CaMeL: Learning Method Preconditions for HTN Planning

Okhtay Ilghami and **Dana S. Nau**
Department of Computer Science
University of Maryland
College Park, MD 20742-3255
{okhtay,nau}@cs.umd.edu

Héctor Muñoz-Avila
Dept. of Comp. Sci. and Eng.
Lehigh University
Bethlehem, PA 18015
munoz@cse.lehigh.edu

David W. Aha
Navy Center for Applied Research in AI
Naval Research Laboratory (Code 5515)
Washington, DC 20375
aha@aic.nrl.navy.mil

Abstract

A great challenge in using any planning system to solve real-world problems is the difficulty of acquiring the domain knowledge that the system will need. We present a way to address part of this problem, in the context of Hierarchical Task Network (HTN) planning, by having the planning system *learn* the HTN methods incrementally under supervision of an expert. We present a general formal framework for learning HTN methods, and a supervised learning algorithm, named *CaMeL*, based on this formalism. We present theoretical results about CaMeL's soundness, completeness, and convergence properties. We also report experimental results about its speed of convergence under different conditions. The experimental results suggest that CaMeL has the potential to be useful in real-world applications.

Introduction

A great challenge in using any planning system to solve real-world problems is the difficulty of acquiring the domain knowledge and the associated control rules (i.e., rules that help the planner to search the search space efficiently) that abstract the real-world domain. One way to address this issue is to design the planning system to *learn* the constituents of the planning domain and the associated control rules. This requires the system to be supervised by a domain expert who solves instances of the problems in that domain. This will result in a *supervised learning* process. In this paper, we discuss a supervised incremental learning algorithm in a Hierarchical Task Network (HTN) planning context.

In recent years, several researchers have reported work on the HTN planning formalism and its applications (Wilkins 1990; Currie & Tate 1991; Erol, Hendler, & Nau 1994). The hierarchical semantics of this kind of planning gives us the ability to model planning problems in domains that are naturally hierarchical. A good example is planning in military environments, where conventional linear STRIPS-style planners (Fikes & Nilsson 1971) cannot be exploited to abstract the planning problems accurately. An example of using HTN planning in such environments is a system called HICAP (Muñoz-Avila *et al.* 1999), which has been used to assist with the authoring of plans for noncombatant evacuation operations. To support plan authoring, HICAP integrates the SHOP hierarchical planner (Nau *et al.* 1999) together with a case-based reasoning (CBR) system named NaCoDAE (Aha & Breslow 1997).

As with any incremental learning problem, there are at least two approaches that one might consider for learning HTN methods. First, a lazy CBR approach can be used to directly replay plans previously generated by the human expert. It assumes that plans that were successfully used in situations similar to the current situation are likely to work now. Second, an eager approach can be used to induce methods that could be used to mimic the human expert. In either approach, adding new training samples, which represent human expert activities while solving an HTN planning problem, is expected to yield better approximations of the domain. However, due to the complexity of the semantics of HTN planning, one should carefully define the inputs and outputs of the learning algorithm and what *learning* means in this context. In this paper, we use an eager approach.

In this paper, we introduce a theoretical basis for formally defining algorithms that learn preconditions for HTN methods. This formalism models situations when we have the following:

- General information about possible decompositions of tasks into subtasks, but without sufficient details to tell where each decomposition will be successful and when it won't.
- Plan traces that are known to be successful or unsuccessful for certain problem instances.

Such situations occur in several important practical domains, such as the domain of Noncombatant Evacuation Operations(DoD 1994; Lamber 1992), in which a military doctrine provides the planner with general information about how to carry out an evacuation operation, while the details of such an operation are not specified.

We also discuss CaMeL (Candidate Elimination Method Learner), an algorithm that instantiates this formalism. We state theorems about CaMeL's soundness, completeness, and convergence properties. Our experimental results show the speed with which CaMeL converges in different situations and suggest that CaMeL has the potential for use in deployed systems.

Hierarchical Task Network Planning

In an HTN planning system, instead of having traditional STRIPS-style operators with delete and add lists used to

achieve goal predicates (Fikes & Nilsson 1971), the main goal of planning is to accomplish a list of given *task*s. Each task can be decomposed into several *subtask*s using predefined *methods*. Each possible decomposition represents a new branch in the search space of the problem. At the bottom level of this hierarchy lie *primitive tasks*, whose actions can be executed using an atomic operator. In summary, the plan still consists of a list of instantiations of operators, partially ordered in some planners and fully ordered in other ones, but the *correctness* definition of the plan differs. In traditional planning, a plan is correct if it is executable, and the goal state is a subset of the world state after the plan's execution (i.e., each goal atom is achieved by some operator in the plan). In HTN planning, a plan is correct if it is executable in the initial state of the world, and it achieves the task list that is given as an input in the planning problem using the methods defined as a part of planning domain. In other words, the main focus of an HTN planner is task decomposition, while a traditional planner focuses on achieving the desired state. In this paper we use a form of HTN planning called *Ordered Task Decomposition* (Nau *et al.* 1999) in which at each point in the planning process, the planner has a totally ordered list of tasks to accomplish.

An *HTN domain* is a triple (T, M, O) where:

- T is a list of tasks. Each task has a name and zero or more arguments, each of which is either a variable symbol[1] or a constant symbol. Each task can be either *primitive* or *non-primitive*. A primitive task represents a concrete action, while a non-primitive task must be *decomposed* into simpler subtasks.
- M is a collection of *methods*, each of which is a triple $m = (NT, DEC, P)$, where NT is a non-primitive task, DEC is a totally-ordered list of tasks called a *decomposition* of NT, and P (the set of *preconditions*) is a boolean formula of first-order predicate calculus. Every free variable in P must appear in the argument list of NT, and every variable in DEC must appear either in the argument list of NT or somewhere in P. We will assume that each method m can be uniquely identified by its first two parts, NT and DEC (i.e., there will be no two different methods (NT, DEC, P) and (NT, DEC, P') such that $P \neq P'$).
- O is a collection of *operators*, each of which is a triple $o = (PT, DEL, ADD)$, where PT is a primitive task, and DEL and ADD are the sets of logical atoms that will be respectively deleted from and added to the world state when the operator is executed. All variables in DEL and ADD must appear in the argument list of PT. We also assume that, for each primitive task $t \in T$, there is at most one operator (PT, DEL, ADD) such that t unifies with PT (i.e., each primitive task can be performed in at most one way).

