CaMeL: Learning method preconditions for HTN planning. *Proceedings of the 6th Int'l Conference on AI Planning and Scheduling* (pp. 168–178).

Khardon, R. (1999). Learning action strategies for planning domains. *Artificial Intelligence*, *113*, 125–148.

Langley, P., & Rogers, S. (2004). Cumulative learning of hierarchical skills. *Proceedings of the Third International Conference on Development and Learning*.

Martin, M., & Geffner, H. (2000). Learning generalized policies in planning using concept languages. *Proceedings of the 7th Int'l Conference on Knowledge Representation and Reasoning* (pp. 667–677).

Mitchell, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the 5th Int'l Joint Conference on Artificial Intelligence* (pp. 305–310).

Nau, D. S., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 968–973).

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. *14th Int'l Conference on Machine Learning* (pp. 278–286).

Ruby, D., & Kibler, D. (1993). Learning recurring subplans. In S. Minton (Ed.), *Machine learning methods for planning*, 467–497. San Mateo, CA: Kaufmann.

Sacerdoti, E. (1975). The nonlinear nature of plans. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 206–214).

Tate, A. (1977). Generating project networks. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 888–893).

van Lent, M., & Laird, J. (1999). Learning hierarchical performance knowledge by observation. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 229–238).