Learning Hierarchical Task Models from Input Traces

Chad Hogg
Dept. of Math. and Comp. Sci., King's College, Wilkes-Barre, PA, USA

Héctor Muñoz-Avila
Dept. of Comp. Sci. and Eng., Lehigh University, Bethlehem, PA, USA

Ugur Kuter
Smart Information Flow Technologies, Minneapolis, MN, USA

March 24, 2014

Abstract

We describe HTN-Maker, an algorithm for learning hierarchical planning knowledge in the form of task-reduction methods for Hierarchical Task Networks (HTNs). HTN-Maker takes as input a set of planning states from a classical planning domain and plans that are applicable to those states, as well as a set of semantically-annotated tasks to be accomplished. The algorithm analyzes this semantic information in order to determine which portion of the input plans accomplishes a particular task and constructs task-reduction methods based on those analyses.

We present theoretical results showing that HTN-Maker is sound and complete. Our experiments in five well-known planning domains confirm the theoretical results and demonstrate convergence toward a set of HTN methods that can be used to solve any problem expressible as a classical planning problem in that domain, relative to a set of goal types for which tasks have been defined. In three of the five domains, HTN planning with the learned methods scales much better than a modern classical planner.

Keywords: HTN Planning, Automated Planning, Machine Learning

1 Introduction

Automated planning systems typically require that a domain expert provide background planning knowledge about the dynamics of the planning domain. At a minimum, the background knowledge includes semantic descriptions (i.e., preconditions and effects) of possible actions, as in classical planning. More recent planning paradigms allow or require additional knowledge about the structural properties of the domain and about the potential problem-solving strategies for planning problems in the domain.

Over the years, *Hierarchical Task Networks (HTNs)* emerged to be one of the best-known approaches for modeling structural and problem-solving knowledge about a planning domain. An HTN planner formulates a plan via *task-reduction methods* (also known as simply *methods*), which describe how to reduce complex tasks into simpler subtasks until tasks are reached that correspond to actions that can be performed directly in the world. The HTN planner SHOP (Nau et al., 1999, 2003) demonstrated impressive gains in runtime performance over earlier classical planners. HTNs provide a natural knowledge-modeling framework in many real-world applications, including military planning (Mitchell, 1997; Muñoz-Avila et al., 1999), strategy formulation in computer games (Smith et al., 1998; Hoang et al., 2005), manufacturing processes (Nau et al., 1999), and story-telling (Cavazza and Charles, 2005).

HTN domain descriptions have other uses as well, such as project planning. Project planning is an endeavor to create products or to deliver services (Project Management Institute, 2013), and is used in a wide variety of activities including organizing public events, planning software engineering projects, and civil construction management. At its core, project planning involves the creation of hierarchical structures called work breakdown structures (WBS). WBSs are equivalent to HTNs (Muñoz-Avila et al., 2002) and HTN planning techniques can be used for project management (Xu and Muñoz-Avila, 2004).

Despite the great success of HTNs as a knowledge-modeling formalism, typically a significant knowledge engineering burden is required to write HTN domain descriptions of planning domains. To alleviate this burden, there have been several advances in automated learning of hierarchical knowledge for planning (Reddy and Tadepalli, 1997; Khardon, 1999; Choi and Langley, 2005; Ilghami et al., 2005; Xu and Munoz-Avila, 2005; Könik and Laird, 2006; Nejati et al., 2006, 2009; Könik et al., 2009). Most of these works require as input some structural knowledge about the world and the relationships among the activities to be accomplished in order to achieve the objectives. For example, the ICARUS family of learners (Choi and Langley, 2005; Nejati et al., 2006) use a hierarchy of *concepts*, which are Horn clauses that describe the relationships between goals and subgoals. The learner uses these concept definitions to chain together subgoals when creating nonprimitive skills that describe how to achieve goals. Another class of systems take as input goal annotations over possible task decompositions and use case-based reasoning (Xu and Munoz-Avila, 2005), version-space learning (Ilghami et al., 2005), or inductive learning (Könik and Laird, 2006) in order to produce the structure of HTN methods and their preconditions.

Currently, project plans are developed manually with the assistance of commercial tools such as Microsoft Project. However, the main difficulty is that task models for project planning are not available. Most knowledge is episodic (i.e., WBSs generated for previous projects). Our work aims at learning domain knowledge to generate HTNs from examples, and thus could be used to learn WBSs as well. As a result, our work could make producing such WBSs more efficient and thus be able to reduce costs for a large number of organizations that today perform this process manually.

In this paper, we describe HTN-Maker, an offline and incremental algorithm for learning HTN methods without requiring background knowledge about the hierarchical relationships among tasks and goals or problem-solving strategies. Even though such knowledge is not provided, HTN-Maker is capable of learning both the structural relationships between tasks and their subtasks as well as the conditions under which a task-reduction method may be applied to a task. In particular, our contributions in this paper are as follows:

- We describe a way to adopt the notion of task models from process-models literature (Murdock, 2001) in a new formalism. This formalism intuitively associates an activity with the conditions that must hold in the world such that it is possible to begin that activity and the effects that must be realized when the activity ends. HTN-Maker uses this formalism to learn the relationships among tasks, which leads to learning the structure of the HTN methods.

- We describe a formalism in which goal regression (Waldinger, 1977; Mitchell et al., 1986) may be applied hierarchically over actions and task-reduction methods and an algorithm, HTN-Maker, that uses this hierarchical goal regression to learn the applicability conditions of HTN methods and to identify their subtasks.

- We demonstrate an equivalence between classical planning problems and (some) HTN planning problems, and present a theoretical study showing that if an HTN planner using methods learned by HTN-Maker generates a plan for an HTN planning problem with an equivalent classical planning problem, then that plan is a solution to the equivalent classical planning problem. We also show that given a set of semantically annotated tasks for a domain and sufficient example traces from which to learn, HTN-Maker is capable of learning a set of HTN methods such that an HTN planner using those methods will be able to solve every problem expressible using those tasks.

- We present an extensive experimental evaluation of HTN-Maker in five benchmark planning domains from the past International Planning Competitions over several thousand planning problems. The hierarchical goal regression technique used by HTN-Maker is able to generalize well from specific training plan traces to general methods that are effective in planning. A reimplementation of SHOP using the learned HTNs was able to solve our experimental problems much faster than FastForward (Hoffmann and Nebel, 2001) and SGPlan6 (Hsu and Wah, 2008) for large problems in three of five planning domains, and is competitive in all but one.

2 Background

2.1 Classical Planning

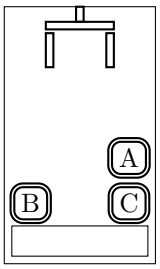
We adopt the usual definitions for classical planning as in Ghallab et al. (2004, Chapter 11). We summarize these definitions below.

We formalize a *classical planning domain description* as a tuple $\Sigma = (S, A, \gamma)$. S and A are the finite sets of all possible states and actions in the domain, respectively. A *state* is a conjunction of ground atomic formulas in predicate logic. An *action* has the form $a = (a^h, a^\phi, a^-, a^+)$, where the head of the action a^h is a grounded predicate and the

HTN-Maker is short for *Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required*

preconditions a^{ϕ}, negative effects a^{-}, and positive effects a^{+} of the action are conjunctions of atomic formulas that use only terms from the head of the action. When convenient, we will describe states and the preconditions and effects of actions in set-theoretic notation rather than logical notation (i.e., a state is a set of ground atomic formulas). For example, Figures 1 and 2 contain descriptions of a state and action, respectively, from the BLOCKS-WORLD domain.

A large number of actions are typically represented compactly by an *operator*, which has the same form as an action but is not required to be grounded. A variable substitution Θ creates an instance of an action from an operator by fully grounding it.




```
( and
  (on-table C) (on-table B)
  (on A C) (clear A)
  (clear B) (hand-empty) )
```

Figure 1: An example state from the BLOCKS-WORLD domain.

```
( :action !Unstack
  :parameters ( A C )
  :precondition
  ( and (on A C) (clear A) (hand-empty) )
  :effects
  ( and ( not (on A C) ) ( not (clear A) )
    ( not (hand-empty) ) (clear C) (holding A) ) )
```

Figure 2: An example action from the BLOCKS-WORLD domain.

In Σ, γ is the *state-transition function*: a partial function $S \times A \rightarrow S$. That is, given a state $s \in S$ and an action $a \in A$, if $s \models a^{\phi}$, then $\gamma(s, a) = (s \setminus a^{-}) \cup a^{+}$. Otherwise, $\gamma(s, a)$ is undefined. If $\gamma(s, a)$ is defined then we say that the action a is *applicable* in the state s; otherwise, a is not applicable in s.

A *plan* $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ is a sequence of actions. A plan $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ is *applicable* to a state s if $\gamma(s, a_1)$ is defined, $\gamma(\gamma(s, a_1), a_2)$ is defined, and each subsequent transition through $\gamma(\gamma(\ldots, \gamma(s, a_1), a_2), \ldots, a_k)$ is defined. As shorthand, we write $\gamma(s, \pi)$ for the state produced through this chain of transitions.

A *classical planning problem* is a triple $P = (\Sigma, s_0, g)$, where $\Sigma = (S, A, \gamma)$ is a classical planning domain, $s_0 \in S$ is the *initial state*, and g is a conjunction of ground atomic formulas known as the *goals* of the problem. A *solution* for the classical planning problem P is a plan $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ such that π is applicable to s_0 and the final state $\gamma(s, \pi)$ satisfies the goals g. A classical planning problem is *solvable* if it has a solution. We call the

Figure 2 uses the syntax of the well-known *Planning Domain Definition Language (PDDL)* (McDermott, 1998), which is also used with extensions for all other examples in this paper. In the PDDL language, negative effects are represented simply as negations of atomic formulas.

sequence of states produced by successively applying the actions in the plan starting from s_0 as the *state trajectory* $\vec{S}_\pi$ induced by the solution plan π. That is, if $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ is applicable to state s_0, it produces state trajectory $\vec{S}_\pi = \langle s_0, s_1, \ldots, s_{n+1} \rangle$, where $s_k = \gamma(s_0, \langle a_0, a_1, \ldots, a_{k-1} \rangle)$ for all $0 < k \leq n + 1$.

2.2 Hierarchical Task Network (HTN) Planning

In HTN planning, a *task* t is a symbolic representation of an activity in the world, usually represented as a logical predicate (Ghallab et al., 2004, Chapter 11). Formally, it is an expression of the form $(name\ arg_1\ arg_2\ \ldots\ arg_k)$ where *name* is a symbol denoting the name of the task. Each arg_i for $i = 1 \ldots k$ is either a variable or a constant symbol, denoting an *argument* of the task.

Let T be the finite set of *tasks* to be performed in a classical planning domain $\Sigma = (S, A, \gamma)$. T includes the head of every action in A, called the *primitive tasks*, as well as some additional tasks called *nonprimitive tasks*. A *task network* w is a sequence (i.e., a totally-ordered list) of tasks. The *empty* task network is $w = \langle\rangle$.

The form of HTN planning in which task networks are totally ordered is known as Simple Task Networks (STNs) (Ghallab et al., 2004, Chapter 11), and is used by planners such as SHOP (Nau et al., 1999). Other forms of HTN planning allow partially ordered or unordered task networks, but are beyond the scope of this paper. Following the lead of other authors, we will use the familiar, general term HTN even though we are writing specifically about STNs.

An HTN planner formulates a plan via *task-reduction methods* (in short, *methods*), which describe how to reduce complex tasks into simpler *subtasks* until tasks are reached that correspond to actions that can be performed directly in the world. Formally, a *method* is a triple $m = (m^h, m^\phi, m^w)$ where the *head* m^h is a nonprimitive task in T, the *preconditions* m^ϕ is a conjunction of atomic formulas, and the *subtasks* m^w is a task network into which the head task may be reduced. A method $m = (m^h, m^\phi, m^w)$ is *applicable* to a task network $w = \langle t_0, t_1, \ldots\ t_n \rangle$ in a state s, if there exists a variable substitution Θ such that $\Theta(m^h) = t_0$ and $s \models \Theta(m^\phi)$.

```
( :method Make-2Pile
  :parameters ( ?a ?b )
  :vars ( ?c )
  :precondition
  ( and (on-table ?b) (on ?a ?c)
    (clear ?a) (clear ?b)
    (hand-empty) )
  :subtasks
  < (!Unstack ?a ?c),
    (Make-2Pile ?a ?b) > )
```

```
( :method Make-2Pile
  :parameters ( ?a ?b )
  :precondition
  ( and (on-table ?b) (clear ?b)
    (holding ?a) )
  :subtasks
  < (!Stack ?a ?b) > )
```

Figure 3: Two example methods from the BLOCKS-WORLD domain.

As an example, Figure 3 shows a definition in an extended PDDL syntax of two methods that might be useful in the BLOCKS-WORLD domain. (In this language, the head of a method is divided into two parts: the name `Make-2Pile` and the parameters `?a` and `?b`. The language also allows us to specify a free variable `?c` that will be used in the parameters

and/or subtasks, which makes it easier to parse the language and find potential errors.) Throughout this paper, we follow the convention that the names of primitive tasks begin with an exclamation point and that the names of logical variables begin with a question mark. The first method shown in Figure 3 is applicable only in states where a block `?a` is clear and on a block `?c`, a block `?b` is clear, block `?b` is on the table, and the gripper is empty. Figure 1 shows one state in which this method is applicable, with the substitution {`?a/A, ?b/B, ?c/C`}.

One can define similar methods for the cases where `?a` is on the table, or `?a` is not clear, or `?b` is not clear or is on another block, or the gripper is not empty, or `?a` is being held, etc. The method on the left is insufficient to solve a problem by itself; because it is recursive, an additional method is needed to reduce its second subtask, such as the one on the right.

A plan $\pi = \langle a_0, a_2, \ldots, a_k \rangle$ *accomplishes* a task network $w = \langle t_0, t_1, \ldots, t_n \rangle$ in state s if any of the following cases can be shown to hold:

Case 0: If both the plan π and the task network w are of length 0.

Case 1: If the first task t_0 is primitive and is an exact match for the head of the first action in the plan, a_0 such that the action a_0 is applicable to state s, and if the successor task network $w' = \langle t_1, \ldots, t_n \rangle$ is accomplished by $\pi' = \langle a_1, a_2, \ldots, a_k \rangle$ in the successor state $s' = \gamma(s, a_0)$.

Case 2: If the first task t_0 is nonprimitive and there exists a method $m = (m^h, m^\phi, m^w)$ and a substitution Θ such that $t_0 = \Theta(m^h)$ and $s \models \Theta(m^\phi)$, such that the plan π accomplishes the reduced task network $w' = \Theta(m^w) \cdot \langle t_1, t_2, \ldots, t_n \rangle$ in state s.

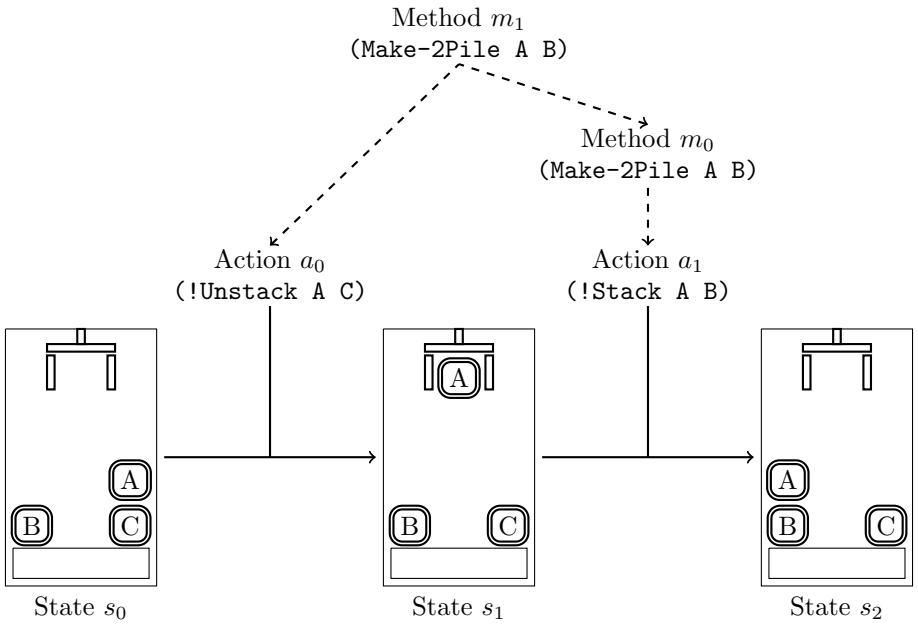


Figure 4: An example decomposition tree in the BLOCKS-WORLD domain.

The replacement of a nonprimitive task t by the subtasks of a method m is known as a *reduction* of the task t with m. A sequence of one or more reductions that results in a task network with no nonprimitive tasks is a *decomposition* of the task. The recursive structure containing the intermediate steps in the decomposition of a task is a *decomposition hierarchy* or *decomposition tree* (or *decomposition forest* if the initial task network contains more than one task). Each node in a decomposition forest represents a task, while each edge represents a reduction of a nonprimitive task into subtasks. The roots of a decomposition forest represent the initial task network. The leaves of a decomposition forest represent primitive tasks, which form a plan when read from left to right.

In Figure 4, `(Make-2Pile A B)` is reduced into $\langle$`(!Unstack A C)`, `(Make-2Pile A B)`$\rangle$, using method m_1 (from left of Figure 3). Within that task network, `(Make-2Pile A B)` is further reduced into $\langle$`(!Stack A B)`$\rangle$ using method m_0 (from right of Figure 3). These two reductions together create a decomposition of the top-level `(Make-2Pile A B)` into the task network $\langle$`(!Unstack A C)`, `(!Stack A B)`$\rangle$, which is also a plan. The decomposition tree itself contains only the nodes for the four tasks and the edges that are shown as dashed arrows; the states below are shown to aid in understanding.

An *HTN planning domain description* is a tuple $\Sigma^H = (S, A, T, M, \gamma)$, where S, A, and γ are the finite sets of states and actions and the state-transition function as in a classical planning domain description, T is a finite set of tasks including the heads of the actions in A, and M is a finite set of methods whose heads are members of T.

An *HTN planning problem* is a tuple $P^H = (\Sigma^H, s_0, w_0)$, where Σ^H is an HTN planning domain description, s_0 is the *initial state*, and w_0 is the *initial task network* (whose elements are members of T). A *solution* for the HTN planning problem P^H is a plan π that accomplishes the initial task network w_0 in the initial state s_0. We say that an HTN planning problem is *solvable* if there is a solution plan for it.

Algorithm 1 contains a high-level description of an HTN planner that searches for a solution plan using the three cases defined above. Both SHOP and our reimplementation HTN-Solver are based on this procedure.

2.3 Annotated Tasks

As described above, a task t in HTN planning is a symbolic representation of an activity. Consequently, t does not specify any information about the meaning of that activity: what purpose the activity serves, under what conditions it can start, or under what conditions it ends. Existing HTN formalisms that associate semantics to tasks (Erol et al., 1996; Nau et al., 2003; Tate, 1977) typically do so through the methods that reduce those tasks. Because we seek to learn the methods themselves, our formalism requires that tasks have inherent semantics.

Several previous works have described similar formalisms; examples include activity representation in software engineering (Sutton et al., 1995), user interactions for AI planning systems (Fdez-Olivares et al., 2006), adaptive agent systems (Ulam et al., 2005), hierarchical planning and plan adaptation (Kambhampati and Hendler, 1992; Biundo and Schattenberg, 2001), and introspective, self-reasoning agents (Murdock, 2001). In our work, we have adopted the formalism of the *task-model language* TMKL (Murdock, 2001). In TMKL, tasks indicate what they accomplish by stating their conditions and effects: if the conditions are true in a state of the world from which the task is accomplished, then the effects must be true in the resulting world state.