An *HTN planning problem* is a triple (I, S, D), where I (the initial task list) is a totally ordered list of ground instances of tasks (i.e., members of T) to be performed, S (the initial state of the world) is a set of *ground* logical atoms, and $D = (T, M, O)$ is an HTN domain.

A *binding* θ is a set of pairs (V, C), where V is a variable symbol, and C is a constant symbol. In other words, θ is a substitution that replaces variables with constants. The result of applying a binding θ to an expression e is denoted by $e\theta$.

The inverse of the binding θ, which replaces every constant C with its corresponding variable V (in θ) is denoted by θ^{-1}.

Let $m = (NT, DEC, P)$ be a method. Then $\bar{m} = (NT, DEC)$ is called the *incomplete version* of m. Similarly, if $m\theta = ((NT)\theta, (DEC)\theta, P\theta)$ is an instance of a method m, then the incomplete version of $m\theta$ is $\bar{m}\theta = ((NT)\theta, (DEC)\theta)$. Note that we have assumed that there are no two different methods (NT, DEC, P) and (NT, DEC, P') such that $P \neq P'$. Therefore, there is a one-to-one correspondence between a set of methods $\{m_1, \ldots, m_k\}$ and their incomplete versions $\{\bar{m}_1, \ldots, \bar{m}_k\}$ and thus each incomplete version $\bar{m}_i$ can be used to represent m_i.

Let $D = (T, M, O)$ be an HTN domain. Then, if $\bar{M}$ denotes the set of incomplete versions of all methods in M, $\bar{D} = (T, \bar{M}, O)$ is defined to be *the incomplete version* of D.

A method (NT, DEC, P) is *applicable* in a state S if there is a binding θ that binds all variables in NT's argument list such that $P\theta$ is satisfied in S.

A *partial solution tree* for a ground instance g of a task in an HTN domain $D = (T, M, O)$ is an ordered tree τ having the following properties:

- τ's root is g;
- Each node of τ is a ground instance of a task;
- For each non-leaf node N of τ, there is a method $m = (NT, DEC, P)$ and a binding θ such that $N = (NT)\theta$ and $(C_1, \ldots, C_k) = (DEC)\theta$, where $(C_1, \ldots, C_k)$ is the list of the children of N. $m\theta = ((NT)\theta, (DEC)\theta, P\theta)$ is called the *method instance* for N.

A *partial solution forest* for an HTN planning problem (I, S, D) is a totally ordered list $(\tau_1, \ldots, \tau_k)$ of partial solution trees, one for each member of I.

In a partial solution forest $F = (\tau_1, \ldots, \tau_k)$, a node u *occurs before* a node v in each of the following cases:

- u and v are in different trees τ_i and τ_j, respectively, where $i < j$.
- There is a node w such that u is the i'th child of w and v is the j'th child of w, where $i < j$.
- u and v have ancestors u' and v', respectively, such that u' occurs before v'. (Note that we may have $u' = u$ and/or $v' = v$.)

Given these definitions, it follows that the leaves of a partial solution forest are totally ordered by the "occurs before" relation.

A *solution* for a planning problem (I, S, D) is a partial solution forest F for (I, S, D) having the following properties:

- For every leaf node L of F, there is an operator $o = (PT, DEL, ADD)$ and a binding θ such that $L = (PT)\theta$. $o\theta$ is called the *operator instance* for L.

[1] We denote variable symbols by names that begin with question marks, such as $?x$.

- For every non-leaf node N of F, let $m = (NT, DEC, P)$ be the method instance for N. Let $(L_1, \ldots, L_k)$ be the totally ordered list of leaf nodes that occur before N, and let $o_1, \ldots, o_k$, respectively, be the operator instances for these nodes. Then P is satisfied in the state produced by starting with S and applying $o_1, \ldots, o_k$ sequentially.

Two HTN domains D_1 and D_2 are *equivalent* if and only if, for any arbitrary task list I and state of the world S, the planning problems (I, S, D_1) and (I, S, D_2) have exactly the same set of possible solution forests. For example, let $T = \{NT_1(?x), NT_2(?x), PT(?x)\}$ be a set of two non-primitive tasks and one primitive task, let $o = (PT(?x), \{\}, \{\})$ be an operator with empty add and delete lists that achieves the primitive task $PT(?x)$, let $m_1 = (NT_1(?x), (NT_2(?x)), a(?x))$ and $m_1' = (NT_1(?x), (NT_2(?x)), a(?x) \wedge b(?x))$ be two methods that can decompose the non-primitive task $NT_1(?x)$ into $NT_2(?x)$, and let $m_2 = (NT_2(?x), (PT(?x)), b(?x))$ be a method for decomposing the non-primitive task $NT_2(?x)$ into the primitive task $PT(?x)$. Then the domains $D_1 = (T, \{m_1, m_2\}, \{o\})$ and $D_2 = (T, \{m_1', m_2\}, \{o\})$ are equivalent.

Inputs to the Learning Algorithm

Motivation

For supervised learning of domains (either in an action-based or an HTN planning environment), two possible forms of input are:

- A set of previously generated plans. These plans can be generated in several ways (e.g., by a human expert in that domain). The learning process consists of extracting domain information from these plans.
- A collection of *plan traces*, which contain not only the correct solution for a planning problem, but also information about inferences derived and decisions made while this plan was generated (e.g., by a human expert).

