2011

Learning Hierarchical Task Networks from Traces and Semantically Annotated Tasks

Chad Michael Hogg
Lehigh University

LEARNING HIERARCHICAL TASK NETWORKS FROM TRACES AND SEMANTICALLY ANNOTATED TASKS

by
Chad Michael Hogg

A Dissertation
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Computer Science

Lehigh University
September 2011

This dissertation is accepted in partial
fulfillment of the requirements for the degree of
Doctor of Philosophy.

_____
(Date)

_____
Héctor Muñoz-Avila

_____
Brian D. Davison

_____
Jeff Heflin

_____
Henry Baird

_____
Ugur Kuter
(Smart Information Flow Technologies)

Acknowledgements

I wish to thank the following persons, without whom this work would not have been possible:

- My wife Rachel, who lovingly and expertly brandished carrots and sticks to ensure progress over the last year.
- My parents Jeffrey and Cynthia who supported me through both my childhood and eleven years of post-secondary education.
- My advisor, Héctor Muñoz-Avila, who has been a source of funding, knowledge, inspiration, and friendship for the past five years.
- My frequent co-author Ugur Kuter, whose ideas strongly influenced Chapters 4 and 5 in particular.
- My other committee members, Brian Davison, Jeff Heflin, and Henry Baird, each of whom have been available and helpful throughout my time at Lehigh.
- My long-time colleague at the InSyTe lab, Stephen Lee-Urban, with whom I have spent many afternoons discussing his work, my work, other people's work, politics, religion, music, literature, and his persistent conspiracy theories.
- Lehigh graduate students with whom I worked for shorter periods of time, notably Philip Garcia, Christopher Creswell, and Bryan Auslander.

- My friends Keith Erekson and Sabrina Terrizzi, who have many times opened their home to me when I needed to be on campus after moving away from Bethlehem.

This work was partially supported by National Science Foundation through grant NSF 0642882.

Contents

List of Tables

List of Figures

Abstract

Hierarchical Task Network planning is a fast and highly expressive formalism for problem solving, but systems based on this formalism depend on the existence of domain-specific knowledge constructs (methods) describing how and in what circumstances complex tasks may be reduced into simpler tasks in order to solve problems. Writing and debugging a set of methods for a new domain has in the past been a difficult and error-prone manual process. This dissertation presents and evaluates a framework in which HTN methods can be automatically learned from a classical planning domain description, a set of example pairs of planning states from that domain with plans applicable in those states, and a set of annotated tasks for that domain. Annotated task are task symbols with associated preconditions and postconditions describing what it means to accomplish those tasks.

The primary algorithm based on this framework is HTN-Maker, which works by searching the input plans for subplans over which an annotated task has been accomplished and using goal regression to build a recursive series of explanations of how the task was accomplished. Each of these explanations becomes a method stating that the task may be reduced into its subtasks and giving conditions under which this will be valid. These methods that have been learned can then be used by an HTN planner to solve HTN planning problems.

The implementation of the HTN-Maker algorithm has a number of configurable options and design decisions relating to such questions as how subtasks should be grouped together, how constants should be generalized into variables, and whether or not effort should be expended in discovering and pruning unnecessary methods. Theoretical results show that if the system is configured properly,

the methods learned by HTN-Maker will accurately model the annotated tasks from which they were learned. They also show that, from a finite number of examples, HTN-Maker is capable of learning a set of methods for a domain that can be used to solve all solvable problems in that domain that can be expressed using the provided annotated tasks. There are HTN planning problems that cannot be expressed using annotated tasks, and thus cannot be solved using methods learned by HTN-Maker, but there are also non-classical problems that can be solved using the methods learned by HTN-Maker.

HTN-MakerND is an extension of HTN-Maker that learns methods that will be effective in domains in which actions do not have deterministic effects. It does so by learning methods with a carefully chosen structure that maximizes flexibility when using those methods. Q-Maker is an extension of HTN-Maker that learns both methods and estimated values of those methods intended to guide a planner toward a near-optimal plan quickly. A further algorithm, Q-Reinforce, uses reinforcement learning to refine these method values through planning experience, and Q-Shop uses the methods and values to solve planning problems, preferring to use methods with higher values when given a choice.

Experimental results in several benchmark planning domains explore the consequences of several of the options and design decisions in HTN-Maker on the amount of useful knowledge it is able to extract from examples and on the speed of planning with the methods that it learns. The evaluation demonstrates that both HTN-Maker and HTN-MakerND are able to learn knowledge that generalizes well to new problems. In four of five deterministic domains and both nondeterministic domains, planning with the learned methods is much faster than non-HTN planning. Q-Shop using methods learned by Q-Maker and refined by Q-Reinforce produces plans that are of higher quality than those produced by traditional satisficing planners while still running much more quickly than an optimal planner in both domains in which it was tested.

Chapter 1

Introduction

There have been great advances in the field of artificial intelligence through the last sixty years, and systems that employ artificial intelligence impact the lives of citizens of developed nations on a daily basis. Navigation systems assist drivers in finding unfamiliar locations or routing around unexpected obstacles, autonomous vacuum cleaners clean homes while their owners are away, and search engines direct a user to information sources that precisely match her query. Artificial intelligence has been used to improve activities as important as diagnosing disease and as frivolous as playing chess or `Jeopardy!`.

In spite of these unqualified successes, there remain many human endeavors in which artificial intelligence technology has not been leveraged, although it would surely be beneficial. This is in part because traditional, symbolic artificial intelligence systems are knowledge-intensive. The proliferation of the Internet has meant that *data* is readily available, but *knowledge* is something more: a context for data, or a theory explaining anecdotes, and is much rarer. There are many exceptions, but knowledge must often be created or curated by a human expert, in a process known as knowledge engineering.

One field of symbolic artificial intelligence that has generated persistent interest throughout the age of artificial intelligence is automated planning, the process by which a machine can reason about actions that it might take to devise a plan that will achieve the system's goals. Traditionally, automated planning systems have

required knowledge in the form of formal descriptions of actions that specify what must be true before an action may be taken (preconditions) and how taking that action will change the world (effects). These formal descriptions typically represent the state of the world and preconditions and effects of actions using the language of first-order logic.

Producing the knowledge constructs representing actions is usually only a moderately difficult process, but more knowledge generally produces more useful systems. Domain-configurable planning paradigms allow or require the use of additional domain-specific knowledge, such as structural properties of the domain or domain-specific problem-solving strategies. This knowledge allows domain-configurable planners to be potentially much more useful than traditional, domain-independent planners, but greatly increases the knowledge engineering burden.

Hierarchical Task Network (HTN) planning is one of the most well-known domain-configurable planning paradigms. The most common type of HTN planner does not have explicit state-based goals to achieve; rather it has tasks that it wishes to accomplish. It does so by using an additional knowledge structure, known as a method. A method consists of a task that it is working toward (the head), a description of what must be true before the method may be used (preconditions), and a series of tasks that, if they are accomplished, will result in the head task being accomplished (subtasks). The lowest-level (primitive) tasks correspond to actions that can be performed directly, and an HTN planner continues using methods to reduce complex tasks into subtasks until all remaining tasks are primitive.

SHOP [63] and its successor SHOP2 [62] have been the most widely used HTN planning systems over the last decade. Because methods encode a great deal of knowledge about a domain, SHOP and SHOP2 are capable of solving problems much more quickly than classical planners, so much so that a separate track has been created for domain-configurable planners at the international planning competitions.

Because tasks are more general than explicit goals, HTN planners are capable of solving problems that cannot even be expressed in the language of classical planners. Furthermore, it is theorized that humans naturally solve complex problems by breaking them down into simpler problems as an HTN planner does [47]. As a

result, HTN planning has proven to be an effective framework for modeling many real-world applications, including military tactics [55, 59], strategy formulation in computer games [27], manufacturing processes [64], project management [95], and story-telling [7].

Countering these significant benefits of the HTN planning formalism is the requirement of the additional knowledge structure, methods. Without them, an HTN planner will be incapable of solving any problems. Developing and debugging a set of HTN methods that will allow an HTN planner to efficiently solve problems in a domain has been a challenging manual task, requiring someone who is both an expert in the domain being modeled and an expert in the planning formalism. This process must be repeated, from scratch, for any new domain in which one would like to use an HTN planner.

To alleviate this knowledge engineering burden, several researchers have studied techniques in which hierarchical knowledge for planning could be learned automatically. Most of these works require that some type of knowledge be provided by a human beyond the preconditions and effects of actions. In some cases this means axiomatic concepts [47, 66], in others the desired relationships between tasks and subtasks [95, 32], and so forth.

This dissertation presents a new framework for learning hierarchical planning knowledge without these requirements, Hierarchical Task Networks with Minimal [1] Additional Knowledge Engineering Required (HTN-Maker) [2]. The HTN-Maker framework requires only that a human list what the tasks are that the planner should learn how to accomplish and for each what it means to accomplish that task, through a knowledge structure called an *annotated task*. HTN-Maker searches through example plans to find subplans through which a task had been accomplished. It then analyzes those subplans, using a hierarchical extension to goal regression to

[1] The knowledge requirements of HTN-Maker are not necessarily *minimal* in a formal, mathematical sense.

[2] Source code for HTN-Maker and related algorithms is available at `http://www.cse.lehigh.edu/InSyTe/HTN-MAKER/`, as is a copy of this dissertation, other publications related to the project, and data from some experiments.

determine how and why the task had been accomplished. Based on this explanation, it produces a recursive series of methods that can be used by an HTN planner to accomplish that task in similar situations.

1.1 Contributions

This dissertation explores the automatic generation of knowledge for a particular category of knowledge-intensive artificial intelligence systems, Hierarchical Task Network planners. It introduces a formalism for annotating purely symbolic tasks with semantics that describe what it means to accomplish those tasks and a framework for reasoning about these annotated tasks and plans to produce knowledge structures that can be used by HTN planners to solve new problems. It explains how this framework can be extended to learn knowledge structures that not only allow HTN planners to find solutions, but to find high-quality solutions quickly.

I formulate and prove theorems regarding the properties of this framework, including the soundness of the knowledge constructs created and the limits of the knowledge than can be learned and the types of problems that can be solved using it. I explain in detail several decisions that must be made when implementing this new framework and their implications for the sort of knowledge that will be learned. I demonstrate the utility of the knowledge that can be learned through a suite of experimental evaluations.

1.2 Outline

Chapter 2 formally defines the classical and HTN planning problems and includes an overview of planning technologies in each formalism. It also discusses the problem of extending the classical and HTN formalisms to work with domains in which actions have multiple possible outcomes and the related technology of reinforcement learning.

Chapter 3 formalizes the notion of an annotated task and the HTN-Maker

framework for learning HTN methods. It includes an extended example of the execution of HTN-Maker and describes several choices made in implementing the algorithm. Additionally, it formulates and proves theorems about the soundness and completeness of the learning algorithm, the types of problems that can be expressed and solved using the learned methods, and the time complexity of learning.

Chapter 4 explains how the HTN-Maker algorithm can be implemented in a way that will learn useful methods from domains in which actions have multiple possible outcomes, producing a new system HTN-MakerND. It includes theorems and their proofs that the soundness and completeness properties of HTN-Maker are preserved in HTN-MakerND.

Chapter 5 discusses an integration of HTN-Maker with reinforcement learning to learn methods that can be used to find high-quality plans. This integration consists of three related systems: Q-Maker, which learns both methods and initial estimates of the value of those methods, Q-Reinforce, which updates these values based on its experiences planning with the methods, and Q-Shop, which uses the methods with values to find high-quality solutions to planning problems.

Chapter 6 reports on a variety of experiments used to evaluate HTN-Maker and related algorithms. It first considers the rate at which HTN-Maker learns from examples, comparing four different configurations of the algorithm. The next section discusses the speed at which an HTN planner is able to solve problems using methods learned by HTN-Maker, comparing to classical planners and to the same HTN planner using methods written by a domain expert. A similar experiment is performed with HTN-MakerND learning from nondeterministic domains. The final section compares the speed of planning and quality of solutions produced by Q-Shop using methods and values learned by Q-Maker and Q-Reinforce to several classical planners.

Chapter 7 discusses prior work that has been done in machine learning of knowledge for automated planning systems, and in particular other systems that learn structures similar to HTN methods. Chapter 8 summarizes the finding in this document and discusses promising avenues for future work.

Chapter 2

Background

The problem of automated planning is to find a sequence of actions (a plan) that will transform one specific state into one of a set of goal states. At its core, automated planning is thus a special case of the graph search problem in which states represent nodes of a graph and actions edges. However, the types of problems addressed in automated planning typically contain far too many states and actions to be represented explicitly in a reasonable amount of time or space. The field of automated planning was created when researchers adopted a logic-based formalism that enables algorithms to reason about large numbers of states and actions without representing all of them explicitly.

The General Problem Solver (GPS-I) [67] was the first major system to address what was then called problem solving. One of the most important features of the General Problem Solver was a decoupling of the algorithm and the description of the problem, such that it could easily be configured to solve various types of problems. This would be key to the later development of domain-independent planning as a field of AI. Although GPS-I introduced the concept of operators that transform objects and the reasoning strategy means-end analysis, it would not be recognizable as a planner today.

Cordell Green took the next major step toward planning as a discipline by introducing the use of logical theorem proving as a reasoning mechanism [23]. His QA3 system formalized states in first-order logic and actions as functions over the set of

states.

What has become known as classical planning was finally formalized in the STRIPS system of Fikes & Nilsson [18], which has been so influential that classical planning is also often referred to as STRIPS planning. Further advancements in the field have used novel algorithms with variants of the STRIPS problem representation.

I begin by defining an abstract notion of planning problems, then show how the classical representation provides an efficient implementation. Although automated planning is in some ways a unified field, many researchers have developed their own way of defining problems and solutions. I have done the same in an attempt to unify notations for classical and HTN planning and to make both as simple as possible. The definitions in this chapter are strongly influenced by those of Ghallab, Nau, and Traverso [22], but not identical. I defer discussion of research in machine learning for automated planning to Chapter 7.

Definition 1. A **generic planning domain** is a 3-tuple $\Sigma[g] = (S, A, \gamma)$, where S is a finite set of **states**, A is a finite set of **actions**, and $\gamma : (S \times A) \rightarrow S$ is a partial function known as the **state-transition function**. The postscript $[g]$ notates that this is a generic domain, rather than a classical ($[c]$) or HTN ($[h]$) domain.