Formally, we define an *annotated task* as a tuple $\tau = (\tau^h, \tau^\phi, \tau^+)$, where the *head* τ^h is

Algorithm 1: A high-level description of a straightforward HTN planner. The input is an HTN planning problem (Σ^H, s_0, w_0). The output is a solution plan or FAILURE.

Procedure HTN-SOLVER($\Sigma^H = (S, A, T, M, \gamma), s_0, w_0 = \langle t_0, t_1, \ldots, t_n \rangle$)
begin
  if $w_0 = \langle\rangle$ **then** // Case 0
    return $\langle\rangle$
  if t_0 *is primitive* **then**
    $\alpha = \{a | a \in A, t_0 = a^h, s \models a^\phi\}$ // Case 1
    if $\alpha \neq \emptyset$ **then**
      nondeterministically select $a \in \alpha$
      return $\langle a \rangle \cdot$ HTN-SOLVER($\Sigma^H, \gamma(s_0, a), \langle t_1, \ldots, t_n \rangle$)
    else
      return FAILURE
  else
    $\alpha = \{\Theta(m) | m \in M, t_0 = \Theta(m^h), s \models \Theta(m^\phi)\}$ // Case 2
    if $\alpha \neq \emptyset$ **then**
      nondeterministically select $\Theta(m) \in \alpha$
      return HTN-SOLVER($\Sigma^H, s_0, \Theta(m^w) \cdot \langle t_1, \ldots, t_n \rangle$)
    else
      return FAILURE
end

a nonprimitive task, the *preconditions* τ^ϕ is a conjunction of atomic formulas, and the *postconditions* τ^+ is a conjunction of atomic formulas. The preconditions of an annotated task represent those facts that must hold in order for it to be possible to attempt that task, while the postconditions represent those facts that must become true as a result of accomplishing that task. Given a sequence of actions $\langle a_p, a_{p+1}, \ldots, a_{q-1}, a_q \rangle$ and a corresponding state trajectory $\langle s_{p-1}, s_p, \ldots, s_{q-1}, s_q \rangle$, an annotated task τ is accomplished by the action trace if its preconditions τ^ϕ are satisfied in the state s_{p-1} and its postconditions τ^+ are satisfied in the state s_q. There are no negative postconditions of an annotated task, as there are no negative goals of a classical planning problem.

```
( :task Make-2Pile
  :parameters ( ?a ?b )
  :precondition
  ( and )
  :postcondition
  ( and (on-table ?b) (on ?a ?b) (clear ?a) ) )
```

Figure 5: An example annotated task in the BLOCKS-WORLD domain.

Figure 5 shows a definition of an annotated task from the BLOCKS-WORLD domain. This task can be attempted from a state in which the preconditions (none) are satisfied and has been accomplished in a state in which the postconditions are satisfied. Note that a method reducing this task may have additional applicability conditions that specify the conditions under which the task is to be accomplished with the subtasks specified in the

method. However, those applicability conditions are not necessarily part of the semantics of the task; they are specific to the particular way of accomplishing the task with that method and may differ from one possible method for the task to another. Similarly, each particular method for accomplishing this task is likely to also produce side effects in addition to the required postconditions. Because there are no preconditions, this particular task may be attempted from any state, such as the state of Figure 1 with substitution {`?a/A, ?b/B`}, or {`?a/B, ?b/C`}, etc. Figure 3 shows two of the many possible methods for accomplishing the task.

The notion of an annotated task enables us to define an equivalence between a task and a set of goals, and consequently, between a classical planning problem and an HTN planning problem. Given a goal statement g, we define the *equivalent annotated task* as $\tau_g = (\tau^h, \emptyset, g)$, where τ^h is an arbitrary nonprimitive task that uniquely represents g. Then, given a classical planning problem $P = (\Sigma, s_0, g)$ and annotated task $\tau_g = (\tau^h, \emptyset, g)$ equivalent to its goals, we define $P^H = (\Sigma^H, s_0, \langle \tau^h \rangle)$ to be an *equivalent HTN planning problem*. The sets of states and actions and the state transition function of the HTN planning domain description Σ^H are the same as those of Σ. The set of tasks in Σ^H contains each of the primitive tasks and τ^h (and may also contain other tasks). Given a classical planning problem P there might be multiple different equivalent HTN planning problems P^H, each with different sets of tasks and methods. Many of those equivalent HTN planning problems may be unsolvable. (Consider, as a trivial example, one in which the set of methods is empty.)

Because the motivation for our work is to automate the creation of HTN methods with very little manual knowledge engineering, our experiments thusfar have used very simple annotated tasks that have no preconditions and whose postconditions correspond directly to problem goals. See Appendix D for the details of these annotated tasks used in our experiments. Our formalism allows the use of more complex annotated tasks (with a carefully chosen set of harmonious postconditions, or with a non-empty precondition set, for example), and it remains as future work to explore whether or not additional human effort would improve results.

By introducing annotated tasks we have not changed the semantics of HTN planning, since the annotations are not used by the HTN planners explicitly. Instead, they are implicit in that the annotated tasks will be used by HTN-Maker to learn methods, which are then given to the HTN planners to generate plans in the usual way. The learned methods are constructed in such a manner that plans generated will be guaranteed to satisfy the conditions of the annotated tasks.

3 Learning HTNs from Solution Plans For Semantically-Annotated Tasks

In this section, we describe HTN-Maker (Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required), a novel offline incremental learning algorithm that learns a set M of HTN methods from an input set of planning states and plans applicable to them and successively updates M with new methods it learns when presented with new plans from the same planning domain.

The idea behind the algorithm is that plans provide demonstrations of how an annotated task might be accomplished. Moreover, in some cases they may show not simply an acceptable way to accomplish a task, but what the creator of the plan considers to be the best way to do so. However, accomplishing a certain task does not need to have been the

objective of a plan in order for HTN-Maker to learn to do so from that plan; plans may coincidentally accomplish tasks, and HTN-Maker will still extract knowledge from them. Thus, HTN-Maker performs Explanation-Based Learning (Minton et al., 1989): using logical inference to explain how an example (a plan) is an instance of a high-level concept (accomplishment of a task).

In order to learn from a plan, it is necessary to determine which part of a plan (perhaps the entire plan, but perhaps not) accomplishes a particular task. Our main algorithm, HTN-Maker, does this by evaluating the preconditions and postconditions of an annotated task in the different states that result from execution of a plan. Section 3.2 explains this algorithm in detail.

Once it has been determined that a (sub-)plan accomplishes a task, the next step is to determine how the task was accomplished and to create a method structure that encapsulates the strategy and that can be used by an HTN planner to use that strategy for accomplishing the task in other situations. This means creating an ordered list of subtasks that represent the activities taken to accomplish the task and a set of preconditions that must be true in order for this to be an appropriate strategy for accomplishing the task. A sub-algorithm, Learn-Method, uses a novel generalization of goal regression known as hierarchical goal regression to find an appropriate subtask list and precondition set and creates a method based on them. Section 3.3 explains this algorithm in detail.

If the subtasks of the methods learned by HTN-Maker were always primitive tasks corresponding directly to actions in the plan we would simply be learning macro-operators, which other researchers have already done (Mooney, 1988; Botea et al., 2005). Instead, some subtasks should be nonprimitive tasks that must be further reduced, creating the hierarchical structures typical of HTN planning. To accomplish this, HTN-Maker examines subplans in a particular order and maintains information about nonprimitive tasks that had been accomplished by sub-sub-plans, so that those nonprimitive tasks may be used as subtasks of methods that will be learned later.

3.1 Example

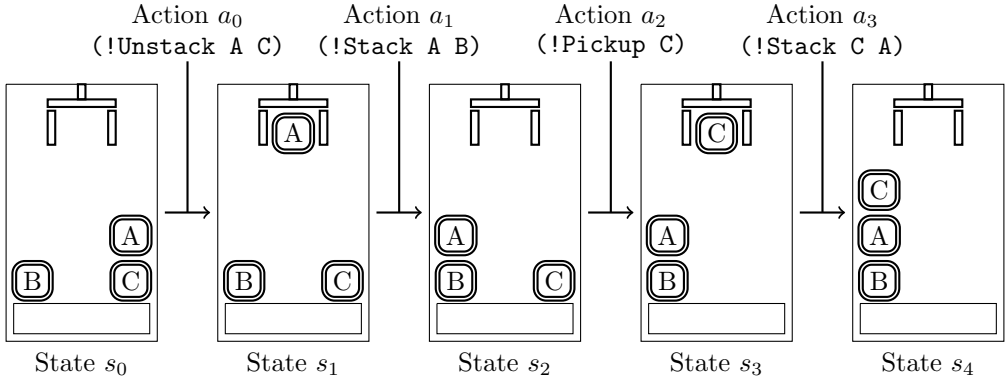


Figure 6: An example plan in the Blocks-World domain.

Before we explain the details of the algorithm, we will show an example of its execution

in the familiar BLOCKS-WORLD domain. Figure 6 shows an initial state s_0 (the same one from Figure 1), a plan of four actions a_0 through a_3 applied to that state, and the rest of the resulting state trajectory. Suppose that there exist annotated tasks for building piles of sizes 1 through 3, where the blocks in a pile are listed from top to bottom, the top block is clear, and the bottom block is on the table. The annotated tasks for size-1 piles and size-3 piles would be analogous to the annotated task for size-2 piles, which was shown in Figure 5.

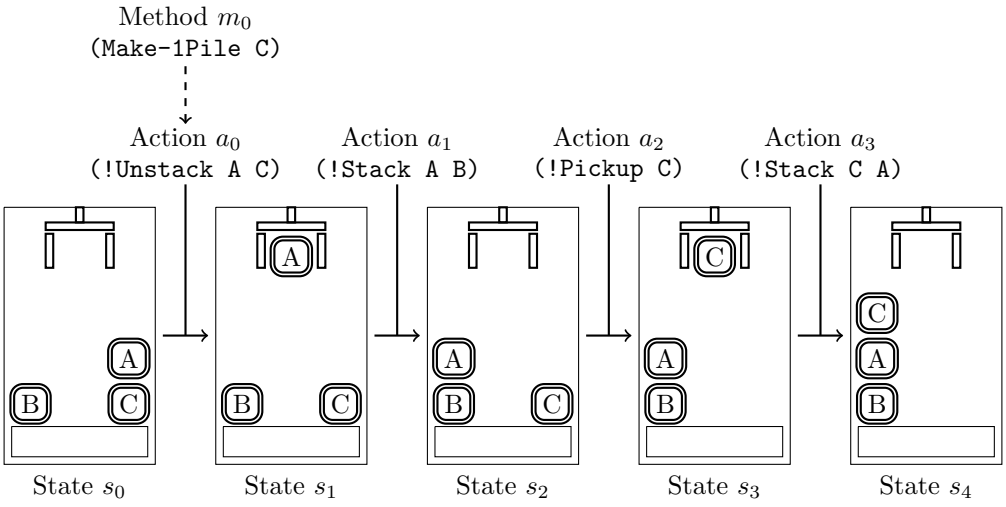


Figure 7: A method that could be learned from the example plan.

HTN-MAKER would begin by considering the subplan containing only action a_0. It would check whether or not there is an annotated task whose preconditions are satisfied in s_0 and whose postconditions are satisfied in s_1 but not in s_0, and would find one: `(Make-1Pile C)`. Thus, the LEARN-METHOD subroutine would learn a method (labeled m_0 on Figure 7) to explain how that task was accomplished. Method m_0 has only one subtask, a_0. It has several preconditions: `(on-table C)`, `(on A C)`, `(clear A)`, and `(hand-empty)`, which were found by regressing the postconditions of the annotated task through a_0.

Next, HTN-MAKER would look at the subplan containing only a_1. This short subplan accomplished a different task: `(Make-2Pile A B)`. So another method (m_1 in Figure 8) would be learned in a similar fashion. This happens to be the second one shown in Figure 3. Then HTN-MAKER would consider the subplan containing both a_0 and a_1. It would find that this larger subplan accomplishes the task `(Make-2Pile A B)` as well, and would thus learn another method m_2. This method has two subtasks: first a_0 and then the nonprimitive task `(Make-2Pile A B)`, which happens to be the same task as the head of the method. Recursive structures like this are not uncommon in the methods learned by HTN-MAKER, and represent the fact that after part of the work has been done by the first subtask there is a new state in which the system may already know how to accomplish the larger task. The preconditions of m_2 will be the postconditions of `(Make-2Pile A B)` regressed first through method m_1 and then through action a_0. Method m_2 happens to be the first one shown in Figure 3. Note that there are now two methods for accomplishing `(Make-2Pile A B)`, which are applicable in different circumstances and which can be used together or

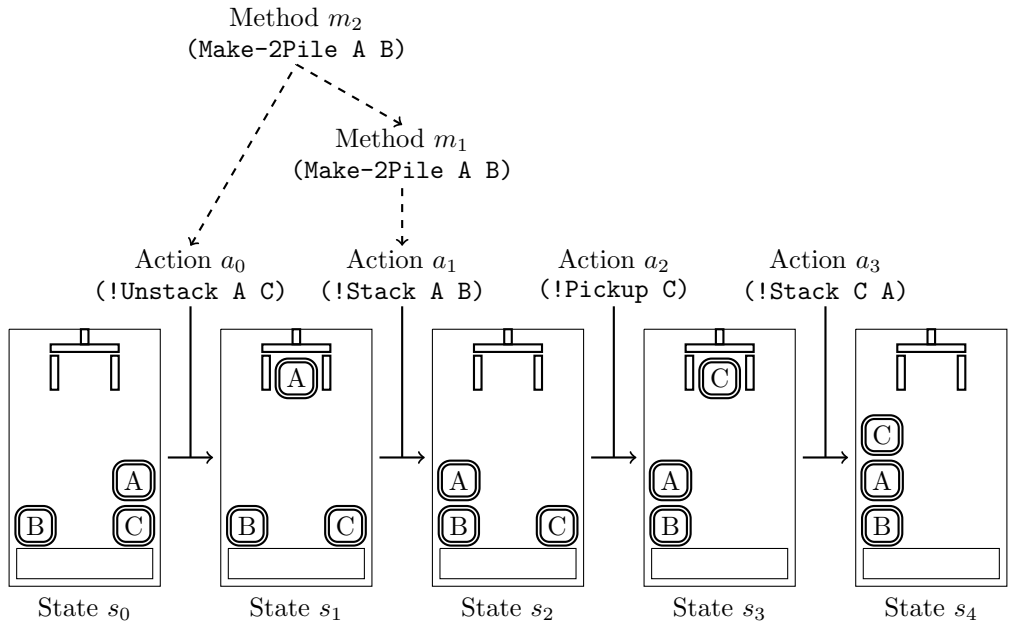


Figure 8: Two additional methods that could be learned from the example plan.

independently.

HTN-Maker next considers the subplan $\langle a_2 \rangle$, but finds nothing interesting is accomplished by that subplan, and finds the same thing to be true of subplans $\langle a_1, a_2 \rangle$ and $\langle a_0, a_1, a_2 \rangle$. The subplan containing only a_3, however, accomplishes `(Make-3Pile C A B)`. HTN-Maker thus learns a new method (m_3 in Figure 9). It then considers ever larger subplans and finds that each one accomplishes this task, and thus learns methods m_4, m_5, and m_6 as well, using the same sort of logic. Thus, from this single five-action plan HTN-Maker learns one way to create a pile of 1 block, two ways to create a pile of 2 blocks, and four ways to create a pile of 3 blocks.

The HTN-Maker algorithm contains a nondeterministic choice that determines what subtasks to give a method when there are several possibilities. This example shows what would happen if that nondeterministic choice were made in one particular way, which corresponds to the choices made in our experiments. (See Section 5.1 for details.) If that nondeterministic choice were made in a different way a different set of methods would have been learned from this example, creating a different hierarchical structure. For example, it might instead have learned to create the pile `C-A-B` by first creating the pile `A-B`, then picking up `C`, and finally stacking `C` on `A`.

For the sake of simplicity this example and the algorithms shown in Sections 3.2 and 3.3 show the creation of methods whose terms are constants. The methods that HTN-Maker learns actually use variables, and Section 3.4 explains how the example and algorithms are extended to accomplish this.

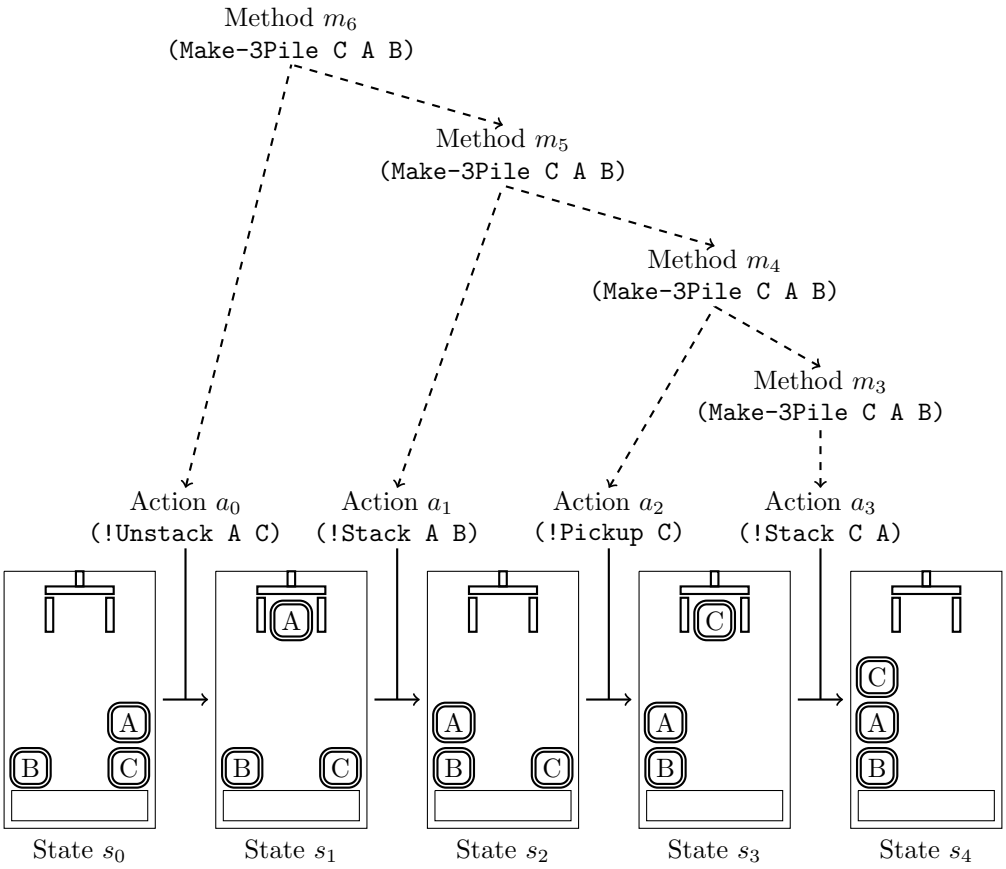


Figure 9: Four additional methods that could be learned from the example plan.

3.2 The Main Algorithm

A *learning example* $e = (s_0, \pi)$ is a pair such that s_0 is a state and π is a plan applicable to that state. The input to HTN-Maker includes a classical planning domain Σ, a finite set of learning examples E, a finite set of annotated tasks $\mathcal{T}$, and a set of existing HTN methods M for the tasks annotated in $\mathcal{T}$. In a typical run of HTN-Maker M would be the empty set, but it can also contain some methods that had been previously learned or hand-crafted.