The second form of input is preferable because it will result in faster and more accurate learning, because plan traces contain much more information about the domain being learned than plans. For this reason, most previous related work has used the second form of input. For example, in PRODIGY, a system that uses learning techniques in an action-based planning context, *derivational traces* are used in the learning process (Veloso & Carbonell 1993). These traces contain information about the planner's internal right and wrong decisions in addition to the final solution.

In this paper, we also use this second form of input, with appropriate adaptations for use in an HTN-planning environment rather than an action-based planning environment. In addition to its efficiency advantages, this form is well suited to our other goals. Ultimately, we want to develop a learning mechanism that can be used for HICAP (Muñoz-Avila *et al.* 1999), an interactive plan authoring system for HTN plans that allows manual editing of the plans by the user. We intend to soon develop a supervisor module on top of HICAP that track a user's edits during plan authoring. This sequence of edits corresponds roughly to the plan traces that we use as input to CaMeL.

In addition to the plan traces, we will assume that the input to the learning algorithm includes the incomplete version of the domain. By definition, the incomplete version includes the operator definitions, which seems reasonable given that operators usually denote concrete/simple actions with obvious effects.

Definitions

We are now ready to formally define plan traces and our inputs. A *plan trace* Π consists of (1) a solution to an HTN planning problem and (2) for each internal node N in the solution forest, an incomplete version of all instances of methods that were applicable. This will obviously include the instance of the method that was actually used to decompose N.

The *inputs* for an HTN method learning algorithm consist of:

- The incomplete version $\bar{D}$ of a domain D.
- A set of HTN planning problems $\{(I_j, S_j, D)\}$, where $1 \leq j \leq n$.
- A plan trace Π_j for each of these problems.

Outputs of the Learning Algorithm

One of the challenges in any learning algorithm is how to define a criterion to evaluate its output. Often, there is not enough information in the input or there are not enough training samples to derive an optimal output. Therefore, learning algorithms may return a set of *candidate* answers instead of a single answer. This phenomenon can affect the definition of soundness and completeness, which play a crucial role in evaluating outputs in a planning context.

We will assume that every constituent of the domains with which we are dealing is *deterministic* and training samples are not noisy. Therefore, there may be two reasons why a complete and correct definition of a method cannot be induced:

- *Lack of knowledge about the internal domain representation:*

  For example, suppose that a learning algorithm does not know whether the preconditions of methods being learned consist of only conjunctions, or whether disjunctions are also allowed. Suppose this learning algorithm is trying to learn a method m that decomposes the task $move(?t, ?s, ?d)$, and assume that the algorithm's inputs include two instances of m: an instance to decompose the task instance $move(truck_1, city_1, city_2)$ and the other one to decompose the task instance $move(truck_2, city_3, city_4)$. Suppose also that these two methods are applied, respectively, in the following world states:

  $$\{truck(truck_1), at(truck_1, city_1), color(truck_1, blue)\}$$
  $$\{truck(truck_2), at(truck_2, city_3), color(truck_2, red)\}$$

  If we give these two instances of m to an HTN method learner, then both

  $$truck(?t) \wedge at(?t, ?s)$$

and

$truck(?t) \wedge at(?t, ?s) \wedge (color(?t, blue) \vee color(?t, red))$

can be preconditions for method m, and the learner has no way to tell which one is the precondition of m if the learner doesn't know that only conjunctions are allowed.

- *Insufficient coverage of the domain information by training samples:*

  Consider another example, with the same method as in last example, and three instances: $move(truck_1, city_1, city_2)$, $move(truck_2, city_3, city_4)$ and $move(truck_3, city_5, city_6)$. The states of the world in which these three instances are applied are respectively:

  $\{truck(truck_1), at(truck_1, city_1), color(truck_1, blue)\}$

  $\{truck(truck_2), at(truck_2, city_3), color(truck_2, blue)\}$

  $\{truck(truck_3), at(truck_3, city_5), color(truck_3, red)\}$

  Suppose that the algorithm knows the preconditions of the methods in this specific domain contain only conjunctions. Knowing this, if the algorithm only encounters the first two examples, it will not be able to induce that $color(?t, blue)$ is not in the preconditions of this method, no matter how well it works, simply because there is not enough information in the given input to infer this. The best the algorithm can do is to say that the precondition of this method is a *generalization*[2] of $\{truck(?t), at(?t, ?s), color(?t, blue)\}$. When given the third training sample, the algorithm will be able to determine that $color(?t, blue)$ is not part of the method's precondition.

In this paper, we will assume that the exact form of the preconditions of methods for the target domain is known a priori. Thus, the only reason not to learn the exact method preconditions will be the fact that given training examples have not covered the whole domain information.

Before we can formally present soundness and completeness definitions in the context of HTN method learning, we need to define *consistent* answers:

A domain D_1 is *consistent* with another domain D_2 with respect to a set of pairs (I_j, S_j) of a task list I_j and a state of the world S_j if and only if for every j, the plan traces for HTN planning problems (I_j, S_j, D_1) and (I_j, S_j, D_2) are exactly the same. This definition says that although there may be differences in the methods and/or operators in D_1 and D_2 when solving problems relative to the task list I_j and state S_j for each j, the resulting traces are identical.

Using the above definition, we can now formally define soundness and completeness for HTN method learning algorithms. In order to make these definitions simpler, we will assume that output of an HTN method learner is a set of HTN domains rather than a set of possible method preconditions. This is a reasonable assumption, because the incomplete version of the domain is given to the learning algorithm as input, and the algorithm induces method preconditions for that domain. These preconditions are the only missing part in the domain definition. Therefore, having these preconditions, the learner can build a complete HTN domain from the incomplete domain given as input.

The definition of soundness is as follows:

Consider an HTN method learning algorithm whose inputs are

- $\bar{D}$, the incomplete version of a domain D;
- A set of HTN planning problems $\{(I_j, S_j, D)\}$, where $1 \leq j \leq n$;
- A plan trace Π_j for each of these HTN planning problems.

An HTN method learning algorithm is *sound* if, whenever it returns a set of HTN domains, each of them is consistent with D with respect to the set of all (I_j, S_j) pairs.