Definition 2. A **generic planning problem** is a 3-tuple $\Psi[g] = (\Sigma[g], s_0, G)$, where $\Sigma[g] = (S, A, \gamma)$ is a generic planning domain, $s_0 \in S$ is the **initial state**, and $G \subseteq S$ is the set of **goal states**.

Definition 3. A **plan** $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ is a finite sequence of actions.

Definition 4. A **state trajectory** $\vec{s} = \langle s_0, s_1, \ldots, s_m \rangle$ is a finite sequence of states.

Definition 5. Given a generic planning domain $\Sigma[g] = (S, A, \gamma)$, a generic planning problem $\Psi[g] = (\Sigma[g], s_0, G)$, and a plan $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ such that each action in π is a member of A, π is a **solution** to $\Psi[g]$ if there exists a state trajectory $\vec{s} = \langle s_0, s_1, \ldots, s_{n+1} \rangle$ beginning with the initial state such that $\forall (0 < i \leq n+1), s_i = \gamma(s_{i-1}, a_{i-1})$ and $s_{n+1} \in G$.

Intuitively, a plan is a solution if it is possible to apply the actions in the plan to the initial state and doing so reaches a final state that is a member of the set of goal states.

2.1 Classical Planning

The abstract notion of a generic planning domain and problem defined above could be implemented in several different ways. The most straightforward representation is a labeled directed multigraph where the members of S are the nodes, the members of A are labels that are applied to the edges (each edge has exactly one label but the reverse is not true, and there might be multiple edges between two nodes with different labels), and the values of γ specify what edges exist and what their labels should be. For large problems, however, the size of the sets of states and actions will be unmanageable. Furthermore, this implementation does not contain any knowledge about the relationships between states other than the existence of transitions between them, and it will thus be impossible to find plans more efficiently than a general-purpose algorithm for finding paths in a directed graph.

Early planners such as STRIPS [18] used a representation based on propositional logic, in which a state was a set of propositions that held concurrently and an action deleted some propositions from a state while adding others. This meant that the set of states did not need to be represented explicitly; rather one state could be created from another by applying actions as necessary. However, the set of actions still needed to be specified and could be quite large. Because states and actions had a meaningful internal representation, it was possible for planning algorithms to reason about states and actions and thus use an informed, rather than a blind, search.

Other planners such as SAS+ [2] have represented a state as the values of a set of variables and actions as changes to the values of certain variables. The most common representation of states and actions, however, is based on first-order logic. This is the representation that I will use throughout this document.

2.1.1 Definitions For Classical Planning

Definition 6. A **constant** is a symbol that refers to one specific object, while a **variable** is a symbol that represents an as-yet unspecified object. A **term** is either a constant or a variable. All terms are represented as strings of characters. We will follow the convention that the first character of a variable is a question mark. Figure 2.1 shows a variety of example terms. The first, third, and fifth are constants, while the second and fourth are variables.

```
block072  ?a  lskdjf0j  ?TheTruck  Aristotle
```

Figure 2.1: Several example terms

Definition 7. A **predicate** is a template for a type of simple statement about the world. It consists of a **predicate symbol** and a non-negative number which is its **arity**.

Definition 8. An atomic formula, or **atom**, is a specific statement about the world. Syntactically, an atom consists of an opening parenthesis, a predicate symbol, a number of terms equal to the arity of the predicate called the **arguments**, and a closing parenthesis. If an atom contains no variables, then it is **ground**. Figure 2.2 shows a variety of example atoms.

```
(On block072 block231)              (Hand-Empty)
(IsPhilosopher Aristotle)  (Truck-At ?TheTruck PackardLab)
(asdf lskdjf0j)                     (a ?b ?c d ?e)
```

Figure 2.2: Several example atoms

Definition 9. A **state** s is a finite set of ground atoms, representing all statements that are true at some particular point in time. We use the closed-world assumption, which means that any atom that does not appear in a state is explicitly false. A ground atom **holds** in a particular state if it is a member of that state.

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

(a) First State

```
(on-table A)
(on-table B)
 (clear A)
 (clear B)
(holding C)
```

(b) Second State

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

(c) Third State

Figure 2.3: Three example states from the BLOCKS-WORLD domain

Figure 2.3 shows three example states. These states and further examples throughout this document are taken from the BLOCKS-WORLD domain [93]. This domain models a number of cubical blocks sitting on a table (possibly on top of each other) and a robotic hand that can hold one block at a time. The five predicates in this domain have semantics that a particular block is directly on the table, on top of another specific block, clear (sitting without anything above it), held by the robotic hand, or that the robotic hand is empty. In the first state, block `A` is on the table with block `C` above it, block `B` is on the table, and the robotic hand is empty. In the second state, blocks `A` and `B` are on the table and block `C` is held by the robotic hand. In the third state, block `A` is on the table, block `B` is on the table with block `C` above it, and the robotic hand is empty. Because of the closed-world assumption, we know that there does not exist a block `D` in any of these states, and that block `C` is not on the table.

Definition 10. A **substitution** u is a collection of variable-term pairs. The result of **applying** a substitution to an atom is a copy of that atom except that any variables that appeared in it and that were the first part of a pair in the substitution are replaced by the second part of that pair in the substitution.

Definition 11. An **action** is a four-tuple $a = (a^h, a^\phi, a^-, a^+)$. The **head** of an action (a^h) has a similar form to that of an atom: an opening parenthesis, the name of the action, 0 or more parameters, and a closing parenthesis. All of the parameters of the head of an action must be constants. By convention, the names of actions begin with an exclamation point. The **preconditions** (a^ϕ), **negative effects** (a^-),

and **positive effects** (a^+) of an action are finite sets of atoms whose parameters all appear in the head of the action.

Definition 12. An action $a = (a^h, a^\phi, a^-, a^+)$ is **applicable** to a state s if and only if each member of the preconditions of the action is also a member of the state. Formally, a is applicable to s iff $a^\phi \subseteq s$. The **result** of applying an applicable action a to a state s is a new state s' that is a copy of s from which the negative effects of a have been removed and to which the positive effects of a have been added. That is, $s' = (s \setminus a^-) \cup a^+$. The result of applying an inapplicable action to a state is undefined.

```
( :action
  :head
  (!Unstack C A)
  :precondition
  { (on C A),
    (clear C),
    (hand-empty) }
  :negative-effects
  { (on C A),
    (clear C),
    (hand-empty) }
  :positive-effects
  { (clear A),
    (holding C) }
)
```

(a) First Action

```
( :action
  :head
  (!Unstack C B)
  :precondition
  { (on C B),
    (clear C),
    (hand-empty) }
  :negative-effects
  { (on C B),
    (clear C),
    (hand-empty) }
  :positive-effects
  { (clear B),
    (holding C) }
)
```

(b) Second Action

Figure 2.4: Two example actions from the BLOCKS-WORLD domain

Some researchers define states and the preconditions and effects of actions to be conjunctive logical formulas rather than sets of atoms. The set-based representation is easier to discuss and visualize and is sufficient to model the knowledge used in this document, so I have chosen it.

Figure 2.4 shows two example actions in the BLOCKS-WORLD domain. This and many other examples throughout the text use a language of my own devising. It is heavily inspired by the Planning Domain Description Language (PDDL) [21], but modified to be more verbose, to more explicitly match the set-based representation used in this document, and to support additional data structures. The action shown in Figure 2.4a unstacks block `C` from on top of block `A`. This action will be applicable to any state in which block `C` is on top of block `A`, block `C` has nothing on it, and the robotic gripper is empty. The state shown in Figure 2.3a is one of many in which this action is applicable. The result of applying this action to that state would be the state shown in Figure 2.3b. The action shown in Figure 2.4b unstacks block `C` from on top of block `B`. It is not applicable in the state of Figure 2.3a, but would be from the state of Figure 2.3c. The result of applying this second action to the state of Figure 2.3c would be the state of Figure 2.3b.

Definition 13. An **operator** is a four-tuple $o = (o^h, o^\phi, o^-, o^+)$. The names and semantics of the components of an operator are the same as those of an action with one exception: the head (and thus, the other components) of an operator may include variables.

Definition 14. The **result** of applying a substitution u to an operator o (notated $u(o)$) is a copy o' of that operator in which variables in o have been replaced by their corresponding term (if there is one) in u. If the resulting operator o' is ground, then it is also an action and called an **instantiation** of the operator. As shorthand, we will say that an operator o is applicable to a state s iff there exists a substitution u such that $u(o)$ is an action that is applicable to s.

Figure 2.5 shows the operators in the BLOCKS-WORLD domain. Figure 2.5a is a generalized version of the two unstacking actions shown in Figure 2.4; it can be used to unstack any block `?above` from any block `?below` that satisfy its preconditions. The action of Figure 2.4a is the result of substituting `C` for `?above` and `A` for `?below`, while the action of Figure 2.4b is the result of substituting `C` for `?above` and `B` for `?below`. The operator of Figure 2.5c has an opposite effect, causing one block to

be placed on another. There are two different substitutions that would make this operator applicable in the state of Figure 2.3b: {`?above` / `C`, `?below` / `A`} and {`?above` / `C`, `?below` / `B`}. The result of applying this operator with the first of these substitutions would be the state of Figure 2.3a, while doing so with the second substitution would yield the state of Figure 2.3c.

Definition 15. A **classical planning domain** is a triple $\Sigma[c] = (C, P, O)$ where C is a finite set of constants, P is a finite set of predicates, and O is a finite set of operators. Every atom that appears in one of the operators in O must correspond to one of the predicates in P.

Because of the logic-based representations of states and actions, the components of a classical planning domain fully specify a generic planning domain. The set of actions can be derived by generating all instantiations of the operators, while the set of states is the power set of the set of all possible atoms, which can be generated from the set of constants and the set of predicates. The state-transition function is implicit in the actions themselves and what it means to apply an action to a state.

A classical planning domain is typically written by a human expert as a way to formalize the types of statements that are relevant to a domain and the way in which actions interact with states made of those relevant statements. As such, the designer of the classical planning domain typically has a semantic interpretation in his own mind for the symbols he is using. These semantics, however, are not explicitly represented in the operators of the classical planning domain. Thus, while it makes sense to the reader that `(On A B)` and `(On A C)` could not be simultaneously true, a planning system has no similar insight. The only way the domain designer can ensure that this is the case is to create operators that will not add one of these unless it also deletes the other or contains preconditions that preclude it from being true.

The discretization of abstract concepts into formal constructs is in some ways a creative process, and two people may design very different classical planning domains to describe the same class of problems. In addition to the domain description that we use, there is another conceptualization of BLOCKS-WORLD that uses three operators

```
( :operator
  :head
  (!Unstack ?above ?below)
  :precondition
  { (on ?above ?below),
    (clear ?above),
    (hand-empty) }
  :negative-effects
  { (on ?above ?below),
    (clear ?above),
    (hand-empty) }
  :positive-effects
  { (clear ?below),
    (holding ?above) }
)
```

(a) Operator `!Unstack`

```
( :operator
  :head
  (!Pickup ?it)
  :precondition
  { (on-table ?it),
    (clear ?it),
    (hand-empty) }
  :negative-effects
  { (on-table ?it),
    (clear ?it),
    (hand-empty) }
  :positive-effects
  { (holding ?it) }
)
```

(b) Operator `!Pickup`

```
( :operator
  :head
  (!Stack ?above ?below)
  :precondition
  { (clear ?below),
    (holding ?above) }
  :negative-effects
  { (clear ?below),
    (holding ?above) }
  :positive-effects
  { (on ?above ?below),
    (clear ?above),
    (hand-empty) }
)
```

(c) Operator `!Stack`

```
( :operator
  :head
  (!Putdown ?it)
  :precondition
  { (holding ?it) }
  :negative-effects
  { (holding ?it) }
  :positive-effects
  { (on-table ?it),
    (clear ?it),
    (hand-empty) }
)
```

(d) Operator `!Putdown`

Figure 2.5: Four example operators from the BLOCKS-WORLD domain

and does not consider states in which the robotic hand is holding a block. Instead, it has one operator that is equivalent to a `!Pickup` followed by a `!Stack`, a second that is equivalent to a `!Unstack` followed by a `!Stack`, and a third that is equivalent to a `!Unstack` followed by a `!Putdown`. Either of these could be further modified by representing the table explicitly as a constant.

Definition 16. A **classical planning problem** is a triple $\Psi[c] = (\Sigma[c], s_0, g)$, where $\Sigma[c] = (C, P, O)$ is a classical planning domain, s_0 is the **initial state**, and g is a finite set of ground atoms called the **goals** of the problem. Each of the atoms in s_0 and g must be a specialization of a predicate in P with constants from C.

As is the case with a domain, a classical planning problem represents a generic planning problem. We have already seen how the domain components are equivalent, and the initial states are the same. The set of goal states of a generic planning problem correspond to those states that contain every member of the goals of a classical planning problem.

Definition 17. Given a classical planning domain $\Sigma[c] = (C, P, O)$, a classical planning problem $\Psi[c] = (\Sigma[c], s_0, g)$, and a plan $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ in which each action is an instantiation of an operator in O with constants from C, π is a **solution** to $\Psi[c]$ if there exists a state trajectory $\vec{s} = \langle s_0, s_1, \ldots, s_{n+1} \rangle$ beginning from the initial state such that $\forall(0 < i \leq n+1)$, a_{i-1} is applicable to s_{i-1} and s_i is the result of applying a_{i-1} to s_{i-1}, and $g \subseteq s_{n+1}$. Intuitively, a plan is a solution if it is possible to apply the actions in plan to the initial state and doing so reaches a final state in which the goals hold.

I follow the lead of many researchers in implementing one minor refinement of classical planning in which each constant and variable has an associated type, which is purely symbolic. In a substitution where a variable is replaced by a term, both must be of the same type. Using typed variables in predicates and operators makes it much easier to debug problems in classical planning domains and can provide a modest improvement in performance when a planner needs to consider only those constants of a particular type. Adding types does not affect the expressivity of

classical planning, because they can be modeled as creating a new predicate for each type, making these predicates true for the constants of those types in the initial state, and including these predicates as preconditions of operators for the types of their variables. The BLOCKS-WORLD domain does not have multiple types of constants, but most others do. In examples where types are needed, a variable declaration consists of the name, a dash, and then the type.