The HTN-Maker algorithm uses a data structure that we call an *indexed method instance*. An *indexed method instance* is a sextuple $x = (x^h, x^+, x^w, x^\phi, x^b, x^e)$ in which x^h is the head of an annotated task τ for which a method has been learned, x^+ is the postconditions of that annotated task, x^w is a task network into which x^h may be reduced, x^ϕ is a set of preconditions under which that reduction is valid, and x^b and x^e are indices in a state trajectory. This indicates that the subtasks are a demonstration of how the annotated task was accomplished within the subplan between states x^b and x^e. Thus, x^ϕ must hold in s_{x^b} and x^+ must hold in s_{x^e}. Indexed method instances are computed automatically by HTN-Maker from the annotated tasks and learning examples.

Algorithm 2: A high-level description of the HTN-Maker procedure. The input includes a classical planning domain description Σ, a set E of learning examples, a set of annotated tasks $\mathcal{T}$, and a set of HTN methods M. The output is an updated set of HTN methods.

Procedure HTN-Maker($\Sigma, E, \mathcal{T}, M$)
begin
  $M \leftarrow M \cup$ Make-Trivial-Methods($\mathcal{T}$)
  $M \leftarrow M \cup$ Make-Verification-Methods($\mathcal{T}$)
  foreach *learning example* $e = (s_0, \pi = \langle a_0, \ldots, a_k \rangle) \in E$ **do**
    initialize $X \leftarrow \emptyset$
    initialize $\vec{S}_\pi \leftarrow \langle s_0 \rangle$
    for $i \leftarrow 1$ *to* k **do** `// Generate state trajectory`
      $s_i \leftarrow \gamma(s_{i-1}, a_{i-1})$; $\vec{S}_\pi \leftarrow \vec{S}_\pi \cdot \langle s_i \rangle$
    for $f \leftarrow 1$ *to* k **do** `// End of sub-trajectory`
      for $i \leftarrow f - 1$ *down to* 0 **do** `// Beginning of sub-trajectory`
        foreach *annotated task* $\tau = (\tau^h, \tau^\phi, \tau^+) \in \mathcal{T}$ **do**
          if $s_i \models \tau^\phi$ *and* $s_f \models \tau^+$ *and* $s_i \not\models \tau^+$ **then** `// Task accomplished`
            $m \leftarrow$ Learn-Method($\pi, \vec{S}_\pi, \tau, X, i, f$) `// Learn new method`
            $M \leftarrow M \cup \{m\}$ `// Store new method`
            $X \leftarrow X \cup \{(m^h, \tau^+, m^w, m^\phi, i, f)\}$ `// Store instance`
  return M
end

Algorithm 2 shows a high-level description of HTN-Maker. Before processing each of the learning examples, HTN-Maker calls two subroutines to create methods that do not require a demonstration from which to learn. The Make-Trivial-Methods subroutine creates (if they do not already exist) a *trivial* method for each annotated task, which can be used to "accomplish" a task that can accomplished without doing anything at all because its postconditions are already true. Specifically, if $\tau = (\tau^h, \tau^\phi, \tau^+)$ is an annotated task, the

trivial method for τ is $m = (\tau^h, \tau^\phi \cup \tau^+, \langle\rangle)$. These methods are needed so that an HTN planner will be able to produce the empty plan when the empty plan is a valid solution to a (sub-)problem.

The MAKE-VERIFICATION-METHODS subroutine creates (if they do not already exist) a *verification* method for each annotated task. If $\tau = (\tau^h, \tau^\phi, \tau^+)$ is an annotated task, the verification method for τ is $m = (t', \tau^+, \langle\rangle)$. Verification methods and trivial methods are very similar, but there is an important difference: the head of a verification task is not the head of the annotated task. Instead, it is a new task symbol t' created uniquely for that annotated task, known as a *verification task*. Verification tasks do not have annotations, and no methods will be learned to reduce them. Instead, the single verification method for an annotated task may be used to reduce the associated verification task into the empty task network when the annotated task has been accomplished. The last subtask in every method that we learn will be a verification task, to ensure that the postconditions of the task really were accomplished. To simplify presentation, this last subtask of each learned method is omitted from figures in this paper. The necessity of verification tasks and methods is explained in Section 4.1 and Appendix C.

HTN-MAKER then processes each learning example in turn. In Line 6, HTN-MAKER initializes an empty set X of indexed method instances. In Lines 7 through 9 it generates the state trajectory $\vec{S}_\pi$ induced by the plan in the current learning example. HTN-MAKER then considers each subsequence of states, s_i through s_f, from the state trajectory (Lines 10–11). Note that these subsequences are processed in a specific order such that when any subsequence is being considered all of *its* subsequences have already been processed in earlier iterations. This ensures that HTN-MAKER learns new HTN methods in a bottom-up fashion, effectively building a possible HTN decomposition hierarchy above the plan.

In Line 12, HTN-MAKER considers each annotated task for each possible subsequence of the state trajectory. If the first state, s_i, and the last state, s_f, of the subsequence satisfy the preconditions and postconditions of the annotated task, respectively, then that task has been accomplished by the plan that corresponds to this state subsequence (Line 13). We skip over situations in which the postconditions of an annotated task were satisfied in both s_i and s_f because in these cases the subsequence does not actually demonstration accomplishment of the task. When a task has been accomplished, HTN-MAKER calls its LEARN-METHOD subroutine to learn a new method that describes how the task was accomplished (Line 14). The LEARN-METHOD subroutine is shown in Algorithm 3, and will be explained in the subsequent section. HTN-MAKER adds the new method that LEARN-METHOD returns to the list of methods (Line 15).

In Line 16, HTN-MAKER stores information about the method that has just been learned in an indexed method instance. This will be used in successive calls to the LEARN-METHOD subroutine to allow complex hierarchies to be learned.

3.3 Hierarchical Goal Regression

We now describe our *hierarchical goal regression* technique, which is the basis of our learning procedure for both the structure (i.e., task-subtask relationships) in an HTN method and its preconditions. Unlike previous work on goal regression (Mitchell et al., 1986), hierarchical goal regression can regress goals both horizontally (through the primitive actions) and vertically (up the task hierarchy through indexed instances of previously-learned HTN methods).

In hierarchical goal regression, a formula can be regressed over either a primitive action

or an indexed method instance for a nonprimitive task. In the case of the former, the regression is performed using the preconditions and effects of the action in the same manner as traditional goal regression. In the case of the latter, the regression is performed over the postconditions of the annotated task and the preconditions of the method learned for that task.

The idea of goal regression is that we can find some set of atoms g' such that, if we reach a state s' in which they hold, we know a procedure to transform s' into a state s in which our goals g hold. In classical goal regression the procedure would be a plan, but in our case it is a task network. Given a set of goals g and a task network w, we can find the set $g' = \mathcal{R}(g, w)$ with the regression operator $\mathcal{R}$ (Reiter, 1991), modified to support tasks. The value of $\mathcal{R}(g, w)$ is the minimal set of atoms that must be true in a state s' to guarantee that w will be decomposable resulting in a plan that produces a state s where g holds:

- If w is the empty task network, then $\mathcal{R}(g, w) = g$.
- If w contains a single primitive task t, which corresponds to the action $a = (t, a^{\phi}, a^{-}, a^{+})$, then $\mathcal{R}(g, w) = (g \setminus a^{+}) \cup a^{\phi}$.
- If w contains a single nonprimitive task t for which we have an indexed method instantiation $x = (t, x^{+}, x^{w}, x^{\phi}, x^{b}, x^{e})$, then $\mathcal{R}(g, w) = (g \setminus x^{+}) \cup x^{\phi}$.
- If w contains two or more tasks $\langle t_0, t_1, \ldots, t_n \rangle$, then $\mathcal{R}(g, w) = \mathcal{R}(\mathcal{R}(g, \langle t_n \rangle), \langle t_0, t_1, \ldots, t_{n-1} \rangle)$.

Note that a particular indexed method instantiation may have incidental side effects that serendipitiously achieve a goal. These side effects are not used in regression because there is no guarantee that they would occur in other situations (in which a task that could be reduced with the same method observed in the example is instead reduced with a different, equally legal method that has different side effects).

Algorithm 3 shows a high-level description of our hierarchical goal regression procedure, called Learn-Method. Intuitively, the algorithm works backward through a given subplan, maintaining a set of open conditions (initially the postconditions of the annotated task τ). At each step, it nondeterministically chooses between the actions and/or indexed method instances whose effects provide an open condition, regresses the open conditions through that action or indexed method instance to create a new set of open conditions, and prepends the chosen action or indexed method instance to a list of subtasks for the new method being learned. When it reaches the beginning of the subplan, the algorithm creates a new method whose preconditions are the remaining open conditions and whose subtasks are those actions and indexed method instances that were chosen.

In Line 3, Learn-Method first initializes the set of open conditions to the postconditions of the annotated task. The open condition set represents those conditions that an indexed method instance or action could cause to become true to assist in accomplishing the task. Learn-Method also initializes a task network that will represent the subtasks of the method to be learned in Line 4 and a current state in Line 5. The single element of the task network is the verification task t' associated with the annotated task τ.

The main loop of Learn-Method iterates the current state backwards through the relevant section of the state trajectory, sometimes by steps (Line 20) and others by leaps (Line 18). The loop begins by initializing an empty set of potential subtasks (Line 7). An indexed instance x of a previously-learned method is a potential subtask if it meets three criteria: the ending index of the instance must be the current state, the beginning index of

Algorithm 3: The LEARN-METHOD procedure that performs hierarchical goal regression over HTNs. The inputs are a plan π, a state trajectory induced by that plan $\vec{S}_\pi$, an annotated task τ, a set X of indexed method instances generated from previous calls to this same algorithm with the same plan, and indices for an initial state i and final state f. The output is a new method m.

Procedure LEARN-METHOD($\pi, \vec{S}_\pi, \tau, X, i, f$)
begin
  $\phi \leftarrow \tau^+$ // Initialize open conditions
  $w \leftarrow \langle t' \rangle$ // Initialize subtask list
  $c \leftarrow f$ // Initialize current state index
  while $c > i$ **do** // Step current state back to initial
    $X' \leftarrow \emptyset$ // Initialize useful instances
    foreach $x = (x^h, x^+, x^w, x^\phi, x^b, x^e) \in X$ **do**
      if $x^e = c \wedge x^b \geq i \wedge x^+ \cap \phi \neq \emptyset$ **then** // Fits and provides an open
        $X' \leftarrow X' \cup \{x\}$ // cond., so its task could be a subtask
    $a_c \leftarrow$ the c-th action in π
    if $a_c^+ \cap \phi \neq \emptyset$ **then** // Current action provides an open
      $X' \leftarrow X' \cup \{(a_c^h, a_c^+, \langle\rangle, a_c^\phi, c-1, c)\}$ // cond., so could be subtask
    if $X' \neq \emptyset$ **then**
      nondeterministically select an instance $x = (x^h, x^+, x^w, x^\phi, x^b, x^e) \in X'$
      $\phi \leftarrow (\phi \setminus x^+) \cup x^\phi$ // Regress open conditions across subtask
      $w \leftarrow \langle x^h \rangle \cdot w$ // Prepend subtask to list
      $c \leftarrow x^b$ // Update current state
    else
      $c \leftarrow c - 1$ // Step backward in plan
  return $m = (\tau^h, \phi \cup \tau^\phi, w)$ // Create new method
end

the instance must be no earlier than the initial state of this subplan, and the postconditions of the instance must contain an open condition (Lines 9–10). The action directly before the current state is a potential subtask if its positive effects contain an open condition (Lines 12–13). It cannot strictly form an indexed method instance, but storing information about it in the same data structure simplifies the presentation of the algorithm.

If there are any potential subtasks, one is nondeterministically selected (Line 15). The subtask selected will determine the structure of the HTNs generated by the learned methods. Section 5.1 discusses the details of the criterion used to make this selection. The open conditions are regressed through the selected indexed method instance in Line 16, the head of the subtask is prepended to the subtask HTN in Line 17, and the current state is moved backward to before this subtask began in Line 18.

If there are no potential subtasks (that is, the previous action was not useful to accomplishing the task), then that action is skipped and the current state moved back by one (Line 20).

At the end of the main loop, the LEARN-METHOD subroutine returns a new method with the same head as the annotated task that was accomplished (Line 21). The preconditions of this new method include the preconditions of the annotated task and the regressed open conditions, and its subtasks are the actions and the heads of the indexed method instances through which they were regressed.

3.4 Generalization of the Learned Methods

The actions in a plan π of a learning example are entirely grounded. On the other hand, the terms in an HTN method are generally variables, and a substitution is applied to ground the method when it is used to reduce a task. We now explain how we generalize each grounded method into a method containing variables. We will explore two different ways to perform generalization. In Section 5, we report how these two versions compare in a variety of domains.

At nearly every step of the LEARN-METHOD algorithm a substitution must be maintained to keep track of the relationships between terms from different subtasks. When a new subtask is added (Lines 16-17 of Algorithm 3), the variables used in the action or indexed method instance that is being added are standardized apart from any variables currently used in the method being built. We consider two possible techniques for determining when a variable used in the new subtask should be unified with a variable that is already in use for the method being built.

3.4.1 Weak Generalizations

In this formalism, variables from the new subtask are only unified with variables already in use if they appear in a positive effect of the new subtask that matches an open condition. For example, if `(on ?a ?b)` is in the open condition set with substitution {`?a/B12, ?b/B3`} and `(on ?x ?y)` is a positive effect of the subtask being added with substitution {`?x/B12, ?y/B3`}, then `?a` will be unified with `?x` and `?b` will be unified with `?y`. Otherwise, variables are not unified and an HTN planner using the method will be free to instantiate it with a substitution that maps each of them to the same constant or to different constants. For example, if `?c`, `?d`, and `?e` are other variables currently in use for the new method with

We use the term generalization as it is frequently used (e.g., see Bergmann and Wilke (1995)). This term describes the process of replacing constants with variables in an object.

substitution {?c/B5, ?d/B7, ?e/B1} and ?z is another variable used by the new subtask with substitution {?z/B7}, then ?d will not be unified with ?z (unless ?z is part of a positive effect of the subtask that matches an open condition that ?b is a part of, as described above).

3.4.2 Strong Generalizations

In *strong generalization* every constant used in the subtasks of a method is mapped to a single variable in that method and in planning no two variables may refer to the same constant. For example, if ?a, ?b, ?c, ?d, and ?e are variables currently in use for the new method with substitution {?a/B12, ?b/B3, ?c/B5, ?d/B7, ?e/B1} and ?x, ?y, and ?z are variables used in the new subtask with substitution {?x/B12, ?y/B3, ?z/B7}, then ?a will be unified with ?x, ?b will be unified with ?y, and ?d will be unified with ?z. This reduces the applicability of the learned method, but may in so doing prevent the method from being used in unhelpful ways.

```
( :method Deliver
  :parameters ( ?p ?y )
  :vars ( ?x ?z )
  :precondition
  ( and
    (is-package ?p)
    (is-location ?x)
    (is-location ?y)
    (is-location ?z)
    (is-truck ?t)
    (package-at ?p ?x)
    (truck-at ?t ?z) )
  :subtasks
  < (!Drive-Truck ?t ?z ?x),
    (Deliver ?p ?y) > )
```

```
( :method Deliver
  :parameters ( ?p ?y )
  :vars ( ?x )
  :precondition
  ( and
    (is-package ?p)
    (is-location ?x)
    (is-location ?y)
    (is-truck ?t)
    (package-at ?p ?x)
    (truck-at ?t ?y)
    (different ?x ?y) )
  :subtasks
  < (!Drive-Truck ?t ?y ?x),
    (Deliver ?p ?y) > )
```

Figure 10: Two example methods that could be learned in the LOGISTICS domain.

The difference between weak and strong generalizations is demonstrated in the two similar methods of Figure 10. These methods are from the LOGISTICS domain, where trucks and airplanes are used to deliver packages to various locations among several cities. The first uses weak generalization, while the second uses strong generalization. Both deliver a package to a location first by driving a truck to where the package is, then recursively trying to deliver (presumably by loading it into the truck, driving the truck to the destination, and unloading it there). The subtle difference is that the first drives the truck from a location ?z, which has no constraints other than that it be a location and that the truck be located there, while the second drives the truck from location ?y, which is also used in the head of the method. Thus, the second method will only be applicable when the truck happens to start in the package's desired destination, while the first will work regardless of where the truck begins.

3.4.3 Method Subsumption

Once the set M of learned methods are generalized to include variable symbols, it might be the case that a generalized method $m \in M$ *subsumes* another method $m' \in M$. Intuitively, this means that in every case where m' is applicable, m will also be applicable and have exactly the same results.

We say that a method m_1 subsumes another method m_2 when there is a substitution that could be applied to one such that they will have identical heads and subtasks, and the preconditions of m_2 will imply the preconditions of m_1. Formally, method $m_1 = (m_1^h, m_1^\phi, m_1^w)$ subsumes method $m_2 = (m_2^h, m_2^\phi, m_2^w)$ if there exists a substitution Θ such that each of the following are true:

1. $m_2^h = \Theta(m_1^h)$
2. $m_2^w = \Theta(m_1^w)$
3. $m_2^\phi \models \Theta(m_1^\phi)$

If a method is subsumed by another, then we can safely remove it from the domain description without reducing the number of problems that may be solved using the domain description. Reducing the number of methods in a domain description is desirable because a planner should be more efficient with fewer constructs to consider. However, determining whether or not condition 3 holds requires solving an instance of the Associative-Commutative Unification problem, which has been shown to be NP-complete (Kapur and Narendran, 1986).

4 Theoretical Properties

In this section, we present the formal properties of the HTN-Maker learning algorithm.

4.1 Soundness

The soundness of the HTN-Solver procedure follows directly from the definition of a solution to an HTN planning problem and has been discussed elsewhere (Ghallab et al., 2004). That is, if HTN-Solver produces a plan as a solution for an HTN planning problem, then that plan is indeed a solution to that HTN planning problem. What we attempt to show is that our procedure for learning planning knowledge from traces and annotated tasks is sound, which is to say that plans produced by HTN-Solver using methods learned by HTN-Maker will be solutions to the *classical* planning problems that are the equivalent of the HTN planning problems being solved by HTN-Solver.

Our main soundness result depends on the verification tasks and methods introduced in Sections 3.2 and 3.3. Each learned method has as its last subtask a verification task, which may only be decomposed (into the empty task network) when the postconditions of the associated annotated task hold.

Theorem 1. *Let $\Sigma = (S, A, \gamma)$ be a classical planning domain, E be a finite set of learning examples for that domain, and $\mathcal{T}$ be a finite set of annotated tasks for that domain. Let M be the result of* HTN-Maker$(\Sigma, E, \mathcal{T}, \emptyset)$ *with verification tasks enabled. Let $P = (\Sigma, s_0, g)$ be a classical planning problem and $P^H = (\Sigma^H, s_0', w_0)$ be an equivalent HTN planning problem*

with $\Sigma^H = (S', A', T, M, \gamma')$. *Let* π *be a plan produced by* HTN-SOLVER *as a solution to* P^H.

Then, π *is a solution to* P.

For a proof of this theorem and of helpful lemmas, see Appendix A. For a discussion of soundness without the use of verification tasks and methods, see Appendix C.

4.2 Completeness

We now establish the completeness of the HTN-MAKER algorithm. We say that a set M of HTN methods is *complete* relative to a set of annotated tasks $\mathcal{T}$ for a classical planning domain description Σ if, for any classical planning problem P from the domain described in Σ that is solvable and whose goals have an equivalent annotated task in $\mathcal{T}$, the HTN-equivalent problem P^H is solvable using M.