Consider an HTN method learner whose inputs are the three listed above. Then this algorithm is *complete* if, for every domain D' that is consistent with D with respect to the set of all (I_j, S_j) pairs, the algorithm's answer includes a domain that is equivalent to D'.

Another useful notion is *convergence*. Intuitively, it tells us whether an algorithm is expected to find a final answer in finite time. An algorithm *converges to the correct answer* in a domain D that satisfies our restrictions and assumptions, if and only if it is given a *finite* set of plan traces for the HTN planning problems (I_j, S_j, D) as input, and it terminates and outputs a set of HTN domains, each of which is consistent with D with respect to the set of *all* possible pairs of an initial task list and a world state. Apparently, all of the domains in the output set of a method learner that has already converged must be equivalent to each other.

Algorithm Implementation

Motivation

The main goal of learning HTN methods is to be able to generate plans for new planning problems (or *queries*). This ability will be obtained by learning how to plan using a set of previously generated plans or plan traces. In the machine learning literature, two entirely different kinds of learning, namely *lazy learning* and *eager learning* are discussed: In the purest form of lazy learning, training samples are simply stored. At query time, the algorithm compares the query instance with its recorded training samples. Thus, learning time is minimized while query time can be high, especially if no attempt is made to prune the number of stored training samples. On the other hand, eager learners induce an abstract concept during the training process. At query time, this concept, rather than the training samples themselves, are used to answer the query. Thus, learning time is higher than for purely lazy algorithms, while query time is usually lower.

In the context of method learning, lazy learning has been done using CBR, which involves locating those training samples that are most similar to the planning problem given as the query (Veloso, Muñoz-Avila, & Bergmann 1996; Hanney & Keane 1996; Lenz *et al.* 1998). This problem is then

[2] A conjunction C_1 of logical atoms is *more general* than another conjunction C_2 of logical atoms if and only if the set of logical atoms appearing in C_1 is a subset of the set of logical atoms appearing in C_2.

solved by adapting the solutions stored in the retrieved training samples. Our focus in this paper is on eager learning. Some of the advantages of using eager learning in our context are:

- *Less query time:* Lazy learning is useful when the number of training samples and frequency of query arrival is small. However, when these numbers are large, finding the most similar training samples can be time consuming (i.e., assuming that no smart indexing method is used). In such situations, eager learners are preferable.
- *Reduced knowledge base size:* Once the methods are learned, the system can discard the training samples and use the induced methods to solve new planning problems. In other words, the learned methods will act as a compact *summary* of the training cases. In contrast, purely lazy approaches require that, for every new planning problem, all previously seen examples must be revisited and therefore must be stored in the algorithm's knowledge base. Also, although several algorithms exist for significantly reducing storage requirements for case-based classifiers, they do not yet exist for case-based HTN planners.
- *Easier plan generation:* If we learn methods completely, then the process of generating a plan for a new planning problem will be much easier. This requires inducing methods for a hierarchical planner so that it can automatically generate plans for new planning problems. In contrast, a case-based planner must dynamically decide, for each new problem, which stored case or combination of cases to apply.

As mentioned earlier, HTN method learning algorithms may not be given enough information in the training set to derive a single exact domain as output. Therefore, the algorithm may return a set of domains. In these situations, a policy is needed to decide which possible domains should be output by the algorithm. Two extreme policies are:

- The *minimal* policy: Any possible domain is added to the output set if and only if there is enough evidence in the input to prove that this domain *must* be in the output.
- The *maximal* policy: Any possible domain is added to the output set if and only if there is not enough evidence in the input to prove that this domain *must not* be in the output.

For example, suppose that an HTN method learner is trying to learn the preconditions of a method m that is used to decompose the task $move(?t, ?s, ?d)$. Consider the following two cases:

1. Suppose the algorithm is told that there was an instance $move(truck_1, city_1, city_2)$ of method m that was applied when the world state was $\{truck(truck_1), at(truck_1, city_1)\}$. Using this training sample (which is a *positive* sample, since it tells us when a method is applicable), a minimal approach will yield: "The method m can be applied whenever $\{truck(?t) \wedge at(?t, ?s)\}$ is true in the world state." However, a maximal approach will yield "Method m is always applicable." This is because the algorithm has no way to prove that method m is not indeed applicable in some cases, according to its input.
2. Suppose the algorithm is told that there was a world state where $\{truck(truck_1), at(truck_1, city_2)\}$ was true, but m was not applicable to decompose the task $move(truck_1, city_1, city_2)$. (Recall that algorithm's inputs are plan traces, so it knows exactly at each state which methods are applicable to decompose a task.) If this *negative* training sample is the only input to the learning algorithm, a minimal approach will answer "Method m is never applicable" because no examples have been given that can prove that m is applicable in some cases. However, a maximal policy will answer with "Method m is applicable whenever the expression $\neg(truck(?t) \wedge at(?t, ?d))$ is satisfied in the world state."

The minimal policy yields a sound algorithm while the maximal policy yields a complete algorithm. However, both of these extreme approaches perform poorly and neither is both sound and complete. This is because there may be possible domains whose existence in the output set cannot be proved or disproved using the current input, simply because there is not enough information in the input to do so and more training samples are required. These possible domains are discarded in the minimal view and added to the output set in the maximal view.

One way to obtain better performance is to track the possible domains that cannot be proved or disproved so that they can be assessed in the future, after more training samples are acquired. Thus, we need an algorithm for tracking possible domains, while preferably maintaining its soundness and completeness.

Candidate Elimination

Candidate elimination is a well-known machine learning algorithm introduced in (Mitchell 1977). Several extensions of the original algorithm have been proposed (Hirsh 1994; Hirsh, Mishra, & Pitt 1997; Sebag 1995). Candidate elimination is based on the concept of a *version space*, the set of possible explanations of the concept that is being learned. This concept is represented by two sets: a set G of maximally general possible predicates to explain the concept, and a set S of maximally specific possible such predicates. Every concept between these two borders is a member of the version space, and is a possible explanation of the concept being learned. If enough training samples are given, the version space converges to a single answer (i.e., sets S and G become equivalent). It is also possible that the version space can collapse if the training samples are not consistent with each other.

Candidate elimination is indeed a very general algorithm. However, to apply this algorithm for a specific application, a generalization topology (a lattice) on the set of possible concepts must be defined (i.e., the generalization/specialization relation, along with the top and bottom of the lattice). In our algorithm CaMel, every method has a corresponding version space, and each member of a method's version space is a possible precondition for that method.

CaMeL requires both negative and positive examples. The concept of a "negative" example in a planning context may not be clear. However, for our context, negative examples can be generated easily. In our definitions, the input to

a method learner includes plan traces, each of which lists all applicable method instances that can be used to decompose a task instance in a specific world state. Therefore, if other methods can be used to decompose this task (i.e., for some other world states), we can infer that they were *not* applicable in those specific world states, and can hence serve as negative examples.

In our context, using candidate elimination has two main advantages. First, the resulting algorithm is sound and complete if our output is the set of all possible domains, where each method's set of preconditions can be any member of its corresponding version space. Second, candidate elimination is an *incremental* algorithm: whenever the algorithm acquires a new training sample, it just updates its version spaces and discards that training sample afterwards. There is no need to keep the training samples.

CaMeL: An HTN Method Learner Based On Candidate Elimination

Before detailing CaMeL, we need to introduce the notion of *normalization*. Suppose that an HTN method learner is trying to learn the preconditions of a method m that is used to decompose the task $move(?t, ?s, ?d)$. Assume that two instances of this method are given. The first instance is used to decompose the task $m_1 = move(truck_1, city_1, city_2)$ in the state $S_1 = \{truck(truck_1), at(truck_1, city_1)\}$, while the second instance is used to decompose the task $m_2 = move(truck_2, city_3, city_4)$ in the state $S_2 = \{truck(truck_2), at(truck_2, city_3)\}$. Apparently, these two training samples contain the same piece of information. Intuitively, it is: "You can move an object $?t$ from any starting location $?s$ to any destination $?d$ if $?t$ is a truck and it is initially at $?s$." But how is a learning algorithm supposed to derive such a general statement from such a specific example? This statement contains three variables, while the facts are about specific constants such as $truck_1$, $city_1$, and $city_2$. A generalization process is required that changes these constants to variables. This is roughly what happens during a normalization process.

Consider a ground instance $m\theta$ of a method m that decomposes task t for world state S. Then, $S\theta^{-1}$ is called a *normalization* of S with respect to $m\theta$, and θ^{-1} is called a *normalizer* for S. For the above example, both the normalization of S_1 with respect to m_1 and the normalization of S_2 with respect to m_2 yield $\{truck(?t), at(?t, ?s)\}$.

The normalization process replaces constants in different training examples that play the same role (e.g., $truck_1$ and $truck_2$) with a variable (e.g., $?t$) in order to generalize the facts that are given as input. Fact generalization is indeed a basic strategy in most eager learning algorithms.

The pseudo-code of our algorithm, CaMeL, is given in Figure 1. The algorithm subroutines are as follows:

- `LocateInstances(`Π`,`$\bar{m}$`)`, where Π is a plan trace and $\bar{m} = (NT, DEC)$ is an incomplete method, is a function that returns a set of triples (loc, θ, np). Each of these triples corresponds to one of the places where an instance $(NT)\theta$ of a non-primitive task NT was decomposed in Π. loc denotes the number of operators that occur before this instance of a task in the plan trace Π, and np is a boolean variable indicating whether method instance $m\theta$ was applicable to decompose $(NT)\theta$.
- `ComputeState(`$\bar{D}$`,`S`,`Π`,`loc`)` is a function that computes the world state after applying the first loc operators of the plan trace Π in incomplete HTN domain $\bar{D}$, where the initial world state is S.
- `CE(`S`,`VS`,`np`)` is an implementation of candidate elimination algorithm on version space VS with training sample S. np is a boolean variable that indicates whether S is a negative or positive example.
- `Converged(`VS_1`,...,`VS_m`)` checks if the algorithm has converged. As mentioned before, a method learning algorithm converges to an answer when and only when all of the members in its output set are equivalent to each other. CaMeL uses this fact to verify whether it has converged to an answer after processing each training sample. Note that, in general case, this function can be extremely expensive to evaluate in practice. However, in many cases where additional restrictions on the form of the preconditions are given, this function can be computed more quickly.

```
Given:
  D̄ = (T, M̄, O), the incomplete version
      of D = (T, M, O)
  I = {I_1, ..., I_n}, a set of task lists
  S = {S_1, ..., S_n}, a set of world states
  Π = {Π_1, ..., Π_n}, a set of plan traces,
      one per planning problem (I_i, S_i, D)

CaMeL(D̄, I, S, Π) =
  FOR each method m̄_j ∈ M̄
    Initialize a version space VS_j.
  FOR each plan trace Π_i ∈ Π
    FOR each method m̄_j ∈ M̄
      inst = LocateInstances(Π_i, m̄_j)
      FOR each (loc, θ, np) ∈ inst
        S'_i = ComputeState(D̄, S_i, Π_i, loc)
        VS_j = CE(Normalize(S'_i, θ), VS_j, np)
        IF Converged(VS_1, ..., VS_m)
          RETURN all VS_j's
  RETURN all VS_j's
```

Figure 1: The CaMeL Method-Learning Algorithm

Algorithm analysis

In this section, we will discuss the assumptions and restrictions a domain must satisfy in order for CaMeL to work correctly. We will also propose a few theorems about CaMeL in the framework we have defined.