2.1.2 Classical Planning Systems

Quite a number of classical planning systems have been devised, and a competition among classical planners is held every two years to evaluate new ideas and enhancements. The primary metric by which planning systems are judged is their running times on various problems. Through advances in representation and reasoning, and to a lesser extent hardware upgrades, modern classical planners are able to solve in seconds problems that would have been infeasible when the field was new. In recent years there has been some focus on not simply finding a solution quickly, but finding an optimal solution (minimizing the number of actions taken or, in more sophisticated representations, the total cost of the plan). The other driver of advances in planning technology is expanding classical planning to more expressive paradigms, allowing the reasoners to work directly with temporal constraints, numeric quantities, and resources.

The simplest algorithm for solving a classical planning problem begins by generating a list of all actions a that are applicable in the initial state s_0, selecting one, and generating the subsequent state $s' = \gamma(s_0, a)$. The algorithm then continues extending the plan from state s' until it reaches a state in which the goals g hold. This is called forward-chaining state-space planning, because it explores the graph of states by iterating forward from the initial state. Although forward-chaining state-space planning is one of the simplest and earliest approaches, many of the best modern planners are based on the same ideas, because knowledge of the current state enables many powerful reasoning techniques. As with other search problems, nodes may be explored in a variety of different orders: depth-first, breadth-first,

iterative deepening, etc, and standard techniques may be used to prune nodes or detect loops in the graph [77].

Algorithm 1 shows pseudocode for a simple forward-chaining, state-space classical planner, FcSsC-Plan. The algorithm begins by checking if the problem is trivially solvable (Line 3) and, if so, returns the empty plan as a solution (Line 4). Next, it checks for an action that is applicable in the initial state (Line 5). If one (or more) are found, one is selected nondeterministically (Line 6), and the state that results from applying that action to the initial state is calculated (Line 7). The selected action is then prepended to the solution to a new problem to achieve the goals from this new state (Line 8). If that problem has no solution, the algorithm backtracks and nondeterministically chooses a different action or, if none are available, reports that no solution to the current problem can be found (Line 9). Because the FcSsC-Plan algorithm follows the definition of a solution directly, it should be clear that it is both sound and complete.

Algorithm 1: A forward-chaining state-space classical planner

```
1  Procedure FcSsC-Plan(Ψ[c] = (Σ[c] = (C, P, O), s_0, g))
2  begin
3      if g ⊆ s_0 then
4          return ⟨⟩
5      if ∃((o ∈ O, u)|u(o^φ) ⊆ s_0) then
6          Nondeterministically select such an o and u
7          Compute new state s' ← (s_0 \ u(o^-)) ∪ u(o^+)
8          return ⟨u(o)⟩· FcSsC-Plan(Ψ'[c] = (Σ[c], s', g))
9      return FAIL
10 end
```

An alternative approach is to begin from the goals g and generate a list of actions a that achieve one or more goals, select one, and add its preconditions to the list of goals to achieve. When the list of outstanding goals are all true in the initial state, the collected actions can be executed in reverse order to transform the initial state into a state in which the goals of the classical planning problem hold. This is called backward-chaining state-space planning, even though the algorithm does not

consider full states during the search. Because there are often fewer actions that achieve a certain goal than there are actions applicable in a certain state, backwards-chaining state-space planning can be much faster than forward-chaining state-space planning in certain domains.

The STRIPS system [18] that popularized a logic-based representation for states and actions used a variant of backward-chaining state-space planning in which plans for different goals may not be interleaved. What this means is that all actions necessary to achieve the first goal must precede all actions necessary to achieve the second goal, and the same is true recursively of subgoals created from the preconditions of selected actions. This restriction greatly prunes the search space and allowed STRIPS to solve larger problems than comparable systems. However, there are problems that cannot be solved without subgoal interleaving (formally, their goals are not trivially serializable), and STRIPS is incapable of solving them.

A particularly well-known example of a problem in the BLOCKS-WORLD domain that the STRIPS system cannot solve is the Sussman anomaly [81]. The state shown in Figure 2.3a is the initial state of the Sussman anomaly, and the goal is to have block `A` on block `B` and block `B` on block `C`. The straightforward way to accomplish the first goal is to unstack `C` from `A`, place `C` on the table, pick up `A`, and stack `A` on `B`. However, any attempt to place `B` on `C` after doing this will require removing `A` from `B`. If the planner instead tries the second goal first, it will simply pick up `B` and stack it on `C`, but again it will need to un-do this to accomplish the first goal.

The first major shift in planning algorithms was from a state-space perspective, as described above, to a plan-space perspective, in which the search is over the set of all possible plans. The primary data structures in plan-space planning are plan steps, binding constraints, ordering constraints, and causal links. Plan steps are simply instantiations of operators, with binding constraints specifying the bindings of variables to constants. Ordering constraints specify that one plan step must occur before a second, and a causal link denotes that a precondition of one step depends on a positive effect of another. The basic plan-space planning algorithm begins with two dummy steps – one with no preconditions and the atoms from the initial state as its positive effects, and one with no effects and the goals of the problem as

its preconditions. The algorithm proceeds by solving a flaw in the plan (either a precondition of a step that is not supported by a causal link or a causal link that is threatened by another step that could delete the effect between the two steps) by introducing either another step or constraint until it arrives at a plan with no flaws. The flexibility of this approach proved much more efficient than previous state-space algorithms. The ideas of plan-space planning were developed by many people over a number of years, but the best-known planner to incorporate them was UC-Pop [71].

The next major algorithm, GraphPlan [5], was provided by researchers outside of the automated planning community. Rather than working directly in either the state or plan space, GraphPlan operates on a new data structure called a planning graph. The planning graph consists of nodes in a sequence of levels and edges between nodes in different levels. The first level contains all of the atoms from the initial state as its nodes, while the second level contains all actions that are applicable in the initial state as its nodes. The third level consists of all atoms that are in the first level as well as those produced by any member of the second level, while the fourth level consists of all actions that are applicable based on the atoms in the third level. In this way, every odd level represents all facts that could potentially be true (although not necessarily simultaneously) after a certain number of actions have been taken.

There are three types of edges within a planning graph. First, there are edges from the nodes in each atom level to the actions in the following action level representing the relationship that the atom is a precondition of the action. Similarly, there are edges from the actions in each action level to their positive and negative effects respectively in the following atom level.

When an even level is reached that contains all elements of the goals, the algorithm attempts to search backwards through the planning graph for a solution. This backwards search marks the goal atoms in their level, then the actions that produce them in the previous level, then the atoms that are the preconditions of the actions in the level before that, and so forth to the level representing the initial state. An action may not be marked if it has a negative effect that is marked in the following

level. If this search fails, it means that there have not yet been enough levels for all of the goals to be *simultaneously* true. Thus, the algorithm continues expanding the planning graph until a solution can be found. The maximum necessary size of the planning graph is limited by the number of actions in the shortest solution.

While GraphPlan was quite effective in itself, it has become even more influential as a subroutine of heuristic planners. FastForward [28] is a forward-chaining state-space planner that uses a hill-climbing search strategy, and a state heuristic calculated by a variant of GraphPlan. Specifically, to calculate the heuristic value of a state s, FastForward runs GraphPlan from that state with a version of the domain in which the negative effects of all operators are removed and counts the number of steps in the solution to this relaxed version of the problem. The removal of negative effects makes GraphPlan very fast because the search backward through the planning graph becomes trivial, but the plans generated are unlikely to be actual solutions to the original problem. The size of these plans is a useful heuristic, however, because it never understates the true cost of finding a solution from that state. In fact, use of this relaxed planning graph heuristic has proven very efficient. Most classical planning systems since FastForward have used further enhancements in heuristic search by using more accurate heuristics, applying heuristics in plan-space planners, or adjusting the search strategy used.

There have been many other systems designed to solve classical planning problems, of which I will mention only a few. Kautz and Selman [37] introduced the idea of casting planning as a Boolean satisfiability problem and using existing algorithms from that field with the SatPlan system. Others have recast planning as a constraint programming problem, such as CPlan [87]. The SGPlan6 [31] system attempts to partition the problem into multiple sub-problems that can be solved in parallel, then combines the solutions to these sub-problems to create a solution to the original problem.

The primary drive behind these systems and other innovations in classical planning has been new techniques for finding plans as quickly as possible. Because the question of whether or not a classical planning problem even has a solution is in the

PSPACE complexity class [14], algorithms that can solve even moderately larger problems in reasonable time and space are considered significant contributions. Many of these systems have also introduced or supported more complex representations such as negative preconditions, conditional effects, and domain axioms. These representational advancements greatly simplify the process of modelling complicated domains, but do not fundamentally alter the problem because they may be converted into a strict classical planning representation.

Other works have changed more fundamental assumptions about classical planning, such as including numbers and arithmetic, resources, and time as first-class concepts about which the planner may reason, or allowing exogenous events or incomplete state information. These extensions allow more complex and interesting domains to be modelled, but significantly complicate planning algorithms.

All of the systems mentioned thusfar have been *domain-independent*, meaning that the process used to find a plan is entirely orthogonal to the details of the states and actions used to model the domain. When certain characteristics of a domain are understood, it may be possible to design a much more efficient *domain-dependent* algorithm that will only work for problems in that domain. More recently, several researchers have proposed *domain-configurable* planners, in which some domain-specific knowledge is used to guide the search algorithm toward a solution. Because this domain-specific knowledge constrains the possible solutions to problems, domain-configurable planners are not classical.

TL-PLAN[1] is one such domain-configurable planner, which uses domain-specific temporal logic formulas. Temporal logic formulas are first-order logic formulas enhanced with modal operators that can specify that something must be true of some state, all states, the next state, and so forth. TL-PLAN uses a depth-first forward-chaining state-space search in which each state trajectory is pruned if it does not satisfy the control formula. With no control formula, TL-PLAN will search randomly and thus perform very poorly. With a well-written domain-specific control formula, however, it can be faster than any known domain-independent planner. The other major category of domain-configurable planners use Hierarchical Task Network planning, which is described in the next section.

2.2 HTN Planning

The idea behind Hierarchical Task Network (HTN) planning is that high-level, abstract *tasks* can be broken down into simpler subtasks. The simplest tasks correspond directly to actions in a classical planning domain, while the more complex tasks and descriptions of the way in which they may be replaced by subtasks (called *methods*) are the domain-specific knowledge used to configure the planner. Unlike TL-Plan, in most HTN planners this domain-specific knowledge is not an optional feature that may enhance performance. Rather, it is a requirement for the planning process. One way to think of this is that temporal logic control formulas guide the search by specifying some paths that *should not* be explored, while HTN methods specify those paths that *may* be explored.

2.2.1 Definitions For HTN Planning

Definition 18. A **task template** is a template for a symbolic representation of an activity in the world. It consists of a **task name** and a non-negative number which is its **arity**.

Definition 19. A **task** is a specific activity that may be undertaken in the world. Syntactically, a task consists of an opening parenthesis, a task name, a number of terms equal to the arity of the task name called the **arguments**, and a closing parenthesis. If a task is equivalent to the head of an action or operator (see Definition 13), then the task is **primitive**. Otherwise, it is **nonprimitive**.

Definition 20. A **task network** $w = \langle t_0, t_1, \ldots t_n \rangle$ is a fully-ordered finite sequence of tasks.

The definition of a task network used here is specific to a particular variant of HTN planning called Ordered Task Decomposition (OTD). This is the dominant form of HTN planning today, and the formalism used throughout this paper. In other types of HTN formalisms, the tasks in a task network do not need to be totally ordered and may also contain other types of constraints. See Section 2.2.2 for details regarding more general HTN planning systems.

Definition 21. A **method** $m = (m^h, m^\phi, m^w)$ is a triple in which m^h is a nonprimitive task called the **head** of the method, m^ϕ is a finite set of atoms known as the **preconditions** of the method, and m^w is a task network called the **subtasks** of the method. Unlike actions and operators, the preconditions and subtasks of a method may contain terms that do not appear in the head of the method. The result of applying a substitution to ground a method is an **instantiation** of that method.

```
( :method
  :head
  (Make-2Pile ?above ?below)
  :precondition
  { (on-table ?below),
    (clear ?below),
    (holding ?above) }
  :subtasks
  < (!Stack ?above ?below) >
)
```

(a) First Method

```
( :method
  :head
  (Make-2Pile ?above ?below)
  :vars
  { ?other }
  :precondition
  { (on-table ?below),
    (clear ?below),
    (on ?above ?other),
    (clear ?above),
    (hand-empty) }
  :subtasks
  < (!Unstack ?above ?other),
    (Make-2Pile ?above ?below) >
)
```

(b) Second Method

Figure 2.6: Two example methods from the BLOCKS-WORLD domain

Figure 2.6 shows two methods for the `Make-2Pile` task in the BLOCKS-WORLD domain. The first method states that when the block that should form the base of the tower is on the table and clear and the block that should form the top of the tower is held by the robotic arm, the tower may be completed by stacking the top on the base. The second method states that when the base is on the table and clear, the top is on some block and clear, and the robotic arm is empty, the tower may be completed by unstacking the top block from its current position, then making a tower in which the top block is on the base block. In this case the second

subtask is nonprimitive, so it will need to be further reduced by another method. It should be clear that executing the first subtask of the second method from a state in which the preconditions of the second method hold should result in a state where the preconditions of the first method hold.

Definition 22. A method instantiation $m = (m^h, m^\phi, m^w)$ is **applicable** to a state s and task network $w = \langle t_0, t_1, \ldots, t_n \rangle$ if $m^\phi \subseteq s$ and $m^h = t_0$. That is, a method instantiation is applicable if its preconditions hold in the current state and its head matches the first task in the current task network. The **result** of applying m to s and w is the unchanged state s and a new task network $w' = m^w \cdot \langle t_1, \ldots, t_n \rangle$. Intuitively, the initial task in the original task network is replaced by the subtasks of the method. This is also called a **reduction** of task t_0. We say that a method is applicable to a state and task network if some instantiation of that method is applicable.

Definition 23. An action $a = (a^h, a^\phi, a^-, a^+)$ is **applicable** to a state s and task network $w = \langle t_0, t_1, \ldots, t_n \rangle$ if $a^\phi \subseteq s$ and $a^h = t_0$. That is, an action is applicable if its preconditions hold in the current state and its head matches the first task in the current task network. The **result** of applying a to s and w is a new state $s' = (s \setminus a^-) \cup a^+$ and new task network $w' = \langle t_1, \ldots, t_n \rangle$. Intuitively, the action is applied to the state as in classical planning and the first task in the task network is removed. We say that an operator is applicable to a state and task network if some instantiation of it into an action is applicable.