As before, we present the theorem here and its proof in Appendix A.

Theorem 2. *Let* Σ *be a classical planning domain description and* $\mathcal{T}$ *be a finite set of annotated tasks for the domain.*

Then, there exists a finite set of learning examples E *for that domain such that the set of methods* M *generated by* HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$ *can be used to solve the HTN equivalent to every problem expressible using* Σ *and* $\mathcal{T}$.

We found a specific type of planning problems that can be expressed and solved using the methods learned by HTN-MAKER, which cannot be expressed in classical planning. This type of problems is called *classically-partitionable* and intuitively are problems where specific subproblems must be solved in a certain order (e.g., a problem requiring a vehicle to go from location A to B and then going back to A). For details please see Appendix B.

5 Experimental Evaluation

There are two main questions that we would like to answer regarding HTN-MAKER: how many examples are required for HTN-MAKER to learn a sufficient set of methods for a domain, and how useful are the learned methods for planning? To answer these two questions, we performed two different types of experiments within five planning domains:

- **Rate of convergence.** To measure this, we generated training and testing problems in each domain and solutions to the testing problems. Then we measured the percentage of testing problems that could be solved by an HTN planner using the methods learned from the first training example, the first two training examples, and so forth. If the methods learned from only a few examples are sufficient to solve most of the testing problems, we say that the set of methods rapidly converges to one that is complete.
- **Planning speed.** To measure this, we generated new sets of training problems of varying sizes in each domain and attempted to solve them using the methods learned in the first set of experiments, comparing with several other planners. If the HTN planner using the learned methods is able to solve problems more quickly than classical planners, we say that the methods are of high quality.

Table 1: Configurations of HTN-Maker

Configuration	Subsumption	Generalization
WeakS	Yes	Weak
StrongS	Yes	Strong
WeakNS	No	Weak
StrongNS	No	Strong

The domains used in our experiments are Logistics, Blocks-World, Satellite, Rovers, and Zeno-Travel, each of which was introduced in one of the past international planning competitions. To be useful for our experiments, a domain needed to have a classical representation that still produced interesting problems. For details about the domains, including the annotated tasks used in the experiments, see Appendix D.

5.1 Implementation Details

The nondeterministic choice in Line 15 of Algorithm 3 allows a great deal of flexibility in the algorithm. This choice determines how subtasks are grouped to form methods. Each of the three decomposition trees shown in Figure 11 could be learned by HTN-Maker depending on this choice. For the implementation tested in this evaluation we caused the algorithm to make specific, deliberate choices. Each time the algorithm reaches Line 15, if there exists one or more indexed instances $x \in X'$ of methods learned in previous iterations, then we consider only those as possible subtasks. (That is, the underlying action is not considered as a possible subtask.) When X' contains multiple indexed instances of methods learned in previous iterations, the instance that extends over the largest subplan is selected. (That is, the instance with smallest beginning index x^b.) If there are multiple such indexed method instances, one is chosen arbitrarily and there is no backtracking to consider others. This decision resulted in deep hierarchies (such as the third tree in Figure 11) rather than shallow ones (such as the first tree in Figure 11) and maximized the potential for methods learned from different examples to be used together.

We also required the first subtask of every method to be a primitive action. The reason for this decision is that a reduction directly to a nonprimitive task does not change the state, and thus does not change the set of applicable methods. Thus, poorly designed left-recursive methods could potentially cause an HTN planner to enter a cycle in which it continues the same series of reductions to an infinite depth.

A much earlier version of our implementation instead explored every possible outcome of each nondeterministic choice in the algorithm, learning very many ways to express the same problem solving strategy in different hierarchies. This created a sort of information overload when problem solvers used the learned knowledge and did not capture any useful knowledge that is not also captured with the carefully chosen nondeterministic choices described above.

5.2 Coverage Experiments

We tested the convergence rate of the set of methods learned by HTN-Maker in four different configurations to determine the effectiveness of subsumption checking and the two different models of generalization. These configurations are shown in Table 1.

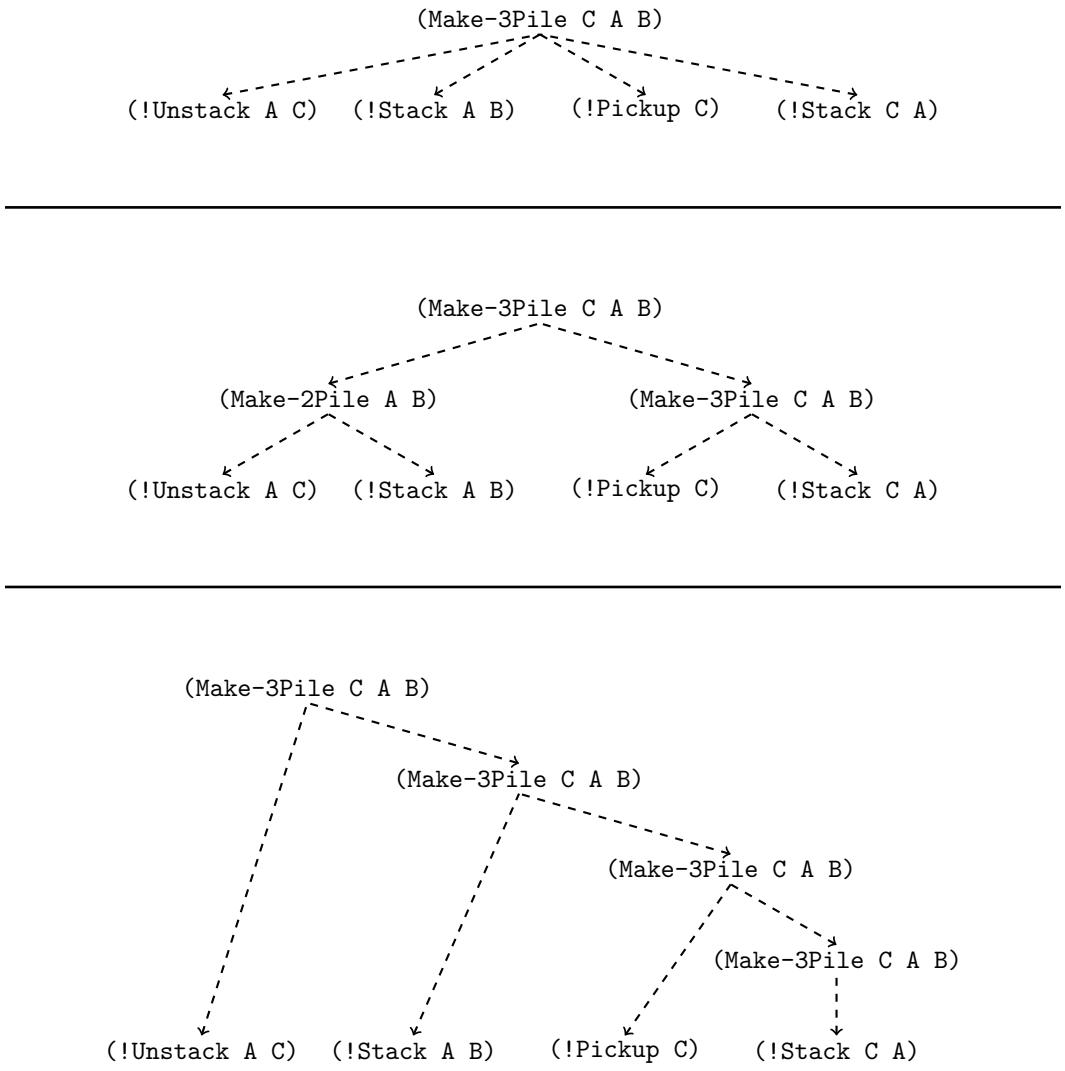


Figure 11: Three possible decomposition trees that could be learned by HTN-Maker.

For each domain we generated 400 random problems of low complexity. For each of five trials, 300 of these problems were randomly selected as a training set and the remaining 100 as a test set. The training and test problems in the Logistics domain had between 1 and 8 packages to be delivered, in the Blocks-World domain between 5 and 10 blocks to reorganize, in the Satellite domain between 1 and 5 images to collect, in the Rovers domain between 3 and 6 waypoints, and in the Zeno-Travel domain between 3 and 8 passengers to transport. Each trial also specified a random ordering over the problems in the training set. Running multiple trials mitigated any bias that might occur from most complex problems being placed in the testing set, or being among the first training examples processed.

We first generated a solution to each of the training problems using the FastForward (Hoffmann and Nebel, 2001) planner. For each training problem, the initial state of the problem and the generated solution formed a learning example. The initial run of HTN-Maker used as input the classical planning domain, a set containing the learning example for the first training problem, the annotated tasks, and an empty set of methods. The second run used as input the classical planning domain, a set containing the learning example for the second training problem, the annotated tasks, and the set of methods produced in the initial run. For each successive training problem we re-ran HTN-Maker, using the set of methods learned from all previous training problems as input. After each run of HTN-Maker, we recorded the number of test problems that could be solved by HTN-Solver (our reimplementation of SHOP) within 30 minutes using the methods learned thusfar. (It is possible but unlikely that some problems could have been solved given more time.)

This experiment was repeated for each of the four configurations of HTN-Maker. The partitioning of problems into training and test sets and the order of the training set for a given trial number are constant between experiments.

The results in the Logistics domain, averaged across the five trials, are shown in Figure 12. Configurations WeakS and WeakNS (those with weak generalization) learn more quickly, but all four configurations solve more than 90% of test problems after learning from 50 training problems. The presence or absence of subsumption has only an extremely small effect.

Figure 13 shows the similar data for the Blocks-World domain. There are no significant differences among the four configurations. As in the Logistics domain, coverage reaches 90% before 50 training problems, but there is a much longer tail.

Figure 14 contains the same information for the Satellite domain. Configuration WeakS, with subsumption and weak generalization, performs relatively poorly in this domain, while the other three learn very quickly. In fact, under configuration WeakS coverage of the domain decreases slightly when it encounters a specific problem, in violation to our Lemma 4. What happens here is that while the new set of methods can still be theoretically used to solve all of the problems that it could solve in the past, our planner is no longer able to do so within a reasonable time limit. Because of the weak generalization and subsumption used in configuration WeakS, HTN-Maker replaces one of the existing methods with a more general version. This generalized method becomes applicable in ways that move toward one solution to the problem, but because the base-case for that solution has not been found, it then moves to a second potential solution where the same conditions are true. Rather than moving on to the solution that can be completed, the planner continues working toward those that cannot. Given enough training examples, even in configuration WeakS we would learn this base case to reach a complete domain.

The Rovers domain (Figure 15) is the most difficult for HTN-Maker to learn. Config-

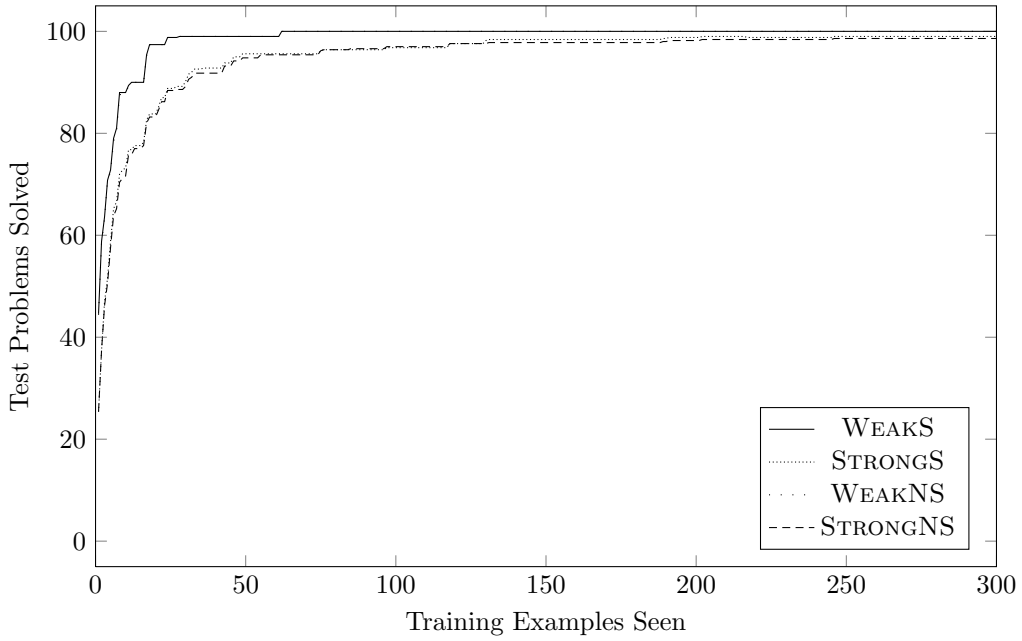


Figure 12: Coverage in Logistics Domain

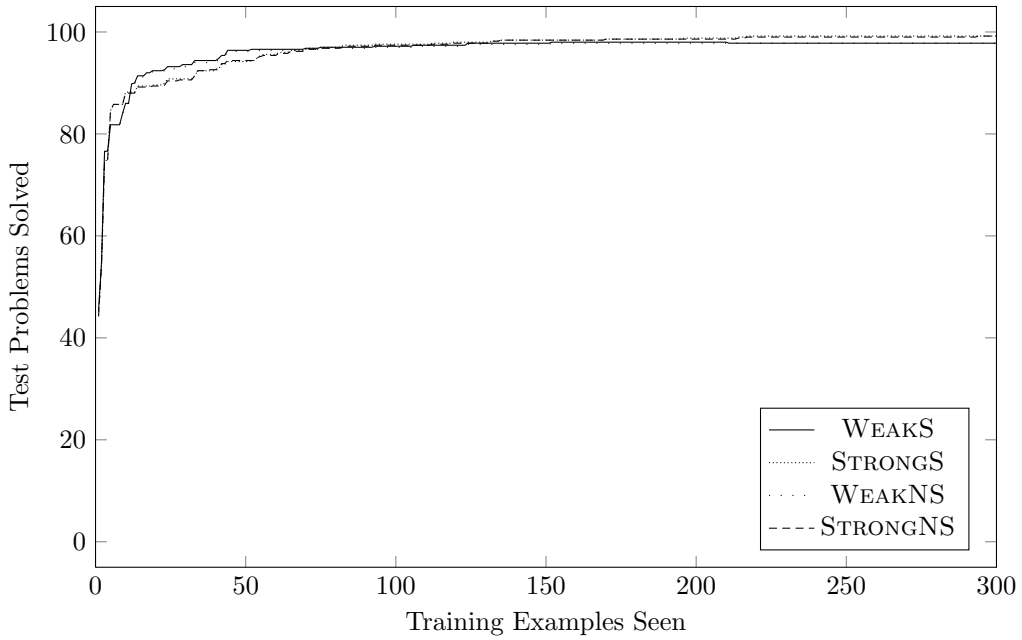


Figure 13: Coverage in Blocks-World Domain

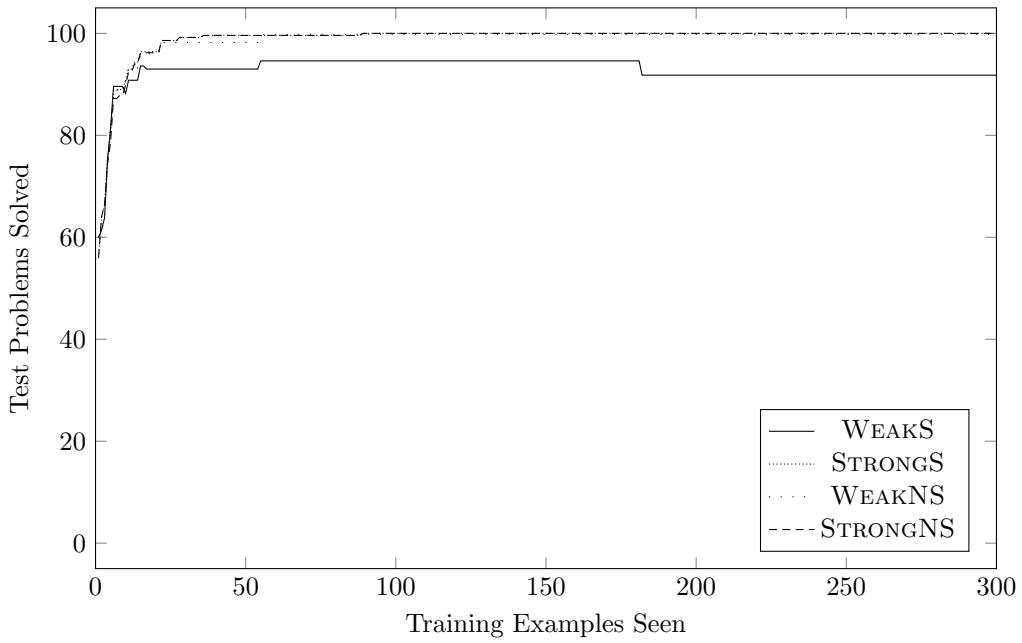


Figure 14: Coverage in Satellite Domain

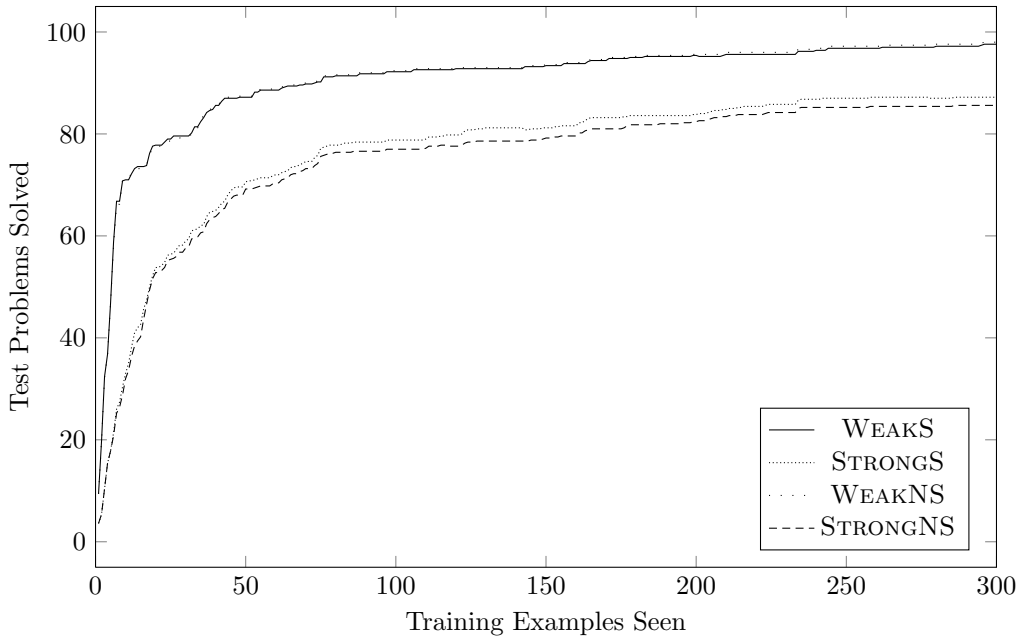


Figure 15: Coverage in Rovers Domain

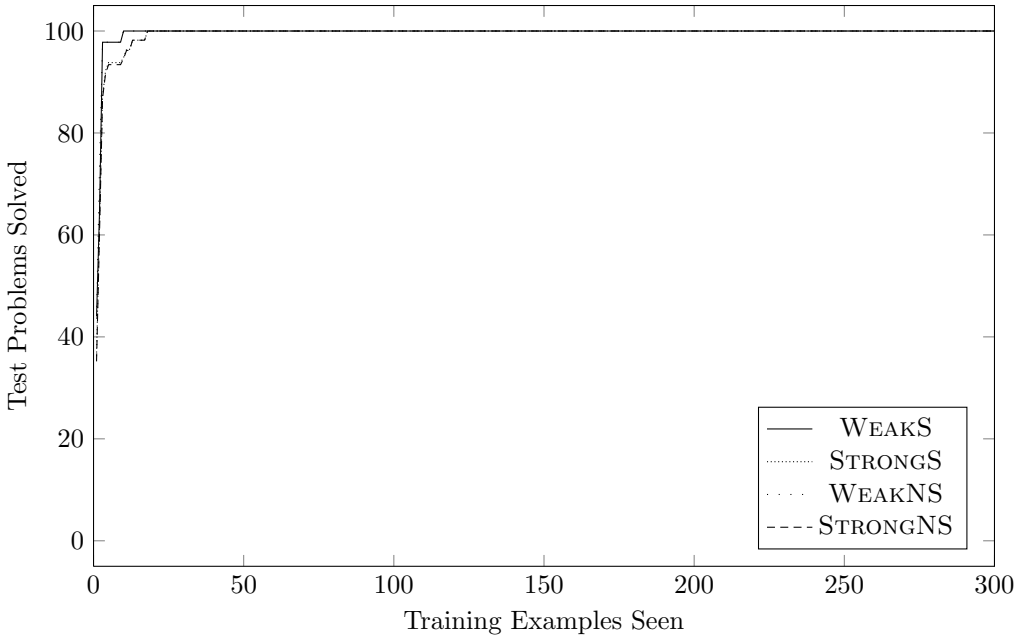


Figure 16: Coverage in Zeno-Travel Domain

urations WeakS and WeakNS still learn reasonably rapidly, passing 80% coverage before 50 training examples, but strong generalization drastically decreases the learning rate. Unlike the other domains in which we tested HTN-Maker, Rovers requires some pathfinding: maneuverability between locations forms a directed graph that is strongly connected but not complete. Thus, a plan to achieve a goal may reference quite a few locations: the one in which the rover is initially located, the one from which a sample needs to be taken, all of those in the path found between these two, the location from which the rover can communicate with the lander, all of those in the path found between the sample location and the communication location, and the location of the lander itself. Requiring that these locations be the same as (or different from) each other only as is strictly necessary to guarantee success in using the methods (weak generalization) produces methods that can be used to solve far more problems than requiring that the relationships among these locations be exactly the same as in the plan from which the methods were learned (strong generalization).

In Zeno-Travel (Figure 16) the system is rapidly able to learn a complete domain regardless of configuration.

We also recorded the total number of methods learned from the 300 training problems, averaged over the five trials. This information is shown in Table 2. As expected, we find that subsumption and weak generalization both decrease the number of methods learned. In general, using subsumption has a more significant impact than the generalization scheme selected.

Finally, we recorded the average time required to learn from a single plan with each of the four configurations of HTN-Maker, which is shown in Table 3. HTN-Maker was much faster in configurations WeakS and WeakNS, in which weak generalization was

Table 2: Number of Methods Learned

Domain	WeakS	StrongS	WeakNS	StrongNS
Logistics	42.2	105.6	230.0	829.4
Blocks-World	146.0	176.7	1000.6	1056.4
Satellite	19.6	24.8	65.8	71.4
Rovers	152.2	419.8	763.6	1957.4
Zeno-Travel	11.0	16.2	4211.6	4371.0

Table 3: Average Number of Seconds to Learn from One Trace

Domain	WeakS	StrongS	WeakNS	StrongNS
Logistics	3.2	13.3	3.3	13.4
Blocks-World	6.8	15.2	6.4	10.9
Satellite	1.66	0.262	0.262	0.225
Rovers	23.7	116.0	21.7	116.0
Zeno-Travel	0.06	0.07	1.61	1.57

used. Surprisingly, although subsumption adds an additional step to the algorithm that is in the worst case NP-complete, it does not significantly affect the average runtime in the Logistics, Blocks-World, or Rovers domains. Our optimized subsumption algorithm can be much faster than its worst-case complexity in the right conditions, and perhaps whatever time was expended searching for methods that subsume one another was offset by having a smaller method file to read and parse for each run.

5.3 Planning Speed Experiments

We also performed a second set of experiments, in which we measured the suitability of the learned methods for quickly solving new problems. In each domain, we generated 20 random classical planning problems of each of several problem sizes, with their HTN equivalents. We then compared the time taken to solve these problems by seven competitors: FastForward (Hoffmann and Nebel, 2001) with the classical version of the domain, SGPlan6 (Hsu and Wah, 2008) with the classical version of the domain, HTN-Solver (the SHOP (Nau et al., 1999) algorithm rewritten and optimized in C++) with a hand-crafted HTN version of the domain, and HTN-Solver with an HTN version of the domain using the methods learned in the first trial (chosen arbitrarily) of the experiments described in Section 5.2 for each configuration of HTN-Maker.

Each planning system was given one hour of CPU time to attempt to solve each problem. Not every competitor was able to solve every problem. In most cases this was because the planner was still working at the end of the time limit. HTN-Solver with the learned methods also failed to solve some problems because its methods did not contain sufficient

FastForward was selected as a "distinguished planner" in the second International Planning Competition in 2000 and remains a common benchmark, while an earlier version of SGPlan6 won the first prize in the satisficing, deterministic planning track of the fifth IPC in 2006.

Table 4: Success rates for each planning domain and planning system

Planner	Logistics	Blocks	Satellite	Rovers	Zeno
FastForward	95.6%	94.0%	51.2%	100.0%	94.8%
SGPlan6	100.0%	99.0%	100.0%	100.0%	100.0%
HTN-Solver (HandHTN)	100.0%	100.0%	100.0%	100.0%	100.0%
HTN-Solver (WeakS)	93.6%	99.0%	100.0%	99.8%	99.2%
HTN-Solver (StrongS)	92.8%	97.0%	98.3%	92.6%	100.0%
HTN-Solver (WeakNS)	89.2%	99.0%	100.0%	99.8%	100.0%
HTN-Solver (StrongNS)	88.1%	94.0%	98.3%	92.4%	100.0%

domain knowledge. The percent of problems solved by each system, for each domain, is shown in Table 4.

Figure 17 shows the average time to solve a problem of each size in the Logistics domain. Note that the vertical axis on this and all other figures in this subsection is logarithmically scaled. The primary result visible in Figure 17 is that FastForward scales very poorly, performing admirably on small problems but taking about 10 times as long as the worst of other competitors at 300 packages. SGPlan6 is better, but still lags behind all configurations of HTN-Solver. Unsurprisingly, HTN-Solver works better with the hand-crafted methods than with those learned by HTN-Maker. Except for two large outliers, HTN-Solver takes about twice as long to solve problems with the learned methods than it does with the hand-crafted methods, regardless of the configuration of HTN-Maker. Among the learned method sets configuration WeakS is best and StrongNS worst, but the differences are slight.

Figure 18 shows the solution times for each configuration in the Blocks-World domain. In this domain FastForward performs even worse, requiring more than 100 times as long to solve large problems as any other competitor. SGPlan6 also scales poorly compared to HTN-Solver, and again HTN-Solver performs better with the hand-crafted methods than the learned ones. This time, there are significant differences between the times for HTN-Solver using methods learned from the different configurations of HTN-Maker. These curves are nearly flat (except for a large outlier in StrongNS), which indicates that at these problem sizes the performance of HTN-Solver is dominated by overhead. It is clear that WeakS has the least overhead and StrongNS the most, as would be expected from the number of methods in Table 2.

Figure 19 shows the data for the Satellite domain. In this domain FastForward performs so poorly that we were not able to collect data on it above a problem size of 150. As in the previous two domains, SGPlan6 performs very well on small problems but appears to scale poorly compared to the HTN approaches. HTN-Solver performs best with the hand-crafted methods, takes about twice as long with the methods learned from configurations WeakS and WeakNS and about twice as much again for configurations StrongS and StrongNS.

Figure 20 shows that FastForward is surprisingly fast in the Rovers domain, beating even the hand-crafted HTN methods. However, HTN-Solver with the handcrafted methods does appear to be scaling better, such that with larger problems it would likely

Later, when presenting data about the average time to solve a problem in each of these systems, we will use only those problems that were solvable by every configuration and show only those difficulty levels where there are at least 10 such problems.

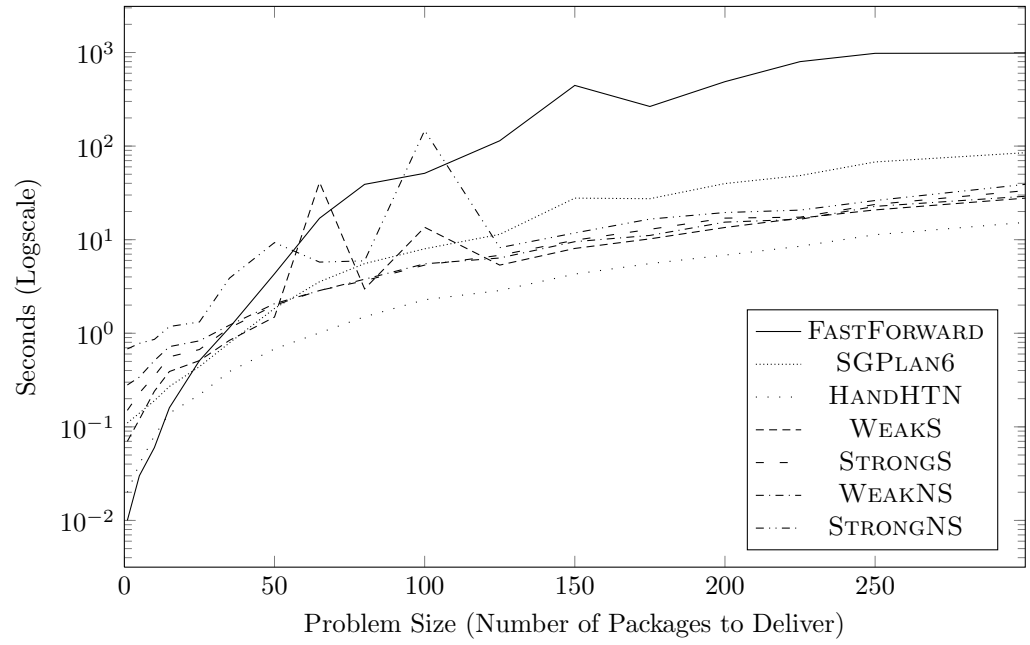


Figure 17: Average problem solving times in Logistics domain

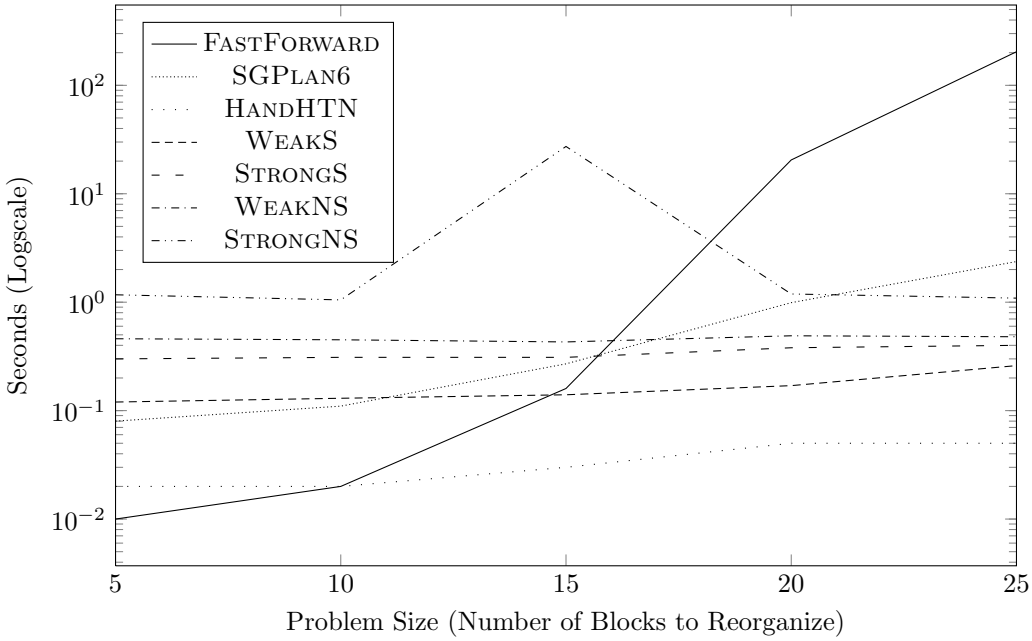


Figure 18: Average problem solving times in Blocks-World domain

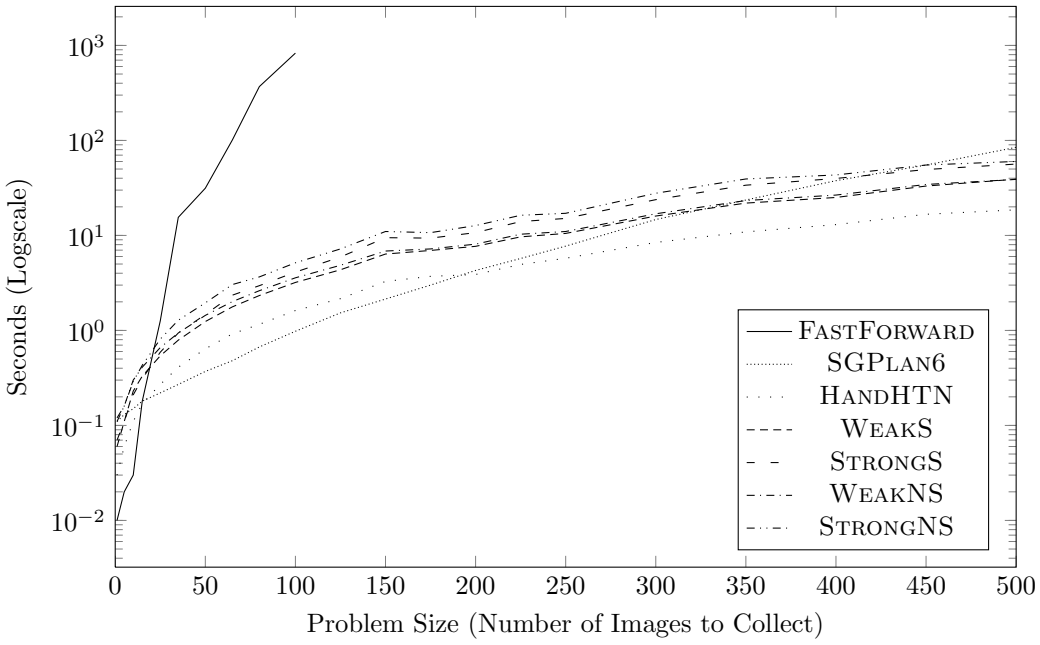


Figure 19: Average problem solving times in Satellite domain

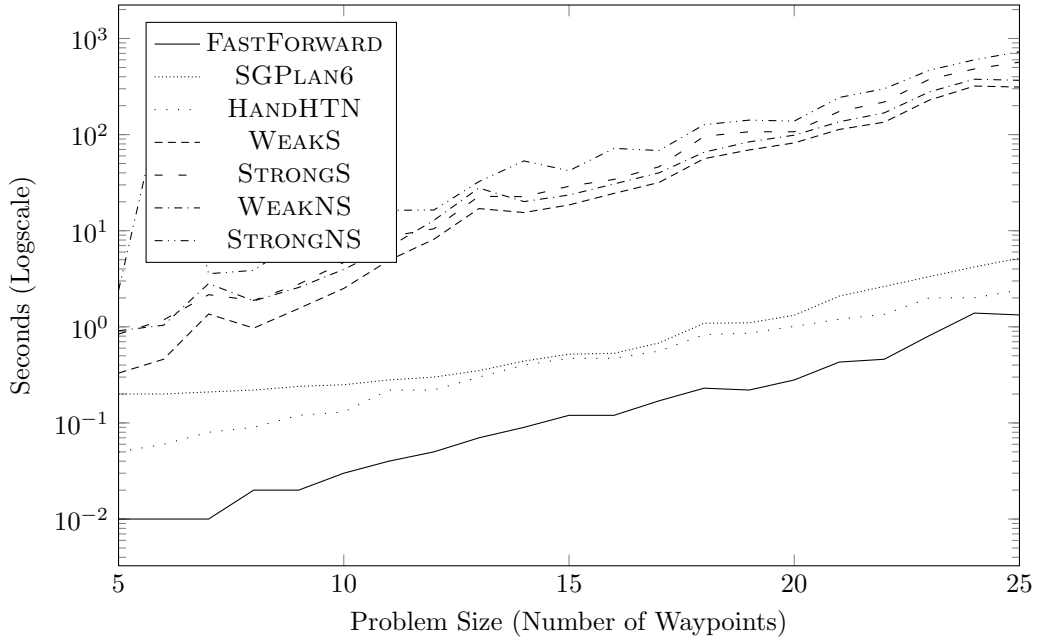


Figure 20: Average problem solving times in Rovers domain

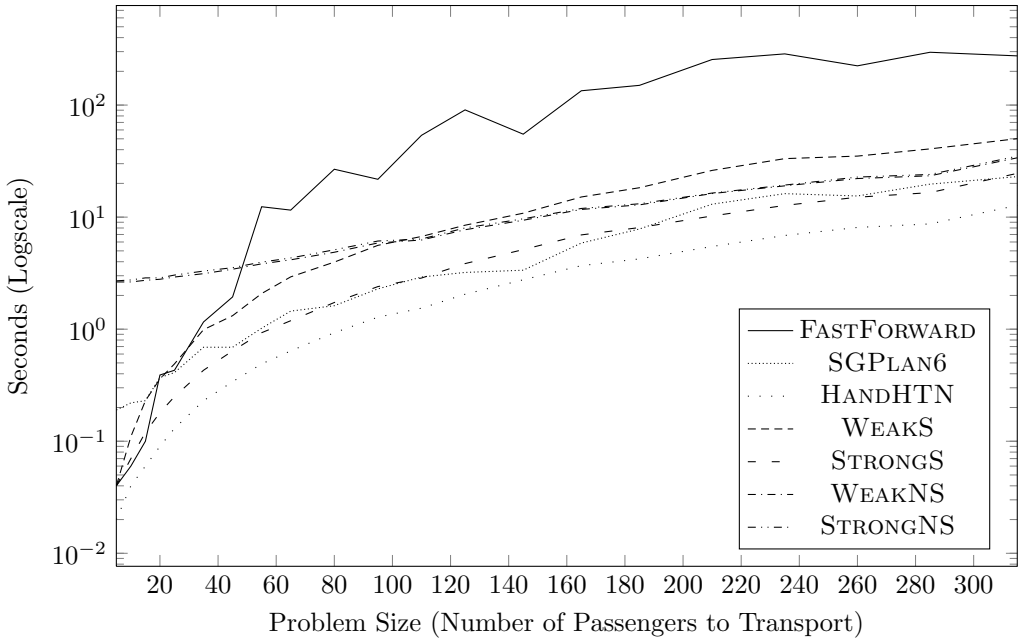


Figure 21: Average problem solving times in Zeno-Travel domain

outperform FastForward as it already outperforms SGPlan6. HTN-Solver with any learned methods performs very poorly. We suspect that this is because the HTN versions of the domain solve a single goal at a time, while the classical planners may consider all of the goals simultaneously and order actions to minimize the necessity of repetition. Among the learned method sets, configuration WeakS is best and StrongNS is worst.

Figure 21 shows the problem-solving times in the Zeno-Travel domain, where FastForward is again quite poor. As before, HTN-Solver with the hand-crafted domain performs best. SGPlan6 and HTN-Solver with the methods learned in configuration StrongS are a bit slower, HTN-Solver with the methods learned in configurations WeakNS and StrongNS are significantly slower, and HTN-Solver with the methods learned in configuration WeakS is slower still.

5.4 Discussion

While it often does not learn a full domain description from the 100 learning examples, HTN-Maker does learn very quickly to solve most problems in a given domain. Solutions for problems in the five domains in which we tested are highly structured, and the HTN-Maker algorithm is able to exploit this structure and generalize from examples to many other problems on which it was not trained. As expected, FastForward is not competitive with more recent planners in four of five domains. The knowledge learned by HTN-Maker can be exploited by HTN-Solver to solve large problems more quickly than even a modern classical planner in three of five domains. The learned HTN methods are not of the same quality as hand-crafted ones, but they are surprisingly close.

The various configurations of HTN-Maker make a significant difference. Subsumption can decrease the number of methods learned by a factor of four or more without requiring substantially more time during the learning process, and usually provides a small but noticeable improvement in problem-solving speed. Using weak generalization makes HTN-Maker run more quickly and generate a more compact set of methods, and requires fewer training examples. Differences in planning times are slight, but also favor weak generalization.

6 Related Work

Structural modeling has led to a number of representations and formalisms, including frames (Minsky, 1975), abstraction techniques (Amarel, 1968), goal graphs (Blum and Furst, 1997), Teleoreactive Logic Programs (Choi and Langley, 2005), and Hierarchical Task Networks (HTNs) (Currie and Tate, 1986; Tate, 1977; Erol et al., 1996; Wilkins, 1988). All of these formalisms have in common the use of certain kinds of constructs (e.g., objects, goals, activities, tasks, and skills) that represent knowledge of varying degrees of complexity and that are connected through structural relations.

6.1 Learning HTNs

Previous work on learning HTNs can be classified according to the following dimensions:

- Task-subtask relations. This dimension refers to whether the learning algorithm assumes that the possible task reductions for each task are given or if they are learned by system.
- Preconditions. This dimension refers to whether the preconditions for each task reduction are given or if they are learned by the system.
- Complete state information. This refers to whether each action in the input traces is annotated with the state that was valid before or after the action. Equivalently, this also includes systems where only the initial state is given but a complete definition of the actions is given, in which case these actions can be used to produce all intermediate states.
- Task expressivity. This indicates whether the tasks must correspond directly to single planning goals or may have richer semantics.
- Incremental. This indicates if the learning system requires a complete set of traces to be able to solve problems or whether it can start solving problems when only a few input traces are given and incrementally solve more problems as more input traces are given. This is a generally desired property as otherwise the system will need to wait until a sufficiently large number of traces is given as input before it can learn a domain to start solving problems.
- Classically-partitionable. Whether or not the learned knowledge can be used to solve classically-partitionable problems, as described in Appendix B. Being able to solve classically-partitionable problems is a desirable property also related to the richness of the expressivity of HTNs.

Table 5: Comparison of Hierarchical Learners

Name	Task-Subtask Rel.	Preconditions	Complete State
HTN-Maker	**Learned**	**Learned**	**Yes**
CaMeL	Given	Learned	Yes
DInCAT	Given	Learned	No
Light	Learned	Learned	Yes
X-Learn	Learned	Learned	Yes
Learn-HTN	Given	Learned	No
L-HTN	Partial	Given	Yes
Name	**Task Expressivity**	**Incremental**	**Classically Partitionable**
HTN-Maker	**Complex**	**Yes**	**Yes**
CaMeL	Complex	No	Yes
DInCAT	Complex	Yes	Yes
Light	Simple	Yes	No
X-Learn	Simple	Yes	No
Learn-HTN	Complex	No	No
L-HTN	Complex	No	Yes

Table 5 compares several HTN learning systems according to these dimensions. The first row is HTN-Maker, which as explained throughout this article learns the task-subtask relations and the preconditions of the methods. HTN-Maker requires that a complete initial state and complete action definitions are included in the input. It can learn general tasks accomplishing single or multiple goals at the same time, and it is incremental as the methods that are learned from the point that it receives the first input trace can be used to solve problems. Finally, the domains it learns can be used by an HTN planner like SHOP to solve classically-partitionable problems.

In the second row is CaMeL (Ilghami et al., 2005). CaMeL assumes that the HTN task reductions are given and that the intermediate states after each action are given in the input trace as well. CaMeL first identifies the literals that change from state to state and propagates these upwards through the given HTN trees. These propagated literals are used as candidates for each task reduction in the HTN. CaMeL then uses the candidate elimination algorithm to learn the best preconditions that cover all literals for each reduction. For this purpose, it also assumes that it receives as input labeled incorrect literals for a reduction, which serve as negative examples. Because it relies on the candidate elimination algorithm it is not incremental. Also, because the task-subtask relationships are provided, it can learn preconditions for methods of any type of tasks, and the methods that it produces can be used by an HTN planner such as SHOP to solve classically-partitionable problems.

The third system is DInCAT (Xu and Munoz-Avila, 2005). Like CaMeL, it assumes that the task-subtask relationships are given. It also assumes that HTN reductions are annotated with applicability conditions, although they do not need to be complete, and that a taxonomy of the types of objects in the domain are given. It then uses inductive generalization techniques to learn the preconditions of methods based on (1) the provided annotations and (2) generalizations of the arguments in these literals based on the taxonomy. It is incremental. Because the HTNs are given, it can learn to solve tasks achieving multiple goals and when given to SHOP its output can solve classically-partitionable problems.

The ICARUS cognitive architecture (Langley and Choi, 2006) uses a variant of HTNs called teleoreactive logic programs. Unlike other systems discussed so far, it does not assume that input traces are given. Instead, it maintains two knowledge bases. The first is called *concepts* and consists of Horn clauses that indicate relationships between goals and subgoals. The second is called *skills* and consists of both constructs similar to actions and constructs similar to methods. When solving a new problem, if ICARUS finds a gap in its library of skills such that it does not know how to proceed from a state it reached to achieve a goal, it uses first-principles planning techniques to fill these gaps. The resulting plan is examined based on the concept and skill hierarchies to explain these gaps and learn new skills that can be used in the future. As a result it is incremental. It requires complete state information. In its current form its tasks are single goals although extensions have been proposed.

The LIGHT system (Nejati et al., 2006; Li et al., 2009) uses similar procedures to learn teleoreactive logic programs from observing an expert. LIGHT formalizes the notion of *goal-indexed HTNs* as a particular special case of HTN formalism that depends on the teleoreactive logic programs. Our indexed method instances, on the other hand, are derived from task-reduction methods in HTN planning.

The skills learned by ICARUS, LIGHT, and other variants cannot be used to solve classically-partitionable planning problems because the skills must achieve a classical goal and need the classical goal statement in their input due to the use of means-ends analysis. One way to enable ICARUS to learn HTN methods for a classically-partitionable planning problem is to re-factor the problem into a series of classical problems and give each problem as input to ICARUS. However, this would require a supervisor system that would do the translation, run ICARUS on the subproblems, and combine the results. HTN-MAKER, on the other hand, can learn from any initial state and sequence of actions, without requiring that they accomplish a particular goal statement.

X-LEARN (Reddy and Tadepalli, 1997) receives planning traces as input and uses inductive generalization to learn *d-rules*, which, similar to the skills of ICARUS, indicate how to reduce a goal into actions and/or other subgoals. X-LEARN has been conceived in the context of bootstrap learning where it assumes that the initial input traces solve simple goals and then more complex traces are given to solve more complex goals. X-LEARN is designed to exploit this by reducing complex goals into subgoals it has already learned to solve. It is incremental and it can solve single goals. It cannot learn how to solve classically-partitionable problems.

Like CAMEL, LEARN-HTN (Zhuo et al., 2009) receives as input the traces and the HTN decompositions used to generate them. But unlike CAMEL it does not assume that the complete intermediate states are given and hence it is designed to learn HTN preconditions when there is limited observability about the state of the world and possibly under noisy conditions. LEARN-HTN is based on the action-model learner ARMS, but generalizes the latter to produce HTN preconditions in addition to action models. We discuss ARMS in the subsequent paragraphs. LEARN-HTN works in three steps: it first builds constraints from given observed decomposition trees to build action models and method preconditions. Second, it then solves these constraints using a weighted MAX-SAT solver. Third, the weighted MAX-SAT solution is then converted to obtain action models and method preconditions. Since the HTNs are given it can learn to solve any tasks and when given to SHOP its output can solve classically-partitionable problems. However, it is not incremental because it usually requires a large number of input traces before a good set of preconditions is learned.

L-HTN (Yang et al., 2007) assumes that a decomposition hierarchy is partially given: it knows the tasks at each level but it does not know how they are reduced into the next level (i.e., which tasks at level n are the parent tasks of which tasks at level $n+1$). It assumes that the tasks at each level are not interleaved (i.e., every nonprimitive task at level n will be reduced into a sequence of contiguous tasks at level $n+1$) and that complete state information is given. It models this learning problem as a Markov Decision Process. Since the HTNs are partially given, it can learn to achieve any task and solve classically-partitionable problems. It is not incremental because it requires enough examples before the MDP converges to a policy.

The work of Biundo and Schattenberg (2001) does not involve learning, but uses structures equivalent to our annotated tasks. In their work, these annotations are used during the planning process so that task-reduction planning can be combined with state-based classical planning to fill in gaps where appropriate methods do not exist.

In our work, we assume that task annotations are provided by a human and that methods should be learned. Marthi et al. (2008), by contrast, assume that methods (called *immediate refinements* in their terminology) are given but that task annotations are not. They demonstrate that several sets of task postconditions may be automatically computed, such as the set of atoms that are guaranteed to become true no matter what refinements are applied or the set of atoms that could possibly become true depending on which refinements are applied. Thus, they provide a way to unify the semantics of SHOP-like systems (determined entirely by the available methods) with those we are using (determined entirely by preconditions and postconditions).

6.2 Other Works on Learning Structural Knowledge

Research on learning HTNs is also related to learning macro-operators, learning action models and learning abstraction. We now discuss these.

Work on learning macro-operators (Mooney, 1988; Botea et al., 2005) is designed to speed up classical planning, as is work on learning search control knowledge (Etzioni, 1993; Minton, 1998; Fern et al., 2004). The aim of search control knowledge algorithms is to learn knowledge constructs that when reused allows the planner to reach its goals more rapidly. For example, macro-operators indicate sequences of two or more actions to be performed when the conditions indicated by the macro-operator are met in the current state. Hence, search control knowledge does not increase the number of problems that theoretically can be solved. However, from a practical stand point, these systems increase the number of problems that can be solved within a reasonable amount of time. Since the learned constructs are part of the classical planning paradigm, they cannot represent classically-partitionable problems.

Fikes et al. (1972) store plans as *triangle tables*, which annotate the reasons that actions were chosen and the relationship between them – essentially the same information computed during goal regression. Triangle tables are then generalized and used to monitor plan execution for correctable failures or surprises. Both entire generalized triangle tables and subtables may be reused as macro-operators in subsequent planning sessions.

Other researchers assumed that hierarchies are given as inputs for learning task models. Garland et al. (2001) use interactive elicitation in which the user provides examples showing how to correctly perform a task and annotates other ways to perform the task in the examples. Tecuci et al. (1999) learn task models where the user interacts with the system providing demonstration and refinement of the models.

Inductive approaches have been proposed for learning action models for classical planning (Martin and Geffner, 2000; McCluskey et al., 2002; Winner and Veloso, 2003). For example, the DISTILL system learns domain-specific planners from an input of plans that have certain annotations (Winner and Veloso, 2003). The input includes the initial state and a complete action model. DISTILL elicits a programming construct for plan generation representing the action model and search control strategies.

Walsh and Littman (2008) present an algorithm to learn operator schemas by observing an agent executing actions, some of which succeed and some of which may fail for unknown reasons, and using a teacher who provides correct plans when the agent makes a mistake. They demonstrate that the number of mistakes made by their algorithm is polynomial in the number of predicates that can be used for preconditions and effects of operators and in the number of actions.

Yang et al. (2007) propose an algorithm called ARMS for learning action models from input plan traces whose intermediate states are partially observable. ARMS uses a series of weighted constraints encoded by the user and extracted from the input traces. For example, one of the constraints says that if a literal occurs in the state before an action but not in the state after the action then it is likely that literal is a negative effect of the action (another plausible explanation is that the literal occurred in the state after the action but was not observed). Traces are parsed and all such constraints are extracted and passed to a weighted MAX-SAT constraint satisfier, which results in truth values for atoms having a high degree of support. These constraints are then used to encode a best-guess model of the preconditions, negative effects, and positive effects of each of the actions.

Another related work is case-based planning (CBP), in which existing plans are stored as cases and then reused for domain or search control knowledge. Systems that use cases to represent domain knowledge, such as the CHEF system (Hammond, 1986) or the BioPlanner (Jin et al., 2009), can be very efficient as the adaptation procedures are specialized for the domain. Their main drawback is that they require the encoding of a new adaptation procedure for every new domain and, hence, the adaptation process lacks clear semantics. Systems that use cases as search control knowledge, such as in the PRODIGY/ANALOGY system (Veloso, 1994) or the ADJ system (Gerevini and Serina, 2010), use a domain-independent adaptation procedure (i.e., the adaptation procedure remains the same regardless of domain), but can be less efficient than domain-specific procedures.

Another related work is abstraction in planning such as the ALPINE (Knoblock, 1993) and the PARIS (Bergmann and Wilke, 1995) systems. These systems take a concrete plan and generalize it. By doing so, it allows the reuse of the generalized plan in different problems by instantiating its conditions. These systems require both an action model and an abstraction model that indicates how to produce abstractions of concrete plans, to be given as input.

DARMOK (Ontañón et al., 2009) learns plan snippets by observing an annotated trace of a human playing of a Real-Time Strategy game. These plan snippets consist of actions to be taken either in parallel or in sequence and possibly subgoals to achieve, and are indexed by the goal that the human indicates they were attempting to achieve with various conditions regarding the state of the world. During planning and execution DARMOK selects an appropriate plan snippet to achieve the current goal and, when it encounters a subgoal, selects another plan snippet to achieve it. In this way, plan snippets can be used to represent hierarchical structure similarly to HTNs. Unlike HTN-MAKER, DARMOK depends on the expert annotations to determine the purpose of each action and does not compute preconditions for snippets.

7 Conclusions

HTN planning is an effective problem-solving paradigm, but the high knowledge engineering cost of developing an HTN domain description is a significant impediment to the wider adoption of HTN planning technology. We have described a new algorithm, HTN-Maker, for incrementally learning HTN domain knowledge in the form of task-reduction methods from planning states and plans applicable to those states. HTN-Maker produces a set of HTN methods by learning the decomposition structure of tasks from annotated tasks and plans. The learner constructs a hierarchy in a bottom-up manner by analyzing the sequences of actions in a plan trace and determines the preconditions of methods by regressing goals through the subtasks of those methods.

We have presented theoretical results showing that the methods learned by HTN-Maker are sound and complete relative to the set of goals for which annotated tasks are provided. Our experiments in five well-known planning domains demonstrated that HTN-Maker converged toward a set of HTN methods that solve all problems in the domain as more problems are presented. In three of the five domains, an HTN planner using the learned methods could solve large problems much more quickly than a modern classical planner.

We intend to expand this work in several future directions. First, our experiments currently use very simple annotated tasks, but our formalism and algorithms support more complex ones. We believe that exerting more knowledge engineering effort to carefully design annotated tasks may allow HTN-Maker to learn a complete set of methods from fewer problems, or to learn sets of methods that enable quicker problem solving, and would like to explore this empirically.

Second, we are currently developing techniques for using reinforcement learning mechanisms on top of HTN-Maker in order to learn the expected values of the HTN methods produced by the algorithm. This will enable us to study optimality and usefulness properties of the learned HTNs.

Third, the heuristics described in Section 5.1 were developed through intuition and experimentation, but there may be much better ways to pick and choose among the types of methods that HTN-Maker is capable of generating. We would like to qualitatively compare our current results against other possible implementations, with an aim toward developing a formal notion of method quality independent of the numerical approach suggested above.

Fourth, we would like to expand the representation of preconditions in our annotated tasks and methods. Currently, preconditions are sets of atoms from the domain. This means that, in the Blocks-World domain for example, a method can have as preconditions one block be directly on top of another, or that there be one block between them, or two, or three, and so forth. But it is not currently possible for a method to have as its preconditions that the first block be somewhere above the second without specifying exactly how many blocks are between them. Other planners, including SHOP, allow method preconditions to include derived predicates that could represent something like "is somewhere above". We would like to experiment with modifying HTN-Maker to replace atoms in the preconditions of methods with predicates that can be derived from them. This would make the methods learned much more general, and likely mean that fewer learning examples would be needed.

Fifth, it would be interesting to use HTN-Maker in an active learning environment similar to how Icarus (Langley and Choi, 2006) works. This would mean attempting to solve problems with an HTN planner using an existing knowledge base and if this fails falling back on a classical planner, then using the result of that classical planner as a new learning instance for HTN-Maker to improve the HTN planner's knowledge base before attempting

future problems. Furthermore, having pre- and postconditions on tasks would allow us to build a planner that interleaves hierarchical and classical planning within the processing of a single problem. Such a planner would use HTN planning whenever possible, but when it finds no applicable methods to reduce a task t from a state s it could use any classical planning technique to find a plan from that state s that accomplishes the postconditions associated with t and then continue reducing later tasks for which it does have sufficient knowledge, after learning from the example provided by the classical planning component.

Acknowledgments

This research was in part supported by the National Science Foundation (NSF 0642882). The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

Amarel, Saul. 1968. On representations of problems of reasoning about actions. *In* Machine Intelligence 3.

Bergmann, R., and W. Wilke. 1995. Building and refining abstract planning cases by change of representation language. Journal of Artificial Intelligence Research, **3**:53–118.

Biundo, Susanne, and Bernd Schattenberg. 2001. From abstract crisis to concrete relief: A preliminary report on combining state abstraction and htn planning. *In* Proceedings of the Sixth European Conference on Planning (ECP-01), pp. 157–168.

Blum, A., and M. Furst. 1997. Fast planning through planning graph analysis. Artificial Intelligence, **90**:281–300.

Botea, A., M. Enzenberger, M. Müller, and J. Schaeffer. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. Journal of Artificial Intelligence Research, **24**:581–621.

Cavazza, M., and F. Charles. 2005. Dialogue generation in character-based interactive storytelling. *In* Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05).

Choi, D., and P. Langley. 2005. Learning teleoreactive logic programs from problem solving. *In* Proceedings of the Fifteenth International Conference on Inductive Logic Programming (ILP-05).

Currie, Ken, and Austin Tate. 1986. O-Plan: Control in the open planning architecture. *In* Proceedings of the Fifth Technical Conference of the British Computer Society Specialist Group on Expert Systems.

Erol, Kutluhan, James Hendler, and Dana S. Nau. 1996. Complexity results for hierarchical task-network planning. Annals of Mathematics and Artificial Intelligence, **18**(1):69–93.

Etzioni, Oren. 1993. Acquiring search-control knowledge via static analysis. Artificial Intelligence, **62**(2):255–301.

Fdez-Olivares, Juan, Luis Castillo, Oscar Garcia-Perez, and Francisco Palao. 2006. Bringing users and planning technology together, experiences in SIADEX. *In* Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS-06).

Fern, A., S. W. Yoon, and R. Givan. 2004. Learning domain-specific control knowledge from random walks. *In* Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04).

Fikes, Richard E., Peter E. Hart, and Nils J. Nilsson. 1972. Learning and executing generalized robot plans. Artificial Intelligence, **3**(1-3):251–288.

Garland, A., K. Ryall, and C. Rich. 2001. Learning hierarchical task models by defining and refining examples. *In* Proceedings of the First International Conference on Knowledge Capture (K-CAP-01).

Gerevini, A., and I. Serina. 2010. Efficient plan adaptation through replanning windows and heuristic goals. Fundamenta Informaticae, **102**(3-4):287–323.

Ghallab, Malik, Dana Nau, and Paolo Traverso. 2004. Automated Planning: Theory and Practice. Morgan Kaufmann.

Hammond, Kristian J. 1986. CHEF: A model of case-based planning. *In* Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86).

Hoang, H., S. Lee-Urban, and H. Muñoz-Avila. 2005. Hierarchical plan representations for encoding strategic game AI. *In* Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05).

Hoffmann, Jörg, and Bernhard Nebel. 2001. The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research, **14**:253–302.

Hsu, C. W., and B. W. Wah. 2008. The SGPlan planning system in IPC-6. *In* Proceedings of the Sixth International Planning Competition at ICAPS-08.

Ilghami, O., H. Munoz-Avila, D. Nau, and D. W. Aha. 2005. Learning approximate preconditions for methods in hierarchical plans. *In* Proceedings of the Twenty-Second International Conference on Machine Learning (ICML-05).

Jin, L., K. Decker, and C. Schmidt. 2009. BioPlanner: A plan adaptation approach for the discovery of biological pathways across species. *In* Proceedings of the Twenty-First Conference on Innovative Applications of AI (IAAI-09).

Kambhampati, S., and J. A. Hendler. 1992. A validation-structure-based theory of plan modification and reuse. Artificial Intelligence, **55**:193–258.

Kapur, Deepak, and Paliath Narendran. 1986. NP-completeness of the set unification and matching problems. *In* Proceedings of the Eighth International Conference on Automated Deduction (CADE-86).

Khardon, R. 1999. Learning action strategies for planning domains. Artificial Intelligence, **113**(1-2):125–148.

Knoblock, C. 1993. Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning. Kluwer Academic Publishers.

Könik, T., and J. E. Laird. 2006. Learning goal hierarchies from structured observations and expert annotations. Machine Learning, **64**(1-3):263–287.

Könik, T., N. Nejati, and U. Kuter. 2009. Inductive generalization of analytically learned goal hierarchies. *In* Proceedings of the Nineteenth International Conference on Inductive Logic Programming (ILP-09).

Langley, Pat, and Dongkyu Choi. 2006. Learning recursive control programs from problem solving. Journal of Machine Learning Research, **7**:493–518.

Li, Nan, David J. Stracuzzi, Gary Cleveland, and Pat Langley. 2009. Learning hierarchical skills for game agents from video of human behavior. *In* Proceedings of the IJCAI Workshop on Learning Structural Knowledge from Observations (StrucK-09).

Marthi, Bhaskara, Stuart J. Russell, and Jason Wolfe. 2008. Angelic hierarchical planning: Optimal and online algorithms. *In* Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08), pp. 222–231.

Martin, M., and H. Geffner. 2000. Learning generalized policies in planning using concept languages. *In* Proceedings of the Seventh International Conference on Knowledge Representation and Reasoning (KR-00).

McCluskey, T. L., N. Elisabeth Richardson, and Ron M. Simpson. 2002. An interactive method for inducing operator descriptions. *In* Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS-02).

McDermott, Drew. 1998. PDDL, the planning domain definition language. Technical report, Yale Center for Computational Vision and Control.

Minsky, Marvin. 1975. A framework for representing knowledge. *In* The Psychology of Computer Vision.

Minton, S. 1998. Learning Effective Search Control Knowledge: an Explanation-Based Approach. Ph. D. thesis, Carnegie Mellon University.

Minton, Steven, Jaime G. Carbonell, Craig A. Knoblock, Daniel Kuokka, Oren Etzioni, and Yolanda Gil. 1989. Explanation-based learning: A problem solving perspective. Artificial Intelligence, **40**(1-3):63–118.

Mitchell, S. W. 1997. A hybrid architecture for real-time mixed initiative planning and control. *In* Proceedings of the Ninth Conference on Innovative Applications of AI (IAAI-97).

Mitchell, T., R. Keller, and S. Kedar-Cabelli. 1986. Explanation-based generalization: A unifying view. Machine Learning, **1**(1):47–80.

Mooney, R. J. 1988. Generalizing the order of operators in macro-operators. *In* Proceedings of the Fifth International Conference on Machine Learning (ICML-88).

Muñoz-Avila, H., K. Gupta, D. W. Aha, and D. S. Nau. 2002. Knowledge Management and Organizational Memories, Chapter Knowledge Based Project Planning. Kluwer.

Muñoz-Avila, H., D. McFarlane, D. W. Aha, J. Ballas, L. A. Breslow, and D. S. Nau. 1999. Using guidelines to constrain interactive case-based HTN planning. *In* Proceedings of the Third International Conference on Case-Based Reasoning (ICCB-99).

Murdock, J. W. 2001. Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation. Ph. D. thesis, Georgia Institute Of Technology.

Nau, D., T.-C Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. 2003. SHOP2: An HTN planning system. Journal of Artificial Intelligence Research, **20**:379–404.

Nau, D., Y. Cao, A. Lotem, and H. Munoz-Avila. 1999. SHOP: Simple hierarchical ordered planner. *In* Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99).

Nejati, Negin, Tolga Könik, and Ugur Kuter. 2009. A goal- and dependency-driven algorithm for learning hierarchical task networks. *In* Proceedings of the Fifth International Conference on Knowledge Capture (KCAP-09).

Nejati, N., P. Langley, and T. Könik. 2006. Learning hierarchical task networks by observation. *In* Proceedings of the Twenty-Third International Conference on Machine Learning (ICML-06).

Ontañón, Santiago, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. 2009. Online case-based planning. Computational Intelligence, **26**(1):84–119.

Project Management Institute. 2013. A Guide to the Project Management Body of Knowledge (Fifth ed.). Project Management Institute.

Reddy, C., and P. Tadepalli. 1997. Learning goal-decomposition rules using exercises. *In* Proceedings of the Fourteenth International Conference on Machine Learning (ICML-97).

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. *In* Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. *Edited by* V. Lifschitz. Academic Press.

Smith, Stephen J. J., Dana S. Nau, and Kutluhan Erol. 1998. Control strategies in HTN planning: theory versus practice. *In* Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence (IAAI-98).

Sutton, Jr., Stanley M., Dennis Heimbigner, and Leon J. Osterweil. 1995. APPL/A: a language for software process programming. ACM Transactions on Software Engineering and Methodology, **4**(3):221–286.

Tate, A. 1977. Generating project networks. *In* Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77).

Tecuci, Gheorghe, Mihai Boicu, Kathryn Wright, Seok Won Lee, Dorin Marcu, and Michael Bowman. 1999. An integrated shell and methodology for rapid development of knowledge-based agents. *In* Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99).

Ulam, Patrick, Ashok Goel, Joshua Jones, and William Murdock. 2005. Using model-based reflection to guide reinforcement learning. *In* IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games.

Veloso, Manuela. 1994. Planning and Learning by Analogical Reasoning. Springer-Verlag.

Waldinger, Richard. 1977. Achieving several goals simultaneously. *In* Machine Intelligence 8. *Edited by* E. W. Elcock and D. Mitchie.

Walsh, Thomas J., and Michael L. Littman. 2008. Efficient learning of action schemas and web-service descriptions. *In* Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08).

Wilkins, David E. 1988. Practical Planning: Extending the Classical AI Planning Paradigm. Morgan Kaufmann.

Winner, E., and M. M. Veloso. 2003. DISTILL: Learning domain-specific planners by example. *In* Proceedings of the Twentieth International Conference on Machine Learning (ICML-03).

Xu, K., and H. Muñoz-Avila. 2004. CaBMA: Case-based project management assistant. *In* Proceedings of the Sixteenth Conference on Innovative Applications of AI (IAAI-04), pp. 931–936.

Xu, K., and H. Munoz-Avila. 2005. A domain-independent system for case-based task decomposition without domain theories. *In* Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05).

Yang, Qiang, Rong Pan, and Sinno Jialin Pan. 2007. Learning recursive HTN-method structures for planning. *In* Proceedings of the ICAPS Workshop on Artificial Intelligence Planning and Learning (AIPL-07).

Yang, Q., K. Wu, and Y. Jiang. 2007. Learning action models from plan examples using weighted MAX-SAT. Artificial Intelligence, **171**:107–143.

Zhuo, Hankui, Qiang Yang, Derek Hao Hu, Chad Hogg, and Héctor Muñoz-Avila. 2009. Learning htn method preconditions and action models from partial observations. *In* Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09).

List of Figures

A Lemmas And Proofs

Theorem 1 from Section 4.1 supports the soundness of our learning procedure. Here we present two lemmas that will assist in proving that theorem, then restate the theorem and provide a proof of it.

Lemma 1. *Let Σ be a classical planning domain, E be a finite set of learning examples for that domain, and $\mathcal{T}$ be a finite set of annotated tasks for that domain. Let M be the result of* HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$.

Then, for each annotated task $\tau = (\tau^h, \tau^\phi, \tau^+) \in \mathcal{T}$ there exists a method $m = (\tau^h, \tau^\phi \cup \tau^+, \langle\rangle) \in M$.

Proof. The MAKE-TRIVIAL-METHODS subroutine of HTN-MAKER generates such a method for every annotated task. □

Lemma 1 simply states that there is a base-case method for each annotated task that allows that task to be reduced to the empty task network when in a state where both the preconditions and postconditions of the annotated task hold.

Lemma 2. *Let $P = (\Sigma, s_0, g)$ be a classical planning problem with $\Sigma = (S, A, \gamma)$, and $P^H = (\Sigma^H, s_0', w_0)$ be an equivalent HTN planning problem with $\Sigma^H = (S', A', T, M, \gamma')$, where M is the result of* HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$. *Let $\pi = \langle a_0, a_1, \ldots, a_n\rangle$ be a plan produced by* HTN-SOLVER *(Algorithm 1) as a solution to P^H.*

Then, π is applicable to the initial state of P, s_0.

Proof. Because P^H is an HTN planning problem equivalent to P, we know that $s_0 = s'_0$, $S = S'$, $A = A'$, and $\gamma = \gamma'$. The correctness of HTN-SOLVER (which has previously been proven in Ghallab et al. (2004)) means that π must be applicable to s'_0, which is s_0. □

Lemma 2 does not depend on any property of the methods in the input, only on the fact that HTN-SOLVER enforces the preconditions of the actions in the plans that it produces.

Theorem 1. *Let $\Sigma = (S, A, \gamma)$ be a classical planning domain, E be a finite set of learning examples for that domain, and $\mathcal{T}$ be a finite set of annotated tasks for that domain. Let M be the result of* HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$ *with verification tasks enabled. Let $P = (\Sigma, s_0, g)$ be a classical planning problem and $P^H = (\Sigma^H, s'_0, w_0)$ be an equivalent HTN planning problem with $\Sigma^H = (S', A', T, M, \gamma')$. Let π be a plan produced by* HTN-SOLVER *as a solution to P^H.*

Then, π is a solution to P.

Proof. In order to prove that π is a solution to P we must show both that it is applicable to s_0 and that it produces a state in which g holds. Lemma 2 guarantees the first part. The remainder of this proof demonstrates the second part.

Because P^H is an equivalent HTN planning problem to P, we know that $s_0 = s'_0$, $S = S'$, $A = A'$, and $\gamma = \gamma'$. Furthermore, we know that there exists an annotated task $\tau = (\tau^h, \emptyset, g) \in \mathcal{T}$ such that $w_0 = \langle \tau^h \rangle$.

If π is the empty plan, then the task τ^h must have been reduced using the method described in Lemma 1. Thus, $s'_0 \models g$. Since $s'_0 = s_0$ and there are no actions in the plan, s'_0 is the result of $\gamma(s_0, \pi)$, and we have shown that it satisfies the goals.

If π is not the empty plan, then some reductions were performed using learned methods to produce it. Consider the method used for the very first reduction. Because it was generated by HTN-MAKER with verification tasks enabled, its final subtask will be a verification task t'. There exists one and only one method, $m = (t', g, \langle\rangle)$ for this verification task. The very last step taken by HTN-SOLVER to produce π will have been a reduction of this verification task using that only method. Because this method was applicable, the planner's current state was one in which the goals were satisfied. Because this is the last step, that current state is also the final state $\gamma(s_0, \pi)$. □

Theorem 2 from Section 4.2 supports the completeness of our learning procedure. Here we present two lemmas and a proof of that theorem.

Lemma 3. *Let Σ be a classical planning domain description, $\mathcal{T}$ be a set of annotated tasks for the domain, $P = (\Sigma, s_0, g)$ be a classical planning problem from the domain, $\tau = (\tau^h, \emptyset, g) \in \mathcal{T}$ be the equivalent annotated task to g, and π be a solution to P.*

Then, the set of methods M learned by HTN-MAKER *from a single learning example $e = (s_0, \pi)$ can be used to solve the HTN-equivalent problem $P^H = (\Sigma^H, s_0, \langle \tau \rangle)$.*

Proof. If π is empty or g is satisfied in s_0, then the trivial method for τ is sufficient to solve the problem. Otherwise, HTN-MAKER will have learned at least one method for accomplishing τ from π. This method must be applicable to s_0 because its preconditions were computed by regressing g through the actions of π, which is applicable to s_0. Furthermore, our goal regression procedure guarantees that whenever the preconditions of a method are satisfied there must be some way to reduce that method's subtasks using other methods from whose indexed instances those subtasks were chosen. □

Lemma 4. *Let Σ be a classical planning domain description, $\mathcal{T}$ be a set of annotated tasks for the domain, and M be a set of methods learned by* HTN-Maker *from any finite set of learning examples E in the domain. Let $e = (s_0, \pi)$ be any learning example from the domain, which may or may not be a member of E. Let M' be the set of methods that* HTN-Maker *learns from e when starting with M.*

Then, if M can be used to solve a problem P^H then M' can be used to solve P^H as well.

Proof. If subsumption checking (see Section 3.4.3) is not enabled, then HTN-Maker never erases a method and hence $M \subseteq M'$. When subsumption checking is enabled, a method m is never removed from the set of methods unless a method m' is being added that is applicable whenever m is applicable and that encodes the same problem-solving strategy. Neither adding an additional method nor replacing a method with a more general version can reduce the set of solvable problems. □

Theorem 2. *Let Σ be a classical planning domain description and $\mathcal{T}$ be a finite set of annotated tasks for the domain.*

Then, there exists a finite set of learning examples E for that domain such that the set of methods M generated by HTN-Maker$(\Sigma, E, \mathcal{T}, \emptyset)$ *can be used to solve the HTN equivalent to every problem expressible using Σ and $\mathcal{T}$.*

Proof. Consider the set S of states in Σ and the set of goal statements G that have an equivalent annotated task in $\mathcal{T}$. Every solvable problem in Σ with an equivalent HTN problem has the form $P = (\Sigma, s_0, g)$ where $s \in S$ and $g \in G$. Because the sets S and G are finite, there is a finite number of such problems. Let the set of learning examples E consist of the initial state of each such problem paired with any solution to that problem. Lemmas 3 and 4 state that the methods that HTN-Maker would learn from the set of learning examples E can be used to solve the HTN equivalents of each of the problems that form a learning example in E. We have previously shown that this includes every solvable problem in the domain that has an HTN equivalent problem using tasks from $\mathcal{T}$. □

Intuitively, this means that HTN-Maker is able to learn a complete HTN description of any classical planning domain. Although our theoretical result shows only that the worst case requires learning from every problem in the domain, our experience indicates that far fewer problems are needed in practice. In one experiment, we were able to solve all solvable Logistics domain problems that required delivering a single package to a location after learning from six carefully chosen learning examples.

B Classically-Partitionable Planning

Simple Task Network planning, using methods with totally ordered subtasks as discussed in this paper, is strictly more expressive than classical planning, and general HTN planning is even more expressive than STN planning (Erol et al., 1996). In general, HTN planning is undecidable, STN planning is EXPSPACE-hard, and classical planning, even with macro-operators, is NP-complete. In the rest of this section we formalize a class of planning problems that are more expressive than classical planning but less expressive than general STN planning, and show that the methods learned by HTN-Maker can be used to solve planning problems in this class, even though they were learned from solutions to classical planning problems.

Consider again the BLOCKS-WORLD planning domain, in which a robotic arm moves blocks around on a table. Suppose we have a planning problem in which block `A` is initially on the table and needs to be moved first to on top of block `B`, then to on top of block `C`, then back to the table. This cannot be represented as a classical planning problem without introducing function symbols into the representation language or otherwise modifying the domain to directly encode information about what facts were true in previous states. On the other hand, it could easily be represented as an HTN planning problem, where the initial task network is `<(Put-On-Block A B), (Put-On-Block A C), (Put-On-Table A)>`. The methods learned by HTN-MAKER on learning examples from the classical BLOCKS-WORLD domain can be used by an HTN planner to solve problems of this form.

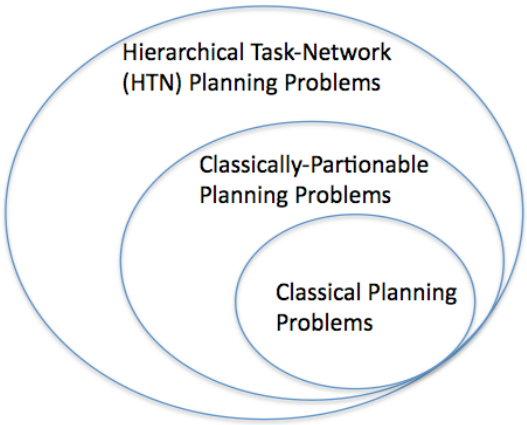


Figure 22: Expressivity of classically-partitionable planning problems.

We formalize the notion of the class of planning problems described above as follows. Let $\vec{G} = \langle g_0, g_1, \ldots g_n \rangle$ be a sequence of goal statements from a classical planning domain. Then $P^P = (\Sigma, s_0, \vec{G})$ is a *classically-partitionable planning problem.* A plan π is a solution to P^P if and only if π may be partitioned into a sequence of subplans $\langle \pi_0, \pi_1, \ldots, \pi_n \rangle$ such that each π_i is applicable in s_i and produces state s_{i+1} such that $s_{i+1} \models g_i$.

Note that classically-partitionable planning problems appear in many planning domains, including LOGISTICS, BLOCKS-WORLD, ROVERS, and others that were used as benchmarks in the past International Planning Competitions.

Given a classically-partitionable planning problem $P^P = (\Sigma, s_0, \langle g_0, g_1, \ldots, g_n \rangle)$, and a finite set of annotated tasks $\mathcal{T}$, there is an *equivalent HTN planning problem* $P^H = (\Sigma^H, s_0, \langle t_0, t_1, \ldots, t_n \rangle)$, where $\Sigma = (S, A, \gamma)$, $\Sigma^H = (S, A, T, M, \gamma)$, each task in T has an annotated version in $\mathcal{T}$, and for each $0 < i \leq n$ there exists an annotated task $\tau = (t_i, \emptyset, g_i) \in \mathcal{T}$. Informally, the initial task network of any equivalent HTN planning problem contains the equivalent annotated task to each goal set in the classically-partitionable planning problem, in the same order. For any classically-partitionable planning problem $P^P = (\Sigma, s_0, \vec{G})$, we can construct an equivalent HTN planning problem $P^H = (\Sigma^H, s_0, w_0)$ by making an equivalent annotated task for each goal set in $\vec{G}$.

Theorem 3. *Let $P^P = (\Sigma, s_0, \vec{G})$ be a solvable classically-partitionable planning problem*

and $\mathcal{T}$ be a finite set of annotated tasks such that $P^H = (\Sigma^H, s_0, w_0)$ is an equivalent HTN planning problem to P^P.

Then, there exists a finite set of learning examples E such that the set of methods M learned by HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$ *will allow an HTN planner to solve P^H.*

Proof. Theorem 2 guarantees that there exists a finite set of learning examples E such that a set of methods M generated by HTN-MAKER$(\Sigma, E, \mathcal{T}, \emptyset)$ can be used to solve the HTN equivalent to every solvable classical planning problem expressible with Σ and $\mathcal{T}$. We will show that any set of methods that can solve the HTN equivalent to every classical planning problem expressible with Σ and $\mathcal{T}$ can also solve the HTN equivalent to every classically-partitionable planning problem expressible with Σ and $\mathcal{T}$.

If P^P has a single goal set, then the HTN equivalent to it is the same as the HTN equivalent to the classical planning problem that uses that goal set. Thus, it can be solved using M.

Suppose that P^P has $n > 1$ goal sets and that this theorem has been proven for all classically-partitionable planning problems with $n - 1$ goal sets. Then an HTN planner using M can solve the HTN equivalent to the classically-partitionable planning problem $(\Sigma, s_0, \langle g_0, g_1, \ldots g_{n-1} \rangle)$. Solving this produces a state s and an empty task network. If the classically-partitionable planning problem had instead included goal set g_n at the end of $\vec{G}$, then the planner would reach a point at which the current state is s and the current task network is t_n by following the same sequence of steps. This is itself the HTN equivalent to a classical planning problem in the domain, and thus M can be used to solve it. □

C Necessity Of Verification Tasks

In Section 4.1 we showed that any solution generated by a sound HTN planner on an HTN planning problem using methods learned by HTN-MAKER is guaranteed to be a solution to the equivalent classical planning problem, but only because of the use of verification tasks and verification methods. Without verification tasks, there are specific cases in which the methods learned by HTN-MAKER might be used to find a solution to an HTN planning problem that is not a solution to the equivalent classical planning problem. This possibility exists because in a sequence of nonprimitive subtasks, a particular valid reduction of a later nonprimitive subtask may negate an effect of an earlier nonprimitive subtask that was needed to accomplish the highest-level task. We now show a concrete example illustrating this situation.

```
( :task Make-2Pile
  :parameters (?x ?y)
  :precondition ()
  :postcondition (on ?x ?y) )
```

```
( :task Make-3Pile
  :parameters (?x ?y ?z)
  :precondition ()
  :postcondition ( and
    (on ?x ?y) (on ?y ?z) ) )
```

Figure 23: Alternate annotated tasks for the BLOCKS-WORLD domain.

Consider the annotated tasks shown in Figure 23, which could be used in the BLOCKS-WORLD domain. These tasks are used for creating piles of varying numbers of blocks, but unlike the task of Figure 5 these allow piles to be nested. That is, if `A` is on `B` and `B` is on `C`, then `A-B-C` is a 3-pile, `A-B` is a 2-pile, and `B-C` is a 2-pile, and these statements are true

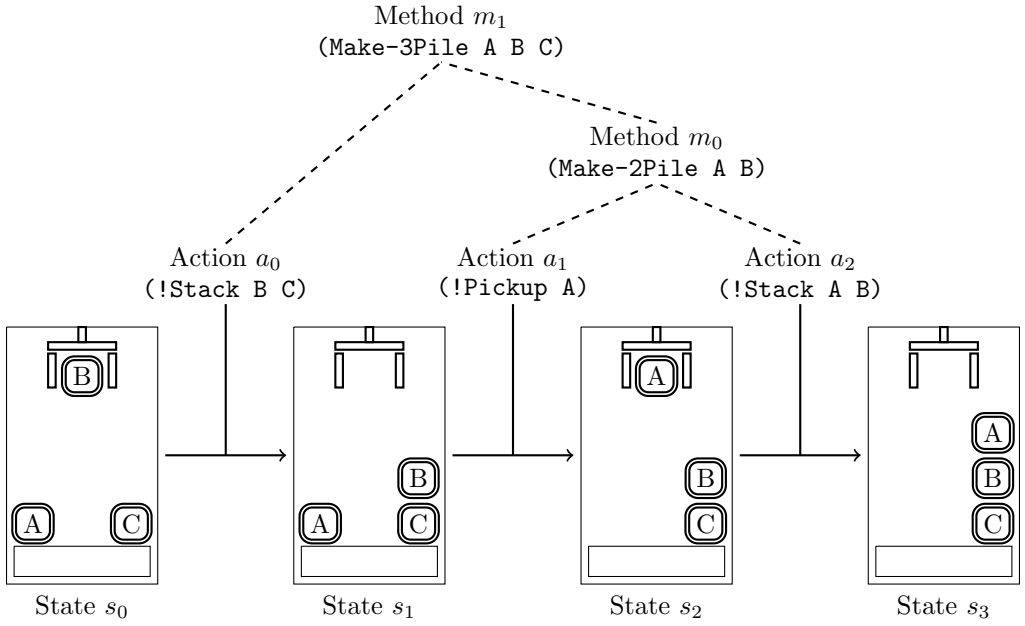


Figure 24: An example plan in the Blocks-World domain with learned methods.

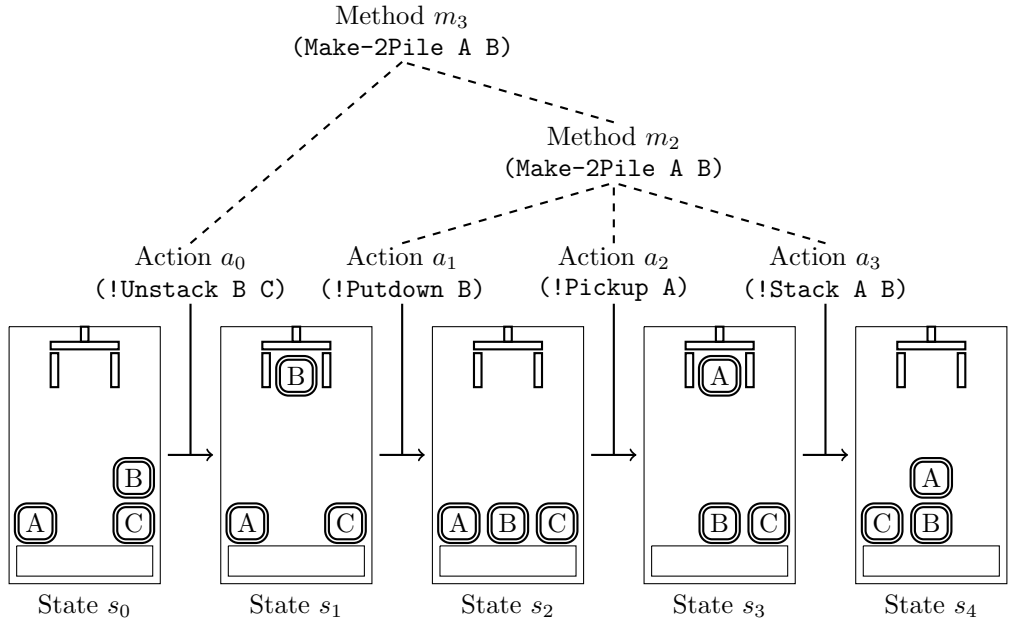


Figure 25: A second example plan in the Blocks-World domain with learned methods.

```
( :method Make-3Pile
  :parameters (?x ?y ?z)
  :precondition ( and
    (on-table ?x) (clear ?x)
    (clear ?z) (holding ?y) )
  :subtasks < (!Stack ?y ?z),
    (Make-2Pile ?x ?y) > )
```

```
( :method Make-2Pile
  :parameters (?x ?y)
  :vars (?z)
  :precondition ( and
    (on-table ?x) (clear ?x)
    (on ?y ?z) (clear ?y)
    (hand-empty) )
  :subtasks < (!Unstack ?y ?z),
    (Make-2Pile ?x ?y) > )
```

Figure 26: Details of methods m_1 (left) and m_3 (right) from Figures 24 and 25.

regardless of whether or not there are additional blocks below `C` or above `A`. This is not necessarily a wise way to define annotated tasks for the Blocks-World domain, but it is legal.

Suppose that the learning example shown in Figure 24 were an input to HTN-Maker with these annotated tasks. In addition to many others, HTN-Maker could learn methods m_0 and m_1 shown in the figure. Further suppose that the learning example shown in Figure 25 were an input to HTN-Maker with these annotated tasks. In addition to many others, HTN-Maker could learn methods m_2 and m_3 shown in the figure.

Figure 26 shows the details of methods m_1 (on the left) and m_3 on the right. Method m_1 seems logical: the first subtask puts the bottom part of the pile together, then the second subtask completes the top part of the pile. But there is no way for this method to guarantee that the way the planner chooses to accomplish the second subtask will not destroy the bottom part of the pile. Indeed, method m_3 does exactly that.

But methods m_1, m_2, and m_3 could be used by a sound HTN planner to produce the plan of Figure 27 when given that initial state and task. Note that `(on B C)` is not true in state s_5 of this figure, which means that the postconditions of the annotated task `(Make-3Pile A B C)` do not hold. Thus, this plan is a solution to the HTN planning problem but not to its equivalent classical planning problem.

D Experimental Domains

We used five different planning domains to experimentally evaluate HTN-Maker. The HTN-Maker algorithm works only for planning domains that have a classical representation. It does not support conditional effects, numerical values, temporally-extended goals, or other extensions included in the ADL language. Extending HTN-Maker to work in domains that require this more expressive representation language remains future work.

The first, Logistics, was first introduced by Veloso (1994) and was used in the 2nd International Planning Competition (IPC-2). In Logistics the objective is to deliver packages between locations in various cities using trucks for intracity transport and airplanes for intercity (and possibly intracity) transport.

The second domain, Blocks-World, has long been used as a testbed and was also used in IPC-2. This domain consists of a number of blocks sitting on a table (possibly on top of each other) and a robotic hand that can grasp one block at a time. The objective is to change the configuration of the blocks on the table using the robotic hand. While this

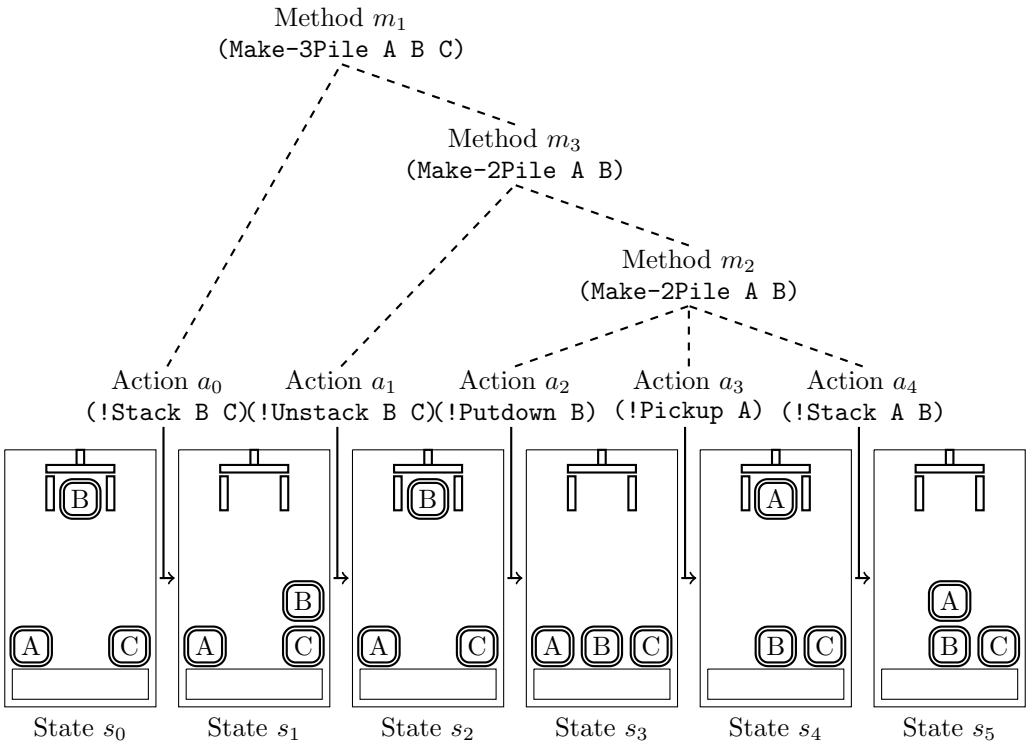


Figure 27: A plan generated from the methods learned in Figures 24 and 25.

domain is conceptually simple, large problems remain quite a challenge for planners.

SATELLITE and ROVERS were introduced for IPC-3. The SATELLITE domain involves using instruments on satellites to record images of various types and targets. The ROVERS domain involves control of a set of robots navigating Mars, taking images as in the SATELLITE domain and analyzing rock and soil samples from various locations.

The fifth, ZENO-TRAVEL, was also first used in IPC-3. It uses airplanes to transport passengers between cities. The interesting feature of the ZENO-TRAVEL domain, in the classical planning variant, is that moving an airplane requires the use of fuel, and thus airplanes may need to be refueled between flights.

In the LOGISTICS domain, our measure of problem size is the number of packages to be delivered. In the BLOCKS-WORLD domain, our measure of problem size is the number of blocks to be reorganized. In the SATELLITE domain, our measure of problem size is the number of images to be collected. In the ROVERS domain, our measure of problem size is the number of waypoints, each of which has a 33% probability of having a rock and/or soil sample. In the ZENO-TRAVEL domain, our measure of problem size is the number of passengers to be transported. In each of these domains there are many factors that determine the difficulty of a problem, but the feature we have chosen to measure problem size is that which most directly matches the number of goals the planner must achieve. For example, at each problem size in the LOGISTICS domain, some problems have twelve locations and four trucks divided between three cities, while others have twenty-three locations and six trucks divided between four cities.

The formal notion of equivalence between a classical planning problem and an HTN planning problem, as defined in Section 2.3, requires that there be a single task in the initial task network of the HTN planning problem, and that the annotations on that task include all of the goals of the classical planning problem. In practice, strictly following this scheme can make the evaluation difficult. This would require the construction of one annotated task for each possible goal atom, one annotated task for each conjunction of two goals, another for each conjunction of three goals, and so forth.

We would like to be able to learn methods from solutions to problems with few goals and use them to solve problems with many goals. Therefore, we have designed our annotated tasks in such a way that an ordered sequence of tasks, each of which represents one goal, will be equivalent to a single task that represents all of the goals together. In short, this means that the tasks must be designed in such a way that nothing the planner does to accomplish tasks t_i through t_n will remove the postcondition associated with task t_{i-1}.

```
( :task Deliver-Pkg
  :parameters ( ?obj ?dst )
  :precondition ()
  :postcondition
  ( obj-at ?obj ?dst ) )
```

Figure 28: Annotated tasks in LOGISTICS

Figure 28 contains the only annotated task used for our experiments in the LOGISTICS domain, which causes a package `?obj` to be at a location `?dst`. For a classical planning problem with goals $g_0, g_1, \ldots g_n$, the initial task network of the pseudo-equivalent HTN planning problem will consist of n instances of the `Deliver-Pkg` task with different parameters. The ordering of these tasks is arbitrary. The nature of the LOGISTICS domain is such that only

two types of actions can remove the `( obj-at ?obj ?dst )` predicate: loading the object into a truck and loading the object into an airplane. Taking either of these actions has no effect on any object in the domain other than `?obj`, and thus HTN-Maker will not learn any method for accomplishing a delivery task that loads any package into a truck or airplane other than the one that it is delivering. Thus, once an object has been delivered to its destination the planner will not move it.

```
( :task Put-On-Table                    ( :task Put-On-Block
  :parameters ( ?b )                      :parameters ( ?b1 ?b2 )
  :precondition ()                        :precondition ()
  :postcondition ( on-table ?b ) )        :postcondition ( on ?b1 ?b2 ) )
```

Figure 29: Annotated tasks in Blocks-World

The Blocks-World domain requires two tasks, which are shown in Figure 29. To preserve the postconditions of earlier tasks when accomplishing latter tasks, they must be serialized in a particular order: no block is placed until all blocks underneath it have been placed. Similarly to Logistics, it is never beneficial to move a block unless it is either the block that is being placed, currently above the block that is being placed, or currently above the block it is to be placed upon. Thus, HTN-Maker will not learn a method that moves a block that is not in one of these three categories. A block that is already in position and that has all blocks below it already in position will never fall into such a category. Thus, the more complex task shown in Figure 5 can be simulated with the sequence of tasks `<( Put-On-Table B ), ( Put-On-Block A B )>`.

```
( :task Get-Image
  :parameters ( ?dir ?mode )
  :precondition ()
  :postcondition
  ( have_image ?dir ?mode ) )
```

Figure 30: Annotated tasks in Satellite

Figure 30 contains the single annotated task used in the Satellite domain, which takes an image in a certain direction `?dir` of type `?mode`. Once an image has been collected, there is no action in the domain that can remove this fact. Therefore, this annotated task trivially satisfies our requirements regardless of ordering.

Three tasks are needed to model the Rovers domain. Figure 31 shows these three tasks, one of which is very similar to the task from the Satellite domain. The other two cause a sample of soil or rocks to be collected from a location `?loc` and information about it communicated back to the base station.Once a sample or image of any type has been collected and transmitted back to the lander, there is no action in the domain that can remove this fact. Therefore, these annotated tasks trivially satisfy our requirements regardless of ordering.

The annotated task for the Zeno-Travel domain is listed in Figure 32. It transports a passenger `?p` to a city `?c`. Like the similar Logistics domain, there is no reason to move a passenger that is currently in a city and not listed in the current task, so a passenger who is at his destination will remain there regardless of task ordering.

```
( :task Get-Soil-Data
  :parameters ( ?loc )
  :precondition ()
  :postcondition
  ( comm_soil_data ?loc ) )
```

```
( :task Get-Rock-Data
  :parameters ( ?loc )
  :precondition ()
  :postcondition
  ( comm_rock_data ?loc ) )
```

```
( :task Get-Image-Data
  :parameters ( ?dir ?mode )
  :precondition ()
  :postcondition
  ( comm_image_data ?dir ?mode ) )
```

Figure 31: Annotated tasks in ROVERS

```
( :task Transport
  :parameters ( ?p ?c )
  :precondition ()
  :postcondition
  ( person-at ?p ?c ) )
```

Figure 32: Annotated tasks in ZENO-TRAVEL

Although it is possible to define annotated tasks with richer semantics, for these experiments we have chosen to provide the minimum amount of knowledge to our system. The goals of a classical planning problem have no explicit ordering, but the tasks of a Simple Task Network problem do have a strict ordering, which must be provided by the problem's author. As discussed above, we used an arbitrary ordering in all domains except for BLOCKS-WORLD. In some circumstances this might be considered to give our STN planner, HTN-SOLVER, an advantage, because it does not need to waste time attempting possible goal serializations that cannot be completed. In other circumstances this puts HTN-SOLVER at a disadvantage, because it will be unable to interleave actions that accomplish multiple subgoals as a classical planner would.