Two kinds of constants may appear in an HTN plan trace Π or its corresponding HTN planning problem (I, S, D). First, *explicit* constants appear explicitly in the domain definition

D (e.g., in the effects of operators, argument list of tasks, or decomposition list of methods). Second, *implicit* constants are those that do not appear in the D explicitly. These constants appear in a plan trace because some of the variables in D were instantiated to them while the plan trace was created.

For example, consider an HTN method m for decomposing the task $go(?x, ?y)$ that decomposes this task to a primitive task $walk(?x, ?y)$. The precondition of this method is $weather(good) \wedge at(?x)$. Now, if there are two atoms $at(home)$ and $weather(good)$ in the current world state and an instance of the method m is used to decompose the task $go(?x, ?y)$ to the subtask $walk(home, station)$ in the corresponding plan trace, then the constants $home$ and $station$ are implicit constants, while in the HTN domain definition, $good$ is an explicit constant.

As another example, in the blocks world domain, $table$ is an explicit constant while names of blocks are implicit constants.

When normalization is used to generalize the training samples, the following crucial assumption is made:

Assumption 1 *No constant can be both implicit and explicit.*

Another assumption should be made because we use the candidate elimination algorithm. In order for candidate elimination to work properly, the terms *more general*, *more specific*, etc., must be defined for the set of possible members of the version space. CaMeL uses version spaces to show the possible preconditions of methods, which in general can be any first order predicate calculus formula. Unfortunately, these terms cannot be defined for the set of all possible boolean formulas in first order predicate calculus.

Assumption 2 *Preconditions of the methods have a* known *form, and this form is such that the relations* more general, less general, more specific, less specific, maximally general, maximally specific, minimal generalization *and* minimal specialization *can be defined for them.*

This assumption defines the learning algorithm's *representational bias*. A representational bias defines the states in the search space of a learning algorithm (Gordon & Desjardins 1995). It guarantees that we can generalize given facts about the training samples (Mitchell 1980).

Given these assumptions, the following theorems can be proved:

Theorem 1 *CaMeL is a sound and complete HTN method learner.*

Theorem 2 *For any given HTN domain D that satisfies our restrictions and assumptions, there exist a finite set of plan traces which causes the CaMeL to converge to D.*

We omit the proofs here, due to lack of space. The proof of theorem 1 is straightforward: It proceeds by applying the relevant theorems about version space algorithms repeatedly, once for each method to be learned. The proof of theorem 2 is more complicated, as it has to deal with interactions among tasks and their subtasks, sub-subtasks, and so forth.

Experiments

Theorem 2 says that there always exists a set of training samples that causes CaMeL to converge if the domain definition satisfies our restrictions and assumptions. However, this theorem does not give us any information about the number of training samples in such a set. What we need in practice is an answer to the question "How many samples will be needed to converge in average case?". In this section, we will discuss our experiments to answer this question.

Test Domain

The domain we used is a simplified and abstracted version of planning a NEO (Noncombatant Evacuation Operation). NEOs are conducted to assist the U.S.A. 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 decision making process for a NEO is conducted at three increasingly-specific levels: *strategic*, *operational*, and *tactical*. The strategic level involves global and political considerations such as whether to perform the NEO. The operational level involves considerations such as determining the size and composition of its execution force. The tactical level is the concrete level, which assigns specific resources to specific tasks. This hierarchical structure makes HTNs a natural choice for NEO planning.

Simulating a Human Expert

One goal in our work is to learn methods for HTN planners in military planning domains. These domains are usually complicated, requiring many samples to learn each method. It is difficult to obtain these training samples for military domains. Even if we had access to the real world NEO training samples, those samples 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 very expensive and time-consuming.

In order to overcome this problem, we decided 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 fed these plan traces to CaMeL and observed its behavior until it converged to the set of methods used to generate these plan traces.

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.

Generating the Training Set

In order to generate each plan trace, we had to generate a random NEO planning problem and feed it to the modified version of SHOP. To generate a random NEO planning problem,

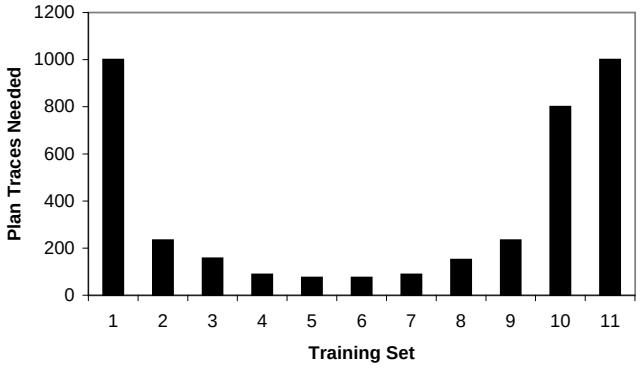

Figure 2: Number of plan traces needed to converge

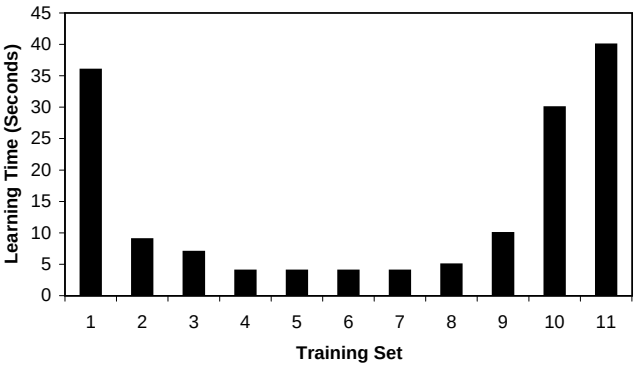

Figure 3: CPU time used by CaMeL to converge

every possible state atom was assigned a random variable, indicating whether or not it should be present the initial state of the world (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*). In our preliminary experiments, we noticed that the distribution of these random variables did not affect our experiments very much. However, there is one exception to this rule: the probability P that there is an airport in a city makes a lot of difference. Therefore, we decided to assign a uniform distribution to all random variables other than P and to perform experiments with several different values of P. We conducted eleven sets of experiments, with $P = \frac{1}{12}, \frac{2}{12}, ..., \frac{11}{12}$.

Results

After generating the eleven sets of training samples for $P = \frac{1}{12}, \frac{2}{12}, ..., \frac{11}{12}$, we fed each training set to CaMeL until it converged. Figure 2 shows the number of plan traces needed in each case in order for CaMeL to converge. Figure 3 shows the time in seconds CaMeL needed to converge in each of those cases[3]. As can be seen, the number of required plan traces and time is minimized when the probability P of a city having an airport is approximately 50%. For other values of P, methods are harder to learn. When P is close to 0, the hard-to-learn methods are those whose preconditions require cities to have airports, because the cases where these methods are applicable somewhere in given plan trances are so rare that learner can not easily induce their preconditions. When P is close to 1, the hard-to-learn methods are those whose preconditions do not require cities to have airports: the probability that there accidentally is an airport whenever these methods are applicable is so high that the learner cannot induce that an airport's presence is not indeed required.

Although it takes CaMeL tens and sometimes a few hundred training samples to learn all of the methods in the domain, CaMeL learns several of the methods in the NEO domain very quickly. Figure 4 shows how many of the methods are learned completely as a function of the number of plan traces, for the cases where $P = \frac{3}{12}, \frac{5}{12}, \frac{7}{12}, \frac{9}{12}$. From examining the raw data that went into this figure, we have observed that:

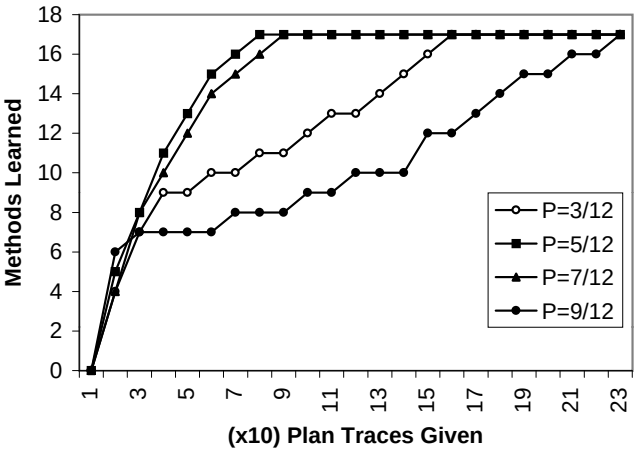

Figure 4: Speed of convergence for different training sets. The total number of methods in the domain is 17.

1. When P is close to 50%, all methods are learned very quickly.
2. When P is close to 0, methods whose preconditions do not require cities to have airports are learned very quickly.
3. When P is close to 1, methods whose preconditions require cities to have airports are learned very quickly.

We believe that these observations are quite important: When P is close to zero, methods that do not use airports are more likely to be used to decompose the tasks and are therefore of more importance than the other methods. The opposite is true when P is close to 1. In other words, CaMeL learns the most useful methods quickly, suggesting that CaMeL may potentially be of use in real world domains even if only a small number of training samples are available.

Related Work

Much of the work done on the integration of learning and planning is focused on conventional action-based planners. Usually, this work, as formulated in (Minton 1990), is aimed

[3]These experiments were conducted on a Sun Ultra 10 machine with a 440 MHz SUNW UltraSPARC-IIi CPU and 128 megabytes of RAM.

at speeding up the plan generation process or to increase quality of the generated plans by learning search control rules. These rules give the planner knowledge to help it decide at choice points and include *selection* (i.e., rules that recommend to use an operator in a specific situation), *rejection* (i.e., rules that recommend not to use an operator in a specific situation or avoid a world state), and *preference* (i.e., rules that indicate some operators are preferable in specific situations) rules. Generally speaking, the input for this kind of learning, as mentioned in (Langley 1996), consists of partial given knowledge of a problem-solving domain and a set of experiences with search through the problem's search space. The idea that this set of experiences can contain *solution paths* (or in our terminology, plan traces) were suggested in (Sleeman, Langley, & Mitchell 1982). In (Mitchell, Mahadevan, & Steinberg 1985), *learning apprentices*, which acquire their knowledge form observing a domain expert solving a problem, were suggested to be used as control rule learning algorithms for the first time. Explanation-Based Learning (EBL) has been used to induce control rules (Minton 1988). STATIC (Etzioni 1993) uses a graph representation of problem spaces to derive EBL-style control knowledge. Kautukam and Kambhampati (Kautukam & Kambhampati 1994), discuss the induction of explanation-based control rules in partial ordered planning. In (Leckie & Zukerman 1998), inductive methods are used to learn search control rules.

There has been some recent work on applying various learning algorithms in order to induce task hierarchies. In (Garland, Ryall, & Rich 2001), a technique called *programming by demonstration* is used to build a system in which a domain expert performs a task by executing actions and then reviews and annotates a log of the actions. This information is then used to learn hierarchical task models. *KnoMic* (van Lent & Laird 1999) is a learning-by-observation system which extracts knowledge from observations of an expert performing a task and generalizes this knowledge to a hierarchy of rules. These rules are then used by an agent to perform the same task.

Another aspect concerning the integration of planning and learning is *automatic domain knowledge acquisition*. In this framework, the planner does not have the full definition of the planning domain and tries to learn this definition by experimentation. In (Gil 1992; 1994), a dynamic environment in which the preconditions or effects of operators change during the time is introduced and methods to derive these preconditions and effects dynamically is discussed. In (Gil 1993), instead of revising existing operators, new operators are acquired by direct analogy with existing operators, decomposition of monolithic operators into meaningful suboperators and experimentation with partially-specified operators.

Several systems have integrated machine learning and planning before. For example, PRODIGY(Minton *et al.* 1989) is an architecture that integrates planning and learning in its several modules (Veloso *et al.* 1995). SCOPE (Estlin 1998) is a system which learns domain-specific control rules for a partial-ordered planner that improve both planning efficiency and plan quality (Estlin & Mooney 1997) and uses both EBL and Inductive Logic Programming (ILP) techniques. SOAR (Laird, Rosenbloom, & Newell 1986) is a general cognitive architecture for developing systems that exhibit intelligent behavior.

Conclusion and Future Work

In this paper, we introduced CaMeL, an algorithm that integrates machine learning techniques with HTN planning. Our ultimate goal is embedding CaMeL as a module in HICAP to help its users in the planning process. CaMeL is supposed to observe domain experts while they are solving instances of HTN planning problems, and gather and generalize information on how these experts solved these problems, so that it can assist other users in future planning problems. As our preliminary experiments suggest, CaMeL can quickly (i.e., with a small number of plan traces) learn the methods that are most useful in a planning domain. This suggests that CaMeL may potentially be useful in real-world applications, because it may be able to generate plans for many problems even before it has fully learned all of the methods in a domain.

CaMeL is an incremental algorithm. Therefore, even if has not been given enough training samples in order to converge, it should be able to approximate the methods that have not yet been fully learned. Our future work will include developing techniques to do these approximations. We also intend to integrate CaMeL into HICAP, and conduct subject study experiments with domain experts to obtain their judgments about the quality and validity of the generated plans.

Acknowledgments

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F306029910013 and F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, and the University of Maryland General Research Board. Opinions expressed in this paper are those of authors and do not necessarily reflect opinion of the funders.

References

Aha, D. W., and Breslow, L. A. 1997. Refining conversational case libraries. In *Proceedings of the Second International Conference on Case-Based Reasoning*, 267–278. Providence, RI: Springer Press/Verlag Press.

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

DoD. 1994. Joint tactics, techniques and procedures for noncombat evacuation operations. Technical Report Joint Report 3-07.51, Second Draft, Department of Defense.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI Press/MIT Press.

Estlin, T. A., and Mooney, R. J. 1997. Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1227–1232.

Estlin, T. 1998. *Using Multi-Strategy Learning to Improve Planning Efficiency and Quality*. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

Etzioni, O. 1993. A structural theory of explanation-based learning. *Artificial Intelligence* 60:93–139.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Garland, A.; Ryall, K.; and Rich, C. 2001. Learning hierarchical task models by defining and refining examples. In *First International Conference on Knowledge Capture*, 44–51.

Gil, Y. 1992. *Acquiring Domain Knowledge for Planning by Experimentation*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Gil, Y. 1993. Learning new planning operators by exploration and experimentation. In *Proceedings of the AAAI Workshop on Learning Action Models*.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*.

Gordon, D., and Desjardins, M. 1995. Evaluation and selection of biases in machine learning. *Machine Learning* 20:5–22.

Hanney, K., and Keane, M. T. 1996. Learning adaptation rules from a case-base. In *Proceedings of the Third European Workshop on CBR*. Lausanne, Switzerland: Springer.

Hirsh, H.; Mishra, N.; and Pitt, L. 1997. Version spaces without boundary sets. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. AAAI Press/MIT Press.

Hirsh, H. 1994. Generalizing version spaces. *Machine Learning* 17:5–45.

Kautukam, S., and Kambhampati, S. 1994. Learning explanation-based search control rules for partial-order planning. In *Proceedings of the 12th National Conference on Artificial Intelligence*.

Laird, J. E.; Rosenbloom, P. S.; and Newell, A. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1:11–46.

Lamber, K. S. 1992. Noncombatant evacuation operations: Plan now or pay later. Technical report, Naval War College.

Langley, P. 1996. *Elements of Machine Learning*. San Francisco, CA: Morgan Kaufmann Publishers.

Leckie, C., and Zukerman, I. 1998. Inductive learning of search control rules for planning. *Artificial Intelligence* 101:63–98.

Lenz, M.; Bartsch-Sporl, B.; Burkhard, H. D.; and Wess, S., eds. 1998. *Case-Based Reasoning Technology: From Foundations to Applications*. Berlin: Springer.

Minton, S. N.; Knoblock, C. A.; Kuokka, D. R.; Gil, Y.; Joseph, R. L.; and Carbonell, J. G. 1989. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

Minton, S. 1988. Learning effective search control knowledge: An explanation-based approach. Technical Report TR CMU-CS-88-133, School of Computer Science, Carnegie Mellon University.

Minton, S. N. 1990. Quantitative results concerning the utility of explanation based learning. *Artificial Intelligence* 42:363–391.

Mitchell, T. M.; Mahadevan, S.; and Steinberg, L. 1985. LEAP: A learning apprentice for VLSI design. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 573–580. Los Angeles, CA: Morgan Kaufmann.

Mitchell, T. M. 1977. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 305–310. Cambridge, MA: AAAI Press.

Mitchell, T. M. 1980. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University.

Muñoz-Avila, H.; McFarlane, D.; Aha, D. W.; Ballas, J.; Breslow, L. A.; and Nau, D. S. 1999. Using guidelines to constrain interactive case-based HTN planning. In *Proceedings of the Third International Conference on Case-Based Reasoning*, 288–302. Providence, RI: Springer Press.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 968–973. Stockholm: AAAI Press.

Sebag, M. 1995. 2nd order understandability of disjunctive version spaces. In *Workshop on Machine Learning and Comprehensibility, IJCAI-95*.

Sleeman, D.; Langley, P.; and Mitchell, T. M. 1982. Learning from solution paths: An approach to the credit assignment problem. *AI Magazine* 3:48–52.

van Lent, M., and Laird, J. 1999. Learning hierarchical performance knowledge by observation. In *Proceedings of the 16th International Conference on Machine Learning*, 229–238. San Francisco, CA: Morgan Kaufmann.

Veloso, M., and Carbonell, J. G. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning* 10:249–278.

Veloso, M.; Carbonell, J. G.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7:81–120.

Veloso, M.; Muñoz-Avila, H.; and Bergmann, R. 1996. Case-based planning: selected methods and systems. *AI Communications* 9:128–137.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational Intelligence* 6:232–246.