Definition 24. Given a state s, task network w, set of operators O, and set of methods M, a **decomposition** of w is a plan π that can be generated by recursive reductions of tasks in w and their subtasks using methods in M. A directed graph in which nodes are tasks and edges are task-subtask relationships is a **decomposition tree**. A set of decomposition trees, one for each task in a task network, is a **decomposition forest**.

Specifically, $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ is a decomposition of $w = \langle t_0, t_1, \ldots, t_m \rangle$ from s if one of the following is true:

- Both π and w are of length 0.
- Task t_0 is the head of action a_0, action a_0 is applicable to state s, and the plan $\langle a_1, \ldots, a_n \rangle$ is a decomposition of task network $\langle t_1, \ldots, t_m \rangle$ from state $\gamma(s, a_0)$.
- There exists a method instantiation $m = (m^h, m^\phi, m^w)$ such that m^h is task t_0, $m^\phi \subseteq s$, and π is a decomposition of the task network $m^w \cdot \langle t_1, \ldots, t_m \rangle$ from state s.

For example, consider the state of Figure 2.3a, the initial task network $w = \langle$ `(Make-2Pile C B)` $\rangle$, and the plan $\pi = \langle$ `(!Unstack C A)`, `(!Stack C B)` $\rangle$. (When writing plans, we will include only the heads of the actions, since they can be used to look up the other details.) Neither the plan nor the task network are of length 0, and the first task in the task network is not the head of the first action in the plan. However, there does exist an instantiation of the method from Figure 2.6b with substitution {`?above/C`, `?below/B`, `?other/A`} such that its head is the first task in the task network, and its preconditions are applicable in the state. Therefore, π will be a valid decomposition of w from the state of Figure 2.3a if it is a valid decomposition of the task network $w' = \langle$ `(!Unstack C A)`, `(Make-2Pile C B)` $\rangle$ from that same state.

Neither π nor this new task network w' are of length 0, but the first task in the task network now does match the head of the first action in the plan. Furthermore, that action is applicable to the state of Figure 2.3a. Thus, π will be a decomposition of w' from the state of Figure 2.3a if the subplan $\pi' = \langle$ `(!Stack C B)` $\rangle$ is a decomposition of the third task network $w'' = \langle$ `(Make-2Pile C B)` $\rangle$ from the state of Figure 2.3b.

Neither π' nor w'' are of length 0, and the first task in w'' is not the head of the first action in π'. However, there does exist an instantiation of the method from Figure 2.6a with substitution { `?above/C`, `?below/B` } such that its head is the first task in w'' and it is applicable to the state of Figure 2.3b. Therefore, π' will be a valid decomposition of w'' from the state of Figure 2.3b if it is a valid decomposition of a fourth task network $w''' = \langle$ `(!Stack C B)` $\rangle$ from that same state.

Neither π' nor w''' are of length 0, but the first task in w''' does match the head of the first action in π'. Furthermore, that action is applicable to the state of Figure 2.3b. Thus, π' will be a decomposition of w''' from the state of Figure 2.3b if the subplan $\pi'' = \langle\rangle$ is a decomposition of a fifth task network $w'''' = \langle\rangle$ from the state of Figure 2.3c. Because π'' and w'''' are both of length 0, it is. Following this recursive chain of causation, we have demonstrated that π is a valid decomposition of w from the state of Figure 2.3a. Figure 2.7 shows the associated decomposition tree.

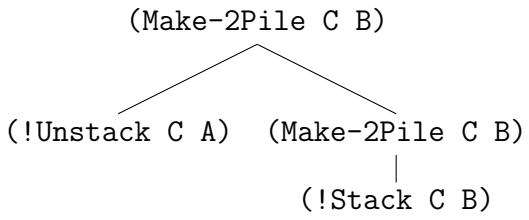

Figure 2.7: An example decomposition tree

Definition 25. An **HTN planning domain** is a 5-tuple $\Sigma[h] = (C, P, O, T, M)$, where C is a finite set of constants, P is a finite set of predicates, O is a finite set of operators, T is a finite set of task templates, and M is a set of methods. Each of the atoms in one of the operators from O or methods from M must correspond to one of the predicates in P. Each of the primitive task templates in T must correspond to the head of an operator in O. The heads of each method in M must correspond to a nonprimitive task template in T.

Definition 26. An **HTN planning problem** is a triple $\Psi[h] = (\Sigma[h], s_0, w_0)$, where $\Sigma[h] = (C, P, O, T, M)$ is an HTN planning domain, s_0 is the **initial state**, and w_0 is the **initial task network**. Each of the atoms in s_0 must be constructed from a predicate in P and constants in C. Each of the tasks in w_0 must be constructed from a task template in T and constants in C.

Definition 27. Given an HTN planning problem $\Psi[h] = (\Sigma[h], s_0, w_0)$, and a plan $\pi = \langle a_0, a_1, \ldots, a_n\rangle$, π is a **solution** to $\Psi[h]$ if π is a decomposition of w_0 from s_0.

Unlike a classical planning domain, there is not a one-to-one correspondence between a generic planning domain and an HTN planning domain, nor between a generic planning problem and an HTN planning problem. In fact, there exist problems that can be modelled as an HTN planning problem but not as a generic planning problem or classical planning problem. Specifically, the set of solutions to a classical planning problem correspond to a regular language, while the set of solutions to an HTN planning problem correspond to a context-free language. However, this work defines a relationship between a certain subset of HTN planning problems and equivalent classical planning problems. For more details, see Section 3.1.

2.2.2 HTN Planning Systems

The ideas that became HTN planning have the same antecedents as plan-space planning. In the same way that steps in plans might be related through ordering and binding constraints and causal links, a step might be related to a set of other steps of which it is an abstraction. The ABStrips system [79] extended the representation of Strips to include abstract versions of operators that have some preconditions removed. After solving a problem using these abstract operators, ABStrips would refine this plan so that it became a solution with the complete operators. For maximum generality, the system could work through a series of progressively more concrete abstraction spaces until it was operating in the complete domain.

In Noah [78], this hierarchy of abstraction spaces was enhanced with the principle of least commitment, which means that temporal orderings among steps were specified only when necessary. Furthermore, this system introduced a language called Soup that allowed the user to provide domain-specific knowledge that described procedurally how to accomplish goals. Thus, it was one of the first domain-configurable planners. As a result of these enhancements, Noah was able to find an optimal solution to the Sussman anomaly that could not be found by ABStrips.

Nonlin [85] is a further refinement of these ideas that uses a task formalism to describe not only preconditions and effects of actions, but ways in which abstract actions may be expanded into lower-level actions or subtasks. These expansion

rules have essentially the same form as the HTN methods described in this document, except that there is no explicit distinction between primitive and nonprimitive tasks and a partial ordering may be defined over subtasks. In addition, expansion rules may specify non-temporal relationships between subtasks, similar to the causal links in plan-space planning. Like SOUP rules, task formalisms are entirely domain-specific. Because NONLIN makes nondeterministic choices in circumstances where NOAH commits to one selection, it is able to solve a greater number of problems.

The SIPE-2 system [90] combines the concept of solving problems at one level of abstraction and then refining that solution with support for reasoning about resources and constraints, and has been used in many "real-world" applications. O-PLAN [9] is similarly based on NONLIN and is designed to allow the use of a variety of search control heuristics.

Although many of these proto-HTN systems were developed in the 1970s, formal definitions of HTN planning were not developed until more than a decade later. Yang formulated a number of rules under which what he called action reduction schemas should be designed to improve efficiency of planners by early recognition of parts of the search space that cannot possibly be completed [96]. The PRIAR system [35] introduced a framework for validating the correctness of nonlinear, hierarchical plans generated as refinements of other existing plans.

The first HTN planner using something like the definitions in Section 2.2.1 was UMCP [12], whose authors also developed the first complete formal analysis of HTN planning. Based on this framework, the authors were able to prove that UMCP is both sound (it never produces an incorrect solution for an HTN planning problem) and complete (if an HTN planning problem has a solution, it will find a solution). The same authors later demonstrated that the problem of determining whether or not an HTN planning problem is solvable using their formalism (in which task networks allow partial orderings and other types of constraints) is undecidable. However, if they restricted task networks to be totally ordered, as in this work, the problem is in the EXPTIME complexity class [13].

SHOP [63] was developed partly as a response to these complexity results. SHOP, which stands for Simple Hierarchical Ordered Planner, uses a forward-chaining

state-space search and the Ordered Task Decomposition formalism described in Section 2.2.1. Because tasks are totally ordered, SHOP avoids the problem of searching for unwanted interactions between tasks as in UMCP, and because SHOP has an explicit representation of the state at any point in a plan, it can support complex representations such as Horn clauses, numeric computations, and even arbitrary external functions. SHOP2 [62] is an extension in which partially ordered subtasks are reintroduced and various additional rich representations are supported.

Algorithm 2 contains pseudocode of a simple forward-chaining, state-space HTN planner similar to SHOP. FCSSH-PLAN begins by checking whether or not the initial task network is empty (Line 3), and if so returns the empty plan as a solution (Line 4). Otherwise, the initial task network must contain at least one task, which is either primitive or nonprimitive. If that task is primitive (Line 5), then the algorithm searches for an action that matches the task and is applicable in the initial state (Line 6). There may be at most one such action. The selected action is then prepended to a solution to a new problem that consists of the resulting state and task network from which that task has been removed (Line 7 - 8). If there is no action matching a first task that is primitive, or no solution can be found that begins with such an action, the algorithm reports that the plan is unsolvable. If the first task is nonprimitive (Line 10), then the algorithm selects a method instantiation that matches the task and is applicable in the initial state (Line 11 - 12) and attempts to solve the problem with the same state and the nonprimitive task replaced by the subtasks of the method instantiation (Line 13). As before, if there does not exist a method instantiation from which this problem can be solved, the algorithm reports failure. Like FCSSC-PLAN, this algorithm follows directly from the definitions, and it should be clear that it is sound and complete.

Algorithm 2: A forward-chaining state-space HTN planner

```
1  Procedure FcSsH-Plan(Σ[h], s_0, w_0)
2  begin
3      if w_0 = ⟨⟩ then
4          return ⟨⟩
5      if w_0 = ⟨t_0, t_1, ..., t_n⟩ and t_0 is primitive then
6          if ∃ an o ∈ O and u such that u(o^φ) ⊆ s_0 and u(o^h) = t_0 then
7              Compute new state s' ← (s_0 \ u(o^-)) ∪ u(o^+)
8              return ⟨u(o)⟩· FcSsH-Plan(Σ[h], s', ⟨t_1, ..., t_n⟩)
9          return FAIL
10     else
11         if ∃ an m ∈ M and u such that u(m^φ) ⊆ s_0 and u(m^h) = t_0 then
12             Nondeterministically select such an m and u
13             return FcSsH-Plan(Σ[h], s_0, u(m^w) · ⟨t_1, ..., t_n⟩)
14         return FAIL
15 end
```

2.3 Planning In Nondeterministic Domains

One of the fundamental assumptions of classical (and most HTN) planning is that the domains are **deterministic**, which means that the result of executing an action in a state are known in advance. While this assumption makes planning much simpler than it would otherwise be, it is generally broken in any "real-world" application. Even in the simple Blocks-World domain, if the planner were controlling a physical robotic arm we would need a way to sense whether or not the robot successfully executed its instructions. When trying to place block `C` on block `B`, improper calibration might cause block `C` to fall to the table, of the arm might carelessly knock over another pile of blocks, or have other unexpected consequences.

2.3.1 Definitions For Planning In Nondeterministic Domains

Domains in which actions will have one of several possible outcomes, to be determined at the time of execution, are **nondeterministic**. Note that it is still necessary for the possible outcomes of an action to be enumerated, although in the

worst case one could assume that the result of applying an action to a state is any member of the (finite) set of states in the domain. In such a domain, a plan is not a sufficient solution to a problem because there is no guarantee that executing the first n actions in the plan will result in a state where the $n+1$th action is applicable. In conditional planning, the solution to a planning problem in a nondeterministic domain is a **policy** specifying what action to take for each reachable state. An agent using this policy would be able to sense the state after executing each action and then take the following action specified by the policy until it reaches a state in which the goals hold. There are other ways of handling nondeterminism that require different sorts of solutions, such as conformant planning, that are outside the scope of this document. The following definitions formalize these ideas.

Definition 28. An **nondeterministic action** is a triple $a = (a^h, a^\phi, a^E)$. The head and preconditions of a nondeterministic action are defined in the same way as the head and preconditions of a deterministic action. Rather than a single set of negative effects and a single set of positive effects, a nondeterministic action has a finite, non-empty set a^E of pairs, each of which contains a set of negative effects and a set of positive effects. During execution, exactly one of the pairs in a^E will be used to transform the current state into the successor state.

Nondeterministic operators and the applicability of nondeterministic actions and operators are defined analogously to their deterministic counterparts. A **nondeterministic planning domain** $\Sigma[c] = (C, P, O)$ has the same form as a classical planning domain but has nondeterministic operators, and a **nondeterministic planning problem** $\Psi[c] = (\Sigma[c], s_0, g)$ is one that occurs in a nondeterministic planning domain [1].

Definition 29. A **policy** is a partial function [2] $\Pi : S \rightarrow A$, where S is a finite set of states and A is a finite set of nondeterministic actions.

[1] I continue to use the suffix $[c]$ for nondeterministic planning domains and problems because they have the same form as classical planning domains and problems and to produce a visual distinction between nondeterministic planning domains that use HTNs and those that do not. They are not, however, classical.

[2] It is not necessary for a policy to be defined over all states.

Definition 30. Given a finite set of states S, a finite set of nondeterministic actions A, and a policy Π over those states and actions, the **execution structure** of Π is a directed graph $G_\Pi = (V, E)$, where V is the set of states S and E is the set of pairs (s, s') such that $\Pi(s) = a$ and s' is a potential outcome of applying a to s. If there is a path in G_Π from state s to state s', then s is a Π**-ancestor** of s' and s' is a Π**-descendant** of s. If there is an edge (a path of length 1) from s to s', the s is a Π**-parent** of s' and s' is a Π**-child** of s.

There are several different types of solutions for a nondeterministic planning problem [8]. These are based not on plans, but on policies, because a plan is inflexible and cannot respond to different possible outcomes of its constituent actions. A weak solution is the least interesting, guaranteeing only that with fortuitous circumstances it is possible to reach a goal state by following a policy. A strong-cyclic solution is more interesting, guaranteeing that no matter the circumstances, given infinite time following the policy will reach a goal state. A strong solution guarantees that in all circumstances following the policy will lead to a goal state in a finite number of iterations. The following definitions formalize these notions.

Definition 31. Given a policy Π and a nondeterministic planning problem $\Psi[c] = (\Sigma[c], s_0, g)$, Π is a **weak solution** to $\Psi[c]$ if and only if there exists a state s' such that $g \subseteq s'$ and s' is a Π-descendant of s_0 in G_Π.

Definition 32. Given a policy Π and a nondeterministic planning problem $\Psi[c] = (\Sigma[c], s_0, g)$, Π is a **strong-cyclic solution** to $\Psi[c]$ if and only if for each state s that is reachable by following Π from s_0 there exists a state s' such that $g \subseteq s'$ and s' is a Π-descendant of s in G_Π, and for each state s that is reachable by following Π from s_0 either $\Pi(s)$ is defined (s has a Π-descendant) or $g \subseteq s$ (s satisfies the goals).

Definition 33. Given a policy Π and a nondeterministic planning problem $\Psi[c] = (\Sigma[c], s_0, g)$, Π is a **strong solution** to $\Psi[c]$ if and only if Π is a strong-cyclic solution to $\Psi[c]$ and there are no cycles in G_Π.

In order to demonstrate how these definitions may be used in practice, I have included three algorithms based on them. Algorithm 3 shows a simple controller

Algorithm 3: A controller that follows a policy

```
1  Procedure ExecutePolicy(Σ[c], s_0, g, Π)
2  begin
3      s ← s_0
4      while true do
5          if g ⊆ s then
6              return SUCCESS
7          if Π(s) is undefined then
8              return FAIL
9          s ← apply Π(s) to s
10 end
```

that executes a policy for a nondeterministic planning problem. If the controller has reached a state that satisfies the goals (Line 5), then it has successfully achieved the goals. If the controller has reached a state for which the policy does not specify a next action (Line 7), then it has failed and is unable to continue. If the policy is a strong or strong-cyclic solution to the problem, then this will never occur. Unlike the planning stage, it is impossible to backtrack during execution time. Otherwise, the controller executes the action specified by the policy (Line 9) and observes which potential effects actually occurred, then continues.

Algorithm 4 shows a simplistic but sound and complete algorithm for finding weak solutions to nondeterministic planning problems. It begins with an empty policy (Line 3), and successively adds a rule for each state, chaining backward from the goals, until the policy contains a rule for the initial state (Line 4). Possible rules that could be added to the policy are those that can lead to a goal state (Line 5) and those that can lead to a state for which a rule has already been added (Line 6). In either case, adding this rule will cause there to be a path in the execution structure from the newly added state to a goal state. If there are no such rules to add (Line 7), then the algorithm must backtrack to different choices and, if this is impossible, report that the problem has no solution. Otherwise, any of the rules may be chosen (Line 9), and the process continues.

Algorithm 4: A backward-chaining algorithm for finding weak solutions to nondeterministic planning problems

```
1  Procedure SolveWeak(Σ[c], s_0, g)
2  begin
3      Initialize Π to be undefined for all states
4      while Π is undefined for s_0 do
5          Y = {(s, a)}|Π is undefined for s and applying a to s can result in a
           state s' where g ⊆ s'
6          Y = Y ∪ {(s, a)}|Π is undefined for s and applying a to s can result in
           a state s' where Π is defined for s'
7          if Y contains no pairs then
8              return FAIL
9          Nondeterministically select (s, a) from Y and add to Π
10     return Π
11 end
```

Algorithm 5 is a basic algorithm for finding strong-cyclic solutions to nondeterministic planning problems, using a forward-chaining search strategy. It begins by initializing an empty policy Π, a "fringe" set S of states that must still be explored, and a "solved" set S_g of states that satisfy the goals (Lines 3 - 5). It then loops while there remain states in the fringe (Line 6).

Within each iteration of this loop, it nondeterministically selects one of the states in the fringe and removes it (Lines 7 - 8). If this state satisfies the goals (Line 9), then it does not need a rule in the policy, and should instead be added to the set of solved states. Otherwise, if there is no rule in the policy for it (Line 11), then a rule must be added to process it. If no actions are applicable to this state (Line 13), then no policy that can reach this state will be a solution, and the algorithm must backtrack. Otherwise, an applicable action is chosen nondeterministically (Line 15) and a rule added to the policy for this state-action pair (Line 16). All states that are Π-children of s in the execution structure are added to the fringe (Line 17), even those that already have rules or are solved. If the selected state already has a rule in the policy, then it must have a Π-descendant that does not already have a rule, or the algorithm will need to backtrack and select a different state at the current

Algorithm 5: A forward-chaining algorithm for finding strong-cyclic solutions to nondeterministic planning problems

```
1  Procedure SolveCyclic(Σ[c], s_0, g)
2  begin
3      Initialize Π to be undefined for all states
4      S = {s_0}
5      S_g = ∅
6      while S ≠ ∅ do
7          Nondeterministically select s ∈ S
8          S = S \ {s}
9          if g ⊆ s then
10             S_g = S_g ∪ {s}
11         else if Π is not defined for s then
12             A = {a}|a is applicable to s
13             if A = ∅ then
14                 return FAIL
15             Nondeterministically select a ∈ A
16             Add a rule to Π that in state s action a should be taken
17             Add to S all states that could result from applying a to s
18         else
19             if s has no Π-descendants in (S ∪ S_g) for which Π is undefined
               then
20                 return FAIL
21     return Π
22 end
```

iteration (Line 19).

Algorithms 4 and 5 should not be interpreted to imply that backward-chaining is best suited to finding weak solutions or forward-chaining to finding strong-cyclic solutions; they are merely examples of different problem-solving strategies. Finding strong solutions is significantly more complicated, and for many problems is impossible.

2.3.2 Systems That Plan In Nondeterministic Domains

Perhaps the best-known planner for nondeterministic planning domains is MBP, which uses symbolic model checking to model the planning problem [8]. Model checking is a broadly-applicable technique in which a problem is represented as a finite-state machine and that machine is analyzed to determine whether or not it models a temporal logic formula. When applied to planning in nondeterministic domains, this formula would be a statement that there exists a solution of a particular type to a planning problem. In symbolic model checking states, actions, and policies are all represented as such temporal logic formulas, often in a compact representation for reasoning, such as a binary decision diagram. MBP is capable of producing weak, strong-cyclic, and strong solutions (or proving that no such solution exists), based on the selection of one of several internal algorithms.

Alternatively, an existing planning algorithm that uses heuristics or domain-dependent knowledge to guide a forward-chaining, state-space planner can be adapted to work in nondeterministic domains by replacing the arbitrary choices made at Lines 7 and 15 in Algorithm 5 with decisions guided by planning-graph heuristics, temporal logic formulas, or other techniques [44]. Of particular interest to us is ND-SHOP2, which uses task decomposition to constrain the actions selected. In ND-SHOP2, the fringe and solved sets of Algorithm 5 are of pairs containing a state and task network. Such a pair is considered solved if the task network is empty, regardless of the contents of the state. When the first task in the task network of the selected pair is primitive, there are potentially many successor states, each paired with the the remainder of the task network. When the first task in the task

network of the selected pair is nonprimitive, there is exactly one successor pair per applicable method, which consists of the same state and the modified task network.

In some domains, the search control provided by task reduction in ND-SHOP2 is very effective, but not all. Similarly, some domains can be expressed very compactly in the BDD-based representation of MBP, and thus solved very quickly. The YOYO planner combines these two ideas, and thus works well in domains where either of these conditions holds, and exceptionally in domains where both are true [45]. Yet another idea is to partition a nondeterministic planning problem into several deterministic planning problems, use existing classical planners to find solutions to each of those sub-problems, and then combine the results into a strong-cyclic solution to the original nondeterministic planning problem, as in NDP [46].

2.4 Reinforcement Learning

Reinforcement learning is a closely related field to planning in nondeterministic domains. The essential difference is that in reinforcement learning there are no explicitly stated goals, nor is there necessarily a symbolic structure specifying the states and actions. Rather, there is an *agent* that interacts with the world by observing its current state and taking an applicable action. After each action that it takes, this agent receives a numerical reward. The objective of the reinforcement learning problem is to discover a policy that will maximize the rewards that the agent receives.

2.4.1 Definitions For Reinforcement Learning

Reinforcement learning is a synthesis of behavioral psychology, control theory, statistics, and artificial intelligence, and is thus resistant to universal, formal definitions. Nevertheless, there are some common components to all reinforcement learning systems [83].

Definition 34. Each reinforcement learning problem consists of an **environment** and an **agent** that acts within that environment.

The environment contains a set S of **states**, which describe a certain situation. At any given time, one and only one state is the **current state**. States do not need to have any particular internal structure, but there must be a way for the agent to determine what is the current state.

The environment also contains a set A of **actions**, each of which is applicable in some (possibly improper) subset of states. Actions do not need to have any particular internal structure, but there must be a way for the agent to determine what actions are applicable in the current state. The environment contains a **state-transition function** $\gamma : S \times A \to S$, which specifies the resulting state when an action is taken from a given current state. This function is usually probabilistic, and is often unknown to the agent.

Additionally, the environment contains a **reward function** $R : S \times A \to \mathbb{R}$, which maps a state-action pair to a real number. This function may be deterministic or probabilistic, and is unknown to the agent.

Definition 35. The environment and agent interact in the following manner:

The agent senses the current state and selects an applicable action. The environment determines the new current state and the reward that the agent should receive for taking this action. The agent senses the new current state and the reward that it received as a result of its most recent choice. The agent chooses an action applicable to the new current state, the environment determines a successor state and reward, and so forth.

If this process continues indefinitely, this is a **continuous** reinforcement learning problem. Otherwise, it is **episodic**.

Definition 36. A **policy** $\Pi : S \to A$ is a function mapping each state to an action that is applicable to that state. When an agent makes decisions by following a policy, it is **exploiting** existing knowledge; when it instead seeks to discover a better policy it is **exploring**.

Definition 37. A policy Π is **optimal** for a reinforcement learning problem if there does not exist another policy Π' such that an agent following Π' would receive a greater sum of rewards than an agent following Π.

2.4.2 Reinforcement Learning Algorithms

One class of algorithms for solving reinforcement learning problems is *dynamic programming*, which does not actually require trial-and-error exploration [30]. Dynamic programming uses a probabilistic policy, which means that rather than mapping each state to an action that should be taken in that state, each pair of state and action is mapped to the probability that the action will be taken from within that state. The initial probabilities in this policy may be assigned arbitrarily, as long as the probability is 0 for any state-action pair where the action is not applicable in the state, and that the sum of probabilities across all state-action pairs containing the same state is 1.

In addition to a probabilistic policy, dynamic programming requires an estimate of the *value* of each state, which represents the expected sum of future rewards that the agent would receive if it found itself in that state and followed the current policy. These state value estimates may also be assigned initial values arbitrarily. The value estimates may then be iteratively updated, by replacing the value of each state with the sum of the expected reward from following the policy in that state and the values of the successor states that could be reached by following the policy from that state. When this iterative process reaches a fixed point, the estimates have converged to the true values of the states, given the current policy.

Based on accurate estimates of state values, it may be possible to improve the existing policy. Specifically, a new policy may be created that chooses actions in each state that lead to states with the highest value. Such a policy is either strictly better than the one from which the state values have been computed, or both are optimal. State values when following this new policy will not be the same as when following the old policy, and so they must again be iteratively improved until they again reach a fixed point. Based on those new state values, the policy may again be improved, and this process may continue until an optimal policy is formulated.

Unfortunately, dynamic programming is a very computationally expensive process, since each iteration requires re-calculating a value for each state at least once, and there are typically very many states and very many iterations required before

resolving to an optimal policy. To a certain extent, this can be mitigated by clever programming and stopping iterative processes before they reach a fixed point. A more significant problem with dynamic programming is that it requires that the agent have a great deal of information about the environment. In order to perform these calculations, the agent needs to have a complete knowledge of the system dynamics (given a state and action that is taken in that state, what are the possible successor states, and what are the probabilities of each occurring) and moderately comprehensive knowledge of the reward signal (given a state and an action that is taken in that state, what is expected value of the reward). In practice, this information is often not available to the agent.

When this advanced knowledge is unknown, the agent must actually try things and observe the rewards that it receives and states that it reaches. Techniques based on analyzing the results of attempts to solve episodic reinforcement learning problems are known as *Monte Carlo methods* [80]. Within an episode, each state is reached some finite number of times n, and for each action that is applicable to that state, the action is chosen some finite number of times $k < n$. For each of those action selections the agent receives a certain **return**, which is the sum of the immediate reward and all (potentially discounted) rewards received thereafter. The value of taking an action in a state is simply the average of the returns seen when doing so. As with dynamic programming, after good estimates of the value of state-action pairs have been found, the current policy may be improved by selecting those actions that have the highest values for each state.

In order to guarantee that all actions are selected often enough to get a reasonable approximation of their values, agents using Monte Carlo techniques typically do not follow their current policies exactly. Instead, each time a decision must be made it either chooses randomly with probability ϵ or chooses the action specified by the policy with probability $1 - \epsilon$, where ϵ is a relatively small value. This is an ϵ-greedy policy, because it usually favors exploitation of existing knowledge to acquire the highest possible returns in the near-term.

A third category of solution strategies is *temporal-difference learning* [82]. TD-learning techniques utilize experience rather than complete knowledge, just as Monte

Carlo techniques do. However, they base the value of a state-action pair directly on the values of potential successor state-action pairs, like dynamic programming techniques.

Specifically, a generic TD-learning algorithm might begin by initializing a value for each state-action pair arbitrarily. Then it will begin an episode and select an action to take in the initial state (perhaps through an ϵ-greedy policy based on the existing state-action values). Each time that it reaches a successor state and receives a reward, it updates the value of the state-action pair that led to it, based on the values of the state-action pairs that are now available and the reward that it received during the transition.

There are algorithms based on each category of solution strategies that are guaranteed to converge to an optimal policy, though some do so much more quickly than others.

Chapter 3

Learning HTN Methods

HTN planning is used more than any other planning technique [91], and there are many reasons to prefer HTN planning over classical planning. Most notably, with appropriate domain knowledge SHOP can solve problems orders of magnitude more quickly than even the fastest classical planner [61]. This means that an HTN planner can solve much larger and more complex problems in a reasonable amount of time than can a classical planner. Moreover, there exist some types of problems that cannot be expressed in the classical planning formalism, but can be expressed and solved in HTN planning [13].

However, the knowledge engineering involved in the creation of methods forms a significant barrier to more widespread adoption of HTN planning. Because every solution must follow from a decomposition using the domain-specific methods, it is necessary that these methods entail at least one solution to every problem in a given domain. Creating a set of methods and verifying their correctness is a time-consuming challenge requiring both an intimate understanding of the domain to be modeled and expertise in the HTN formalism.

This dissertation describes a process for automating the creation of the methods of an HTN planning domain. The general idea is to consider only tasks that have clear semantics relative to the atoms in the domain and analyze example plans to find situations in which such a task has been accomplished. From such a situation, the system will learn one or more methods describing exactly how the task was

accomplished and the conditions under which it would be valid to accomplish similar tasks the same way in other problems.

3.1 Definitions

In order to determine when and how a task has been accomplished, it is necessary to know what it means to accomplish that task. Although this was natural in some proto-HTN planning systems, tasks in the SHOP formalism have no formal semantics. Rather, what it means to accomplish a task is defined by which methods can decompose it in which circumstances. If the methods are unknown, a task is simply a meaningless symbol. In this work, I associate semantic meanings to the tasks for which methods should be learned as *annotated tasks*. Annotated tasks are similar to tasks as they appear in the Task-Method-Knowledge Language (TMKL) formalism for process models [60].

Definition 38. An **annotated task** is a triple $\tau = (\tau^h, \tau^\phi, \tau^+)$, in which the **head** τ^h is a task and the **preconditions** τ^ϕ and **positive effects** τ^+ are finite set of atoms whose arguments are all arguments of the head.

The preconditions of an annotated task specify those conditions that must be true before that task can be attempted, while the positive effects specify the essential results of a successful accomplishment of the task. Unlike an action, these positive effects do not completely specify how accomplishing the task will alter the state. There are likely many different ways to accomplish a given annotated task, each of which will have their own *side effects*. The commonality between these different ways to accomplish the annotated task is that all of them at a minimum cause the positive effects of the annotated task to become true. Annotated tasks are not allowed to have negative effects because classical planning does not allow negative goals or preconditions. There do exist other, more complicated formalisms in which this is legal, such as the Action Description Language [70], but it is possible to simulate such structures in this formalism by adding new atoms that are true only when their counterparts are false.

```
( :annotated-task
  :head
  (Make-2Pile ?above ?below)
  :precondition
  {}
  :positive-effects
  { (on-table ?below),
    (on ?above ?below),
    (clear ?above) }
)
```

(a) `Make-2Pile`

```
( :annotated-task
  :head
  (Invert-2Pile ?o-ab ?o-be)
  :precondition
  { (on-table ?o-be),
    (on ?o-ab ?o-be),
    (clear ?o-ab) }
  :positive-effects
  { (on-table ?o-ab),
    (on ?o-be ?o-ab),
    (clear ?o-be) }
)
```

(b) `Invert-2Pile`

Figure 3.1: Two example annotated tasks from the BLOCKS-WORLD domain

Figure 3.1 shows two annotated tasks from the BLOCKS-WORLD domain. The first formalizes what is meant by creating a pile of two blocks. Namely, a pile of two blocks has been created when the block to become the bottom is on the table, the block to become the top is on the block to become the bottom, and the block to become the top is clear. This annotated task has no preconditions because given the existence of the two blocks in question and the operators of the BLOCKS-WORLD domain it is always possible to create a pile of the blocks. The two example methods shown in Figure 2.6 each accomplish this annotated task. The correctness of the first method should be apparent: if you begin from a state in which `(on-table ?below)` is true and execute an action that causes `(on ?above ?below)` and `(clear ?above)` to be true without causing `(on-table ?below)` to become false, then each of those three atoms will be true in the succeeding state. The correctness of the second method depends on what other methods exist for the `Make-2Pile` task; if the method used to decompose the final subtask of this method is correct, then this method too will be correct.

The second annotated task in Figure 3.1 is quite similar to the first. In fact, the positive effects of the two annotated tasks are identical. What makes the second

annotated task interesting is that it can only be performed when the two blocks begin in a pile that has the reverse order of what is desired. That is, the block that is supposed to be on top of the pile is on the bottom and vice versa. This precondition is essential to the understanding of inversion; it is not meaningful to talk about inverting a pile that does not exist.

The examples of Figure 3.1 contain a formal description of what the reader intuitively understands the symbols `Make-2Pile` and `Invert-2Pile` to mean. This is not true merely of these examples, but of the process of defining annotated tasks in general. To state the types of tasks we might like to accomplish and what we mean by those tasks is quite straightforward. On the other hand, formally defining all possible ways to accomplish a task is quite complicated. Thus, I argue that the amount of knowledge engineering required to produce a set of annotated tasks for a domain is trivial compared to that required to design a set of methods for that domain.

Given the semantics of annotated tasks, it is possible to define an equivalence between a task and a set of goals. Based on that equivalence, it is further possible to define an equivalence between a classical planning problem and an HTN planning problem. Note that while all classical planning problems have an equivalent HTN planning problem, the reverse is not true. An HTN planning problem only has a classical equivalent if its initial task network consists of a single task that has been annotated with no preconditions.

Definition 39. The **equivalent annotated task** to a set of goal atoms g is $(\tau^h, \emptyset, g)$, where τ^h is an arbitrary nonprimitive task symbol that uniquely represents this set of goals.

Definition 40. The **HTN-equivalent planning problem** to a classical planning problem $(\Sigma[c], s_0, g)$ is an HTN planning problem $(\Sigma[h], s_0, w_0)$ where $\Sigma[c] = (C, P, O)$, $\Sigma[h] = (C, P, O, T, M)$, T contains the primitive tasks corresponding to O and at least one annotated task τ that is the equivalent of g, M is some set of correct methods for the nonprimitive tasks in T, and $w_0 = \langle \tau \rangle$.

Definition 41. A **learning example** is a pair $e = (s_0, \pi)$, where s_0 is a state and π is a plan applicable to that state.

The objective of this work is to capture knowledge from learning examples that describes how tasks might be accomplished. Specifically, given a classical planning domain, a finite set of learning examples, and a finite set of annotated tasks, the system should learn methods that, combined with the components of the classical planning domain and the annotated tasks, will form a useful HTN planning domain. The next section describes an algorithm for solving this problem.

3.2 The HTN-Maker Algorithm

My algorithm for the problem of learning HTN methods from examples is called Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required (HTN-Maker). The HTN-Maker algorithm analyzes plans to determine when and how tasks are accomplished and produces methods that can be used to accomplish those tasks in similar situations. An additional data structure is required for bookkeeping; an *indexed method instance* describes how a certain subplan accomplishes a task.

Definition 42. An **indexed method instance** is a 6-tuple $(x^h, x^+, x^w, x^\phi, x^b, x^e)$, where x^h is the head of an annotated task, x^+ is the positive effects of that annotated task, x^w is a task network into which that annotated task may be reduced, x^ϕ is a finite set of preconditions under which that reduction is valid, and x^b and x^e are nonnegative integers representing indices of the beginning and ending of a sub-state trajectory.

An indexed method instance is created when HTN-Maker learns a method to accomplish the annotated task given by x^h from a subplan that begins directly after the state indexed by x^b and ends directly before the state indexed by x^e. The preconditions of the new method, the subtasks of the new method, and the postconditions of the annotated task are stored in x^ϕ, x^w, and x^+, respectively.

Algorithm 6: A high-level description of the HTN-MAKER procedure. The input includes a classical planning domain description $\Sigma[c]$, a finite set of learning examples E, a finite set of annotated tasks $\mathcal{T}$, and a (possibly empty) finite set of methods M. The output is an updated set M of HTN methods.

```
1  Procedure HTN-MAKER(Σ[c], E, T, M)
2  begin
3      foreach learning example e = (s_0, π) ∈ E do
4          initialize X ← ∅
5          initialize S⃗ ← ⟨s_0⟩
6          for i ← 1 to k do
7              s_i ← γ(s_{i−1}, a_{i−1})
8              S⃗ ← S⃗ · ⟨s_i⟩
9          for f ← 1 to k do
10             for i ← f − 1 down to 0 do
11                 foreach annotated task τ = (τ^h, τ^φ, τ^+) ∈ T do
12                     if τ^φ ⊆ s_i and τ^+ ⊆ s_f then
13                         m ← LEARN-METHOD(π, S⃗, τ, X, i, f)
14                         M ← M ∪ {m}
15                         X ← X ∪ {(m^h, τ^+, m^w, m^φ, i, f)}
16     return M
17 end
```

This data structure will be used to determine whether or not the annotated task of x^h may be used as a subtask in other methods that are learned later from the same learning example.

Algorithm 6 shows a high-level pseudocode for the HTN-MAKER algorithm. The algorithm iterates through each of the learning examples independently (Line 3). For each learning example, it begins by initializing an empty set of indexed method instances (Line 4) and a state trajectory consisting of the initial state from the example (Line 5). The rest of the state trajectory is generated by applying each of the actions in the plan from the example in turn (Lines 6 - 8).

HTN-MAKER then considers each non-empty, contiguous subplan (Lines 9 - 10). Specifically, f is an index pointing to the state that immediately follows the subplan in question, while i is an index pointing to the state that immediately precedes it.

For each subplan, HTN-Maker further considers every possible annotated task. If the preconditions of that task are satisfied in the state that precedes the subplan and the positive effects of that task are satisfied in the state that follows the subplan, then the subplan accomplishes that task (Line 12). In that case, HTN-Maker calls an auxiliary procedure Learn-Method to generate a method describing how that task was accomplished (Line 13), adds that new method to the set of known methods (Line 14), and stores an indexed method instance describing how the task was accomplished (Line 15). That indexed method instance will be used in future calls to Learn-Method, which is described in Algorithm 7. After processing each learning example, HTN-Maker returns the expanded set of methods as its output.

The order in which subplans are considered is quite deliberate, and follows the pattern $\langle\langle a_0\rangle, \langle a_1\rangle, \langle a_0, a_1\rangle, \langle a_2\rangle, \langle a_1, a_2\rangle, \langle a_0, a_1, a_2\rangle \ldots, \pi\rangle$. Notably, whenever a part of the plan is being considered all of its contiguous subplans have already been processed. This is important because it means that whatever knowledge could be learned from those subplans is already known and stored in the set of indexed method instances. The Learn-Method procedure will thus be able to use these indexed method instances to produce complex hierarchies of tasks.

3.2.1 The Learn-Method Algorithm

I now describe my *hierarchical goal regression* technique, which is the basis of my procedure for learning the appropriate preconditions and subtasks for an individual method. Unlike traditional goal regression, 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).

Traditional goal regression [56] works horizontally on actions similarly to the backward-chaining state space algorithm for classical planning presented in Section 2.1.2. Supposing that the system has a set of goals g, it reasons that action a can assist in achieving those goals if any member of g is also a member of a^+. In this case, the system can formulate a plan to achieve g if it appends a to a plan that achieves the preconditions of a and any other goals in g. Thus, the system continues

with a new set of goals g' that contains the members of a^ϕ and any members of g that are not also members of a^+. We say that goal g has been *regressed* through action a. Given a set of goals g and a plan π, we can find the set $g' = \mathcal{R}(g, \pi)$ with a regression operator similar to the one defined in [75], but simplified due to the use of a set-theoretic formulation:

- If π is the empty plan, then $\mathcal{R}(g, \pi) = g$.
- If π contains a single action $a = (a^h, a^\phi, a^-, a^+)$ and $a^+ \cap g \neq \emptyset$, then $\mathcal{R}(g, \pi) = (g \setminus a^+) \cup a^\phi$.
- If π contains a single action $a = (a^h, a^\phi, a^-, a^+)$ and $a^+ \cap g = \emptyset$, then $\mathcal{R}(g, \pi) = g$.
- If π contains two or more actions $\langle a_0, a_1, \ldots, a_n \rangle$, then $\mathcal{R}(g, \pi) = \mathcal{R}(\mathcal{R}(g, \langle a_n \rangle), \langle a_0, a_1, \ldots, a_{n-1} \rangle)$.

In hierarchical goal regression, a set of goals 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. Each indexed method instance $(x^h, x^+, x^w, x^\phi, x^b, x^e)$ contains a description of how certain atoms (those in x^+) can be made true through decomposition of a task (x^h) in states where x^ϕ are true. In this case, the regression is performed over the postconditions of the annotated task and the preconditions of the method learned for that task. Given a set of goals g and a task network w, we can find the set $g' = \mathcal{R}(g, w)$ with the following extension of the regression operator:

- If w is the empty task network, then $\mathcal{R}(g, w) = g$.
- If w contains a single primitive task t that corresponds to the action $a = (t, a^\phi, a^-, a^+)$ and $a^+ \cap\ g \neq \emptyset$, then $\mathcal{R}(g, w) = (g \setminus a^+) \cup a^\phi$.
- If w contains a single primitive task t that corresponds to the action $a = (t, a^\phi, a^-, a^+)$ and $a^+ \cap g = \emptyset$, then $\mathcal{R}(g, w) = g$.

- If w contains a single nonprimitive task t for which there exists an indexed method instance $x = (t, x^+, x^w, x^\phi, x^b, x^e)$ and $x^+ \cap\ g \neq \emptyset$, then $\mathcal{R}(g, w) = (g \setminus x^+) \cup x^\phi$.

- If w contains a single nonprimitive task t for which there exists an indexed method instance $x = (t, x^+, x^w, x^\phi, x^b, x^e)$ and $x^+ \cap\ g = \emptyset$, then $\mathcal{R}(g, w) = g$.

- If w contains two or more tasks $\langle t_0, t_1, \ldots, t_n \rangle$, then $\mathcal{R}(g, w) = \mathcal{R}(\mathcal{R}(g, \langle t_n \rangle), \langle t_0, t_1, \ldots, t_{n-1} \rangle)$.

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.

The LEARN-METHOD subroutine uses this regression operator to determine a sequence of tasks that accomplishes an annotated task and the conditions under which it will do so, which become the subtasks and preconditions of a new method. It is not quite a straightforward implementation of the regression operator because the task network through which goals should be regressed is not known in advance and must be created.

Algorithm 7 shows a high-level pseudocode for the LEARN-METHOD procedure. The input to LEARN-METHOD consists of a plan, a state trajectory induced by that plan, an annotated task, a finite set of method instances indexed into that state trajectory, and initial and final indices specifying a subplan in which the annotated task is accomplished. The algorithm begins by initializing a set of open conditions ϕ as the positive effects of the annotated task (Line 3). These are the goals that will be regressed through the actions of the subplan and relevant indexed method instances. It additionally initializes an empty task network w and current action index c (Lines 4 - 5). The task network w will eventually become the subtasks of the method learned in this call to LEARN-METHOD.

The current state index begins at the very end of the subplan and moves backwards to the initial state of the subplan (Line 6). Within this loop, the algorithm initializes an empty set of *potentially useful indexed method instances* (Line 7). An

Algorithm 7: The LEARN-METHOD procedure that performs hierarchical goal regression over HTNs. The inputs are a plan π, a state trajectory $\vec{S}_\pi$, an annotated task τ, a finite set of indexed method instances X, and indices for an initial state i and final state f. The output is a new method m.

```
1  Procedure LEARN-METHOD(π, S⃗_π, τ, X, i, f)
2  begin
3      φ ← τ⁺
4      w ← ⟨⟩
5      c ← f
6      while c > i do
7          X′ ← ∅
8          foreach x = (x^h, x⁺, x^w, w^φ, x^b, x^e) ∈ X do
9              if c = x^e ∧ i ≤ x^b ∧ x⁺ ∩ φ ≠ ∅ then
10                 X′ ← X′ ∪ {x}
11         a_c ← the c-th action in π
12         if a_c⁺ ∩ φ ≠ ∅ then
13             X′ ← X′ ∪ {(a_c^h, a_c⁺, ⟨⟩, a_c^φ, c − 1, c)}
14         X′ ← X′ ∪ {(nop, ∅, ⟨⟩, ∅, c − 1, c)}
15         nondeterministically select x = (x^h, x⁺, x^w, x^φ, x^b, x^e) ∈ X′
16         φ ← (φ \ x⁺) ∪ x^φ
17         if x^h ≠ nop then
18             w ← ⟨x^h⟩ · w
19         c ← x^b
20     m = (τ^h, φ ∪ τ^φ, w)
21     return m
22 end
```

indexed method instance is deemed potentially useful if it fulfills three criteria: the current state is the end state of the method instance, the initial state is no later than the begin state of the initial state, and there exists an atom in both the open conditions and the positive effects of the method instance (Lines 8 - 10). The first two criteria ensure that this indexed method instance applies to the subplan currently being considered, while the last ensures that this indexed method instance helped to accomplishing the open conditions. Furthermore, the action that resulted in the current state (a_c) is potentially useful if there exists an atom in both the open conditions and the positive effects of the action (Lines 12 - 13). (In this presentation

of the algorithm, I create a structure that might be called an indexed action instance to store information the action. This information is not actually stored with the indexed method instances, but it is convenient to present it as such.) Finally, doing nothing, represented as "nop", is not useful (in that it does not contribute to the open conditions), but may also be selected (Line 14).

The algorithm makes a nondeterministic choice between one of the potentially useful indexed method instances (if any exist), the action (if it was useful), and the "nop" (Line 15). The open conditions are regressed through the selection by removing any open condition that is also a positive effect of the indexed method instance or action and adding the preconditions of the indexed method instance or action (Line 16). If an indexed method instance or the action was selected, its head is prepended to the task network (Line 18). The current state index is moved back to the last state before the subplan covered by the indexed method instance (or to the previous state if the action or "nop" was selected) (Line 19), and the loop continues with the new value of c.

When the entire plan has been processed, a new method is created (Line 20). The head of this new method is the annotated task that was accomplished. The subtasks are the collected heads of the indexed method instances and actions through which the positive effects of the annotated task have been regressed, and the preconditions are the remaining open conditions and any preconditions of the annotated task itself.

The action of the HTN-Maker and Learn-Method algorithms is best demonstrated through an example, which is shown in the following section.

3.3 Example

Consider the following state in the Blocks-World domain: blocks `C` and `B` are on the table, block `A` is on block `C`, and there are no other blocks. That state, combined with the plan `<(!Unstack A C), (!Stack A B), (!Pickup C), (!Stack C A)>` forms a learning example. In addition to the `Make-2Pile` and `Invert-2Pile` annotated tasks from Figure 3.1, suppose that there exist analogous tasks for piles of 1

and 3 blocks. We will demonstrate the HTN-MAKER algorithm with the classical planning domain for BLOCKS-WORLD, a set containing the aforementioned learning example, a set containing these annotated tasks, and an empty set of methods as its input.

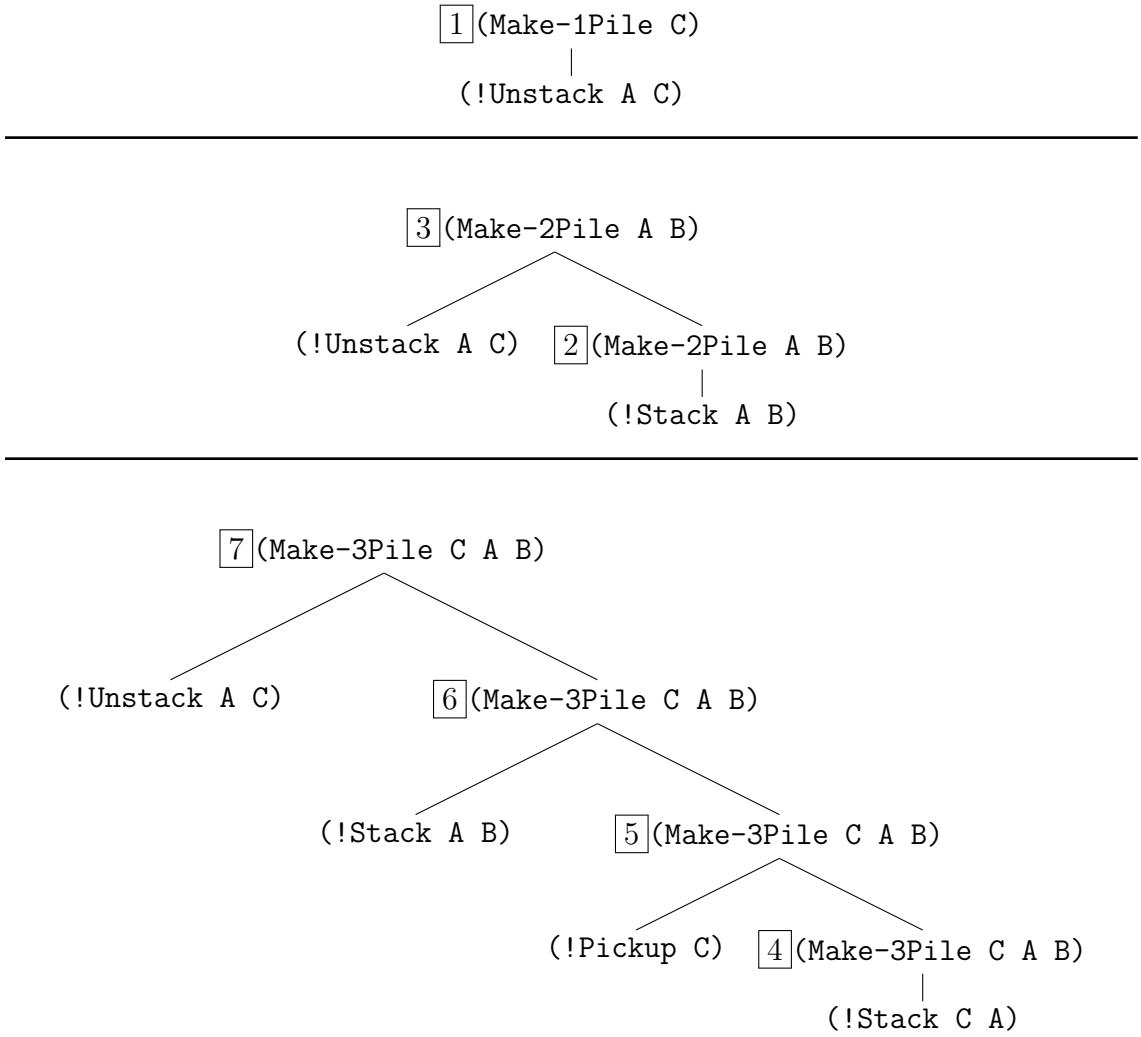

Figure 3.2: Example of decomposition trees obtained by HTN-MAKER

Figure 3.2 shows one particular structure of methods that HTN-MAKER might learn from this input, depending on what nondeterministic choices are made. The first subplan that HTN-MAKER would consider is ⟨ (!Unstack A C) ⟩. This subplan does not result in the creation or inversion of any piles of size 2 or 3, but it does

create a pile of size 1. Namely, block `C` becomes a pile of size 1 because it is on the table and no longer has anything above it. Thus, the LEARN-METHOD procedure will be called for this subplan and the `Make-1Pile` annotated task. Initially the open conditions are `(on-table C)` and `(clear C)`. There are initially no indexed method instances, but the action `(!Unstack A C)` does provide one of the open conditions, `(clear C)`. After regression through this action, the new open conditions become `(on-table C)`, `(on A C)`, `(clear A)`, and `(hand-empty)`. That is the end of this subplan, so LEARN-METHOD produces a method that has `(!Unstack A C)` as its only subtask and the open conditions listed above as its preconditions. This is marked as method 1 in the figure. HTN-MAKER would add this new method to its list of methods and create an indexed method instance describing how that subplan accomplished the task. The first segment of Figure 3.2 shows this simple hierarchy.

Next, HTN-MAKER would consider the subplan ⟨ `(!Stack A B)` ⟩. In this case, the `Make-2Pile` annotated task has been accomplished. The initial open conditions would be `(on-table B)`, `(on A B)`, and `(clear A)`. The only existing indexed method instance does not fit within this subplan. (It began in state 0 and ended in state 1, while the system is now considering the subplan from state 1 to state 2.) However, the action `(!Stack A B)` is potentially useful. It produces both `(on A B)` and `(clear A)`. After regression through this action, the open conditions would be `(on-table B)`, `(clear B)`, and `(holding A)`. Thus, method 2 has these as its preconditions and `(!Stack A B)` as its only subtask.

The next loop is for subplan ⟨ `(!Unstack A C)`, `(!Stack A B)` ⟩ (that is, state 0 to state 2). Once again, the `Make-2Pile` annotated task has been accomplished, and the initial open conditions are `(on-table B)`, `(on A B)`, and `(clear A)`. Now there is an indexed method instance that ends at the current state index and begins no earlier than the initial state index: the one associated with method 2 that was just learned. The positive effects associated with this indexed method instance are the positive effects of the annotated task that it explains, which also happens to be the annotated task for which a method is currently being learned. As a result, this indexed method instance provides all of the initial open conditions. The new open condition list is the same as the preconditions of method 2: `(on-table B)`,

`(clear B)`, and `(holding A)`. Instead of this indexed method instance, the algorithm could have nondeterministically chosen the action `(!Stack A B)` or a "nop", but for the purposes of this example, suppose that it did not. With the current state index decremented, the indexed method instance associated with method 1 that was learned from the 0-1 subplan covers the proper portion of the plan. However, it is not potentially useful because its only positive effects are `(on-table C)` and `(clear C)`, neither of which is an open condition. Instead, the algorithm selects the action `(!Unstack A C)`, which is potentially useful. After regressing through this action, the open condition list is `(on-table B)`, `(clear B)`, `(on A C)`, `(clear A)`, and `(hand-empty)`. Thus, method 3 has these preconditions and the subtasks ⟨ `(!Unstack A C)`, `(Make-2Pile A B)` ⟩. This hierarchy is shown in the second segment of Figure 3.2.

This example followed a particular set of nondeterministic choices in the creation of method 3, and several others could be made as well. In fact, the system could create a method for the `Make-2Pile` annotated task from this subplan with any of the following subtask lists: ⟨⟩, ⟨ `(Make-2Pile A B)` ⟩, ⟨ `(!Stack A B)` ⟩, ⟨ `(!Unstack A C)`, `(Make-2Pile A B)` ⟩, and ⟨ `(!Unstack A C)`, `(!Stack A B)` ⟩, with varying preconditions.

Analysis of further subplans yields methods 4, 5, 6, and 7 in the figure, or potentially many others depending on nondeterministic choices. These seven methods can be used to solve many problems where the elements of the initial task network are construction of piles of size 1, 2, or 3, but not all. For example, these methods do not encode the knowledge necessary to create a pile of size 2 when both blocks are initially on the table.

3.4 Implementation Details

The generic algorithm described in Section 3.2 is quite flexible, and must be made significantly less abstract before it can be used as an executable program. Even the examples of Section 3.3 required making several choices. This section discusses

how the HTN-Maker algorithm was specialized into the program used in the experimental evaluation.

3.4.1 Removing Nondeterminism

The most obvious missing component is a mechanism for making the nondeterministic choice at Line 15 in Algorithm 7. Early versions of HTN-Maker tried all possible choices, and thus learned not a single method from each call to Learn-Method, but rather quite a few. This approach worked, but resulting in learning what were essentially many ways of describing the same problem-solving strategy. Brevity in an HTN domain description is valuable both for computer programs that must use the set of methods and for humans who wish to read and understand them, so this redundancy was deemed inappropriate.

Another possibility would be to make a single, arbitrary selection and never backtrack. This would result in learning only one way to express a problem-solving strategy, but perhaps a different way for each such strategy. In addition to making it difficult to understand how the learned methods are supposed to work, this may select suboptimal or even useless ways of expressing a particular strategy.

Consider again the plan and annotated tasks used in Section 3.3. There were five different ways in which the first two actions could be grouped into a task hierarchy, which would result in the five different methods of Figure 3.3.

When Learn-Method is called for this subplan (that is, i is 0 and f is 2), at its first nondeterministic choice point it will have three options: the indexed method instance that was learned from the 1-2 subplan, the action `(!Stack A B)`, and "nop". Suppose that it selects "nop". In that case, at its second nondeterministic choice point it will have only one option: "nop" again. Making these two selections produces the method shown in Figure 3.3a. This is a *trivial method* as described in Section 3.4.2, and it does provide a useful function. Trivial methods like these allow the planner to mark a task as accomplished when the postconditions of that task are already true. However, it is not necessary to learn these methods from traces, and doing so in this case ignores some useful information about how to accomplish

```
( :method
  :head
  (Make-2Pile ?a ?b)
  :precondition
  { (on-table ?b),
    (on ?a ?b),
    (clear ?a) }
  :subtasks
  <>
)
```

(a) Method A

```
( :method
  :head
  (Make-2Pile ?a ?b)
  :precondition
  { (on-table ?b),
    (clear ?b),
    (holding ?a) }
  :subtasks
  < (Make-2Pile ?a ?b) >
)
```

(b) Method B

```
( :method
  :head
  (Make-2Pile ?a ?b)
  :precondition
  { (on-table ?b),
    (clear ?b),
    (holding ?a) }
  :subtasks
  < (!Stack ?a ?b) >
)
```

(c) Method C

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

(d) Method D

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

(e) Method E

Figure 3.3: Several methods resulting from different choices

this task when its postconditions are not already true. Thus, we would prefer not to learn this particular method from this trace.

Suppose instead that LEARN-METHOD chooses the method that was learned from the indexed method instance from the 1-2 subplan. Then it will have two options when it reaches the nondeterministic choice point for the second time: the action `(!Unstack A C)` and "nop". Suppose that it selects "nop". Making these two selections produces the method shown in Figure 3.3b. At first glance this appears to be a rather useless method, allowing a task to be reduced into another copy of itself. In fact, it is worse than useless. After using this method, neither the current state nor the current task list will have changed, which means it is still applicable. Thus, methods of this form can easily lead an HTN planner into infinite recursion if the planner does not implement some sort of loop checking.

Suppose that LEARN-METHOD chooses the method that was learned from the indexed method instances from the 1-2 subplan at its first choice point and the action `(!Unstack A C)` at its second choice point. Making these two selections produces the method shown in Figure 3.3d, and this is in fact the decision that was made while following the example in Section 3.3. This method, combined with the method that was learned from the 1-2 subplan, is a very reasonable way to express the strategy demonstrated in the 0-2 subplan.

Suppose that LEARN-METHOD chooses the action `(!Stack A B)` at its first choice point. Then at its second choice point it will have two options: the action `(!Unstack A C)` and "nop". Suppose that it selects "nop". These two selections lead to the method shown in Figure 3.3c. Unfortunately, this method is identical to the one that was learned from the 1-2 subplan, and the information about how to handle situations where the block `?a` is not held has not been captured from the trace. Thus, the system should avoid learning this method.

Finally, if LEARN-METHOD chooses the action `(!Stack A B)` at its first choice point and `(!Unstack A C)` at its second, it will learn the method shown in Figure 3.3e. This captures all of the knowledge that we are interested in, but does so in a way that is not very interesting. Rather than building a hierarchy of methods that can be used together and potentially with methods learned from other traces,

this type of choice causes HTN-Maker to learn entire, concrete plans. These can either be re-used in their entirety or not at all.

In summary, the method of Figure 3.3d is a good choice, the method of Figure 3.3e is acceptable but less interesting, and other options are either not needed or actively harmful. To learn methods of the type shown in Figure 3.3d and avoid learning all of the other types, the implementation of the HTN-Maker algorithm follows the following rules:

1. If at least one potentially useful indexed method instance exists, choose one of them. Among these, choose the one that "covers" the longest subplan. That is, the one with lowest x^b value. If several are tied for longest, choose one arbitrarily and do not backtrack.

2. If there are no potentially useful indexed method instances and the current operator is potentially useful, select it.

3. Only select "nop" when neither of the above is true.

Following these rules, HTN-Maker will learn the method of Figure 3.3d and make the choices followed in Section 3.3.

Why is the "nop" choice needed at all? It allows HTN-Maker to learn correctly from traces that contain actions that are legal but do not contribute to achieving the task that HTN-Maker is currently analyzing. A long trace in a complex domain will have many such actions for each call to Learn-Method.

3.4.2 Trivial Methods

In Section 3.4.1, we referred to the method of Figure 3.3a as a "trivial method". This name derives from the fact that these methods are used when a task can be accomplished trivially, by the empty plan. Scenarios in which such a task appears in an HTN planning problem are not uncommon for three reasons. First, most planning problem generators do not take care to ensure that the goals of a problem are not true in the initial state, and if they did the space of expressible problems

would decrease. Second, in a complex problem it is likely that accomplishment of some tasks will have the serendipitous side effect of accomplishing other, unrelated tasks in the network. Finally, the recursive nature of methods means that a "base case" is necessary, although that base case is usually a method that reduces to a single primitive subtask rather than the empty task network.

Because trivial methods are important and do not require a demonstration from which to be learned, HTN-Maker begins by searching the domain to see if they exist before processing any learning examples. If they do not, it creates one for each annotated task in a straightforward process. The head of the trivial method is the same as the head of the annotated task. The preconditions of the trivial method are the union of the preconditions and positive effects of the annotated task. The subtasks of the trivial method are the empty task network.

3.4.3 Verification Tasks

As will be discussed in Section 3.5.1, there are situations in which the generic HTN-Maker algorithm can learn methods that can be used to perform reductions that do not result in the positive effects of their annotated tasks being true. One way to avoid this issue is to add additional infrastructure into the domain to verify that a task has been (and remains) accomplished. The idea is that the final task in each method that is learned will be a special one that can only be reduced by a trivial method. Because it is not always needed, support for this is provided as a configuration option of HTN-Maker.

Definition 43. A **verification task** is simply a (non-annotated) nonprimitive task. There must be one unique verification task associated with each annotated task used in a domain.

Definition 44. Given an annotated task $\tau = (\tau^h, \tau^\phi, \tau^+)$ and its associated verification task t', the **verification method** for τ has the following form: $m = (t', \tau^\phi \cup \tau^+, \langle\rangle)$.

When the use of verification tasks is enabled, HTN-Maker begins by creating

a verification task for each annotated task (through simple name-mangling) that does not already have one. It then creates a verification method for each annotated task that does not already have one. This verification method is used to reduce the verification task into the empty task network in a state in which both the preconditions and positive effects of the annotated task hold. Every time LEARN-METHOD creates a new method, it inserts the verification task associated with the head at the end of its subtasks. Thus, any sound algorithm for solving HTN planning problems will also automatically enforce the relationship between tasks and goals.

3.4.4 Subsumption

I have previously mentioned the advantage of brevity in an HTN planning domain description. Thus, the system should avoid adding methods that do not increase the number of problems that could be solved or represent a new way to solve a problem. A first, straightforward step to accomplish this is to avoid adding a method if an equivalent one already exists.

Definition 45. A method $m_1 = (m_1^h, m_1^\phi, m_1^w)$ is **equivalent** to another method $m_2 = (m_2^h, m_2^\phi, m_2^w)$ if and only if there exists substitutions u and u' such that $m_1^h = u(m_2^h)$, $m_1^\phi = u(m_2^\phi)$, $m_1^w = u(m_2^w)$, $m_2^h = u'(m_1^h)$, $m_2^\phi = u'(m_1^\phi)$, and $m_2^w = u'(m_1^w)$.

However, preventing the addition of duplicate methods is not in itself sufficient. The system should also avoid adding a method m' if it can guarantee that in every situation in which it would be applicable to a state and task network there already exists another method m that will be applicable and that will have the same results. We say that method m subsumes method m' if this is the case.

Definition 46. A method $m_1 = (m_1^h, m_1^\phi, m_1^w)$ **subsumes** another method $m_2 = (m_2^h, m_2^\phi, m_2^w)$ if and only if there exists a substitution u such that $m_1^h = u(m_2^h)$, $m_1^\phi \subseteq u(m_2^\phi)$, and $m_1^w = u(m_2^w)$.

HTN-MAKER has an option to check for and remove subsumed methods. When this option is active, after each call to the LEARN-METHOD subroutine it tests each

existing method to see whether or not it subsumes the newly learned one. If any do, the newly learned method is discarded. Otherwise, it then tests to see if any existing methods are subsumed by the newly learned one before adding it. If any do, they are removed. When this option is inactive, HTN-Maker instead tests every newly learned method for strict equivalence with all existing methods and discards it only if it is equivalent to an existing method. The algorithms for checking subsumption and equivalence require standardizing apart the variables used in the two methods and then exhaustively searching for a substitution that meets the requirements of the definition.

There are no simple examples in the Blocks-World planning domain where one method would subsume another but not be equivalent to it, so we will introduce a second common domain here, Logistics [88]. The Logistics domain models the problem of delivering packages from one location to another. Objects in the world include packages, cities, locations within cities (some of which are airports), trucks, and airplanes. Packages may be loaded into a truck or airplane if the package and vehicle are in the same location, and may be unloaded from a truck or airplane to the current location of the vehicle. Trucks may travel between any two locations within the same city, while airplanes may travel between any two locations that are airports. The basic annotated task in this domain is to deliver a specific package to a specific location.

Unlike the Blocks-World domain, Logistics includes a variety of types of objects that can be interacted with in different ways. To simplify the description of the domain, we use a simple, non-hierarchical type system in which the type of a constant or variable is given when that term is declared.

Figure 3.4a shows the basic annotated task for the Logistics domain, and the rest of Figure 3.4 shows three methods that could be learned to accomplish that task. The method of Figure 3.4b is more general than the method of Figure 3.4c, because it does not require that the location `?b` be an airport. Thus, we expect that the method of Figure 3.4b should subsume the method of Figure 3.4c. Indeed it does, with substitution {`?d/?a`, `?e/?b`, `?f/?c`}.

```
( :annotated-task
  :head
  (Deliver ?obj - package
    ?dest - location)
  :precondition
  {}
  :positive-effects
  { (pkg-at ?obj ?dest) }
)
```

(a) Annotated Task

```
( :method
  :head
  (Deliver ?a - package
    ?b - location)
  :vars
  { ?c - truck }
  :precondition
  { (truck-at ?c ?b),
    (in-truck ?a ?c) }
  :subtasks
  < (UnloadTruck ?a ?c ?b) >
)
```

(b) Method A

```
( :method
  :head
  (Deliver ?d - package
    ?e - location)
  :vars
  { ?f - truck }
  :precondition
  { (truck-at ?f ?e),
    (in-truck ?d ?f),
    (is-airport ?e) }
  :subtasks
  < (UnloadTruck ?d ?f ?e) >
)
```

(c) Method B

```
( :method
  :head
  (Deliver ?g - package
    ?h - location)
  :vars
  { ?i - truck,
    ?j - truck }
  :precondition
  { (truck-at ?i ?h),
    (in-truck ?g ?i),
    (in-truck ?g ?j) }
  :subtasks
  < (UnloadTruck ?g ?j ?h) >
)
```

(d) Method C

Figure 3.4: An annotated task and several methods to achieve it

The relationship between the method of Figure 3.4b and the method of Figure 3.4d is not so obvious. The latter involves two different truck variables, `?i` and `?j`, such that truck `?i` is known to be at location `?h`, package `?g` is known to be in both trucks, and the instruction is to unload package `?g` from truck `?j` at location `?h`. An observant human with an intuitive understanding of the domain physics will recognize that a package may not simultaneously be inside two different trucks, and thus that this method will only be applicable when variables `?i` and `?j` are replaced by the same constant. This means that the two methods are functionally equivalent, even though they are not logically equivalent. However, the method from Figure 3.4b subsumes the method from Figure 3.4d with substitution {`?g/?a`, `?h/?b`, `?i/?c`, `?j/?c`}. There is no subsumption in the opposite direction.

3.4.5 Generalization

The actions in a plan π of a learning example are entirely grounded. The terms in an HTN method, however, are generally variables. When an HTN planning system uses a method, it applies a substitution to it that makes it ground. Thusfar I have glossed over the issue of how HTN-Maker generalizes from constants in a plan to variables suitable for a method. This is both because it would be too complex to introduce with the main algorithm and because there are multiple possible design choices that could be made.

On Line 12 of Algorithm 6, HTN-Maker actually searches for a substitution u^* such that $u^*(\tau^\phi) \subseteq s_i$ and $u^*(\tau^+) \subseteq s_f$, and calls the Learn-Method subroutine for each such substitution that can be found. The substitution u^* that maps variables in the annotated task to constants in the state trajectory is passed to the Learn-Method subroutine as an additional parameter. When Learn-Method returns on Line 13 of Algorithm 6 it returns both the new method m and an updated substitution u^*, which is stored as part of the indexed method instance.

On Line 8 of Algorithm 7, each indexed method instance contains an additional component named x^u, which is a substitution that maps each variable that appears in any of x^h, x^+, x^w, and x^ϕ to constants from the state trajectory.

On Line 9 of Algorithm 7, LEARN-METHOD searches for a substitution u such that $c = x^e$, $i \leq x^b$, and $u(x^u((x^+)) \cap \phi \neq \emptyset$. The substitution u maps variables in the indexed method instance to variables in the outstanding precondition list, which could have come from the positive effects of the annotated task or from preconditions of previously added subtasks. For each such substitution, the pair (x, u) is actually added to the list of potentially useful indexed method instances. Line 12 similarly searches for an appropriate substitution u.

Once a (x, u) pair has been selected, there are at least two ways in which the variables in x^u may be integrated with those already mapped in u^*. I refer to the possibility that produces more general methods as weak generalization and the possibility that produces more specific methods as strong generalization.

In **weak generalization**, variables in x^u are unified with variables in u^* only if they appear in u as well. That is, these variables are only unified if doing so is necessary to represent the fact that this indexed method instance or action is fulfilling an outstanding precondition. For example, if `(on ?a ?b)` is part of the open condition list and u^* includes {`?a/B12`, `?b/B3`} and `(on ?x ?y)` is a positive effect of the indexed method instance or action such that x^u includes {`?x/B12`, `?y/B3`}, then u will include {`?x/?a`, `?y/?b`}, and so `?x` will be unified with `?a` and `?y` will be unified with `?b`.

Otherwise, variables used in the indexed method instance or action become new variables in u^*. For example, suppose that u^* also contains {`?c/B5`, `?d/B7`, `?e/B1`}, and that none of these variables appear in u. Even if x^u contains {`?z/B7`}, `?z` will not be unified with `?d`. Thus, when an HTN planner uses the method that is being built, it will be free to map `?z` and `?d` to the same constant or to different constants, as makes sense for the current state and task network.

In **strong generalization**, on the other hand, all variables in x^u will be unified with variables in u^* as long as each are mapped to the same constant. Furthermore, additional preconditions are added that will prevent an HTN planner from mapping two variables of the same type to the same constant. (Variables of different types could not be mapped to the same constant regardless of this.) In the example above, `?x` would be unified with `?a`, `?y` would be unified with `?b`, and `?z` would be unified

with ?d. Preconditions would be added to ensure that ?x, ?y, ?z, ?c, and ?e all refer to distinct objects at all times.

HTN-MAKER includes a switch that allows either weak or strong generalization to be selected. Methods learned using strong generalization will be applicable in the same or fewer situations than methods learned using weak generalization. However, this may be a good thing as methods could be applicable even when they are not useful, and such overly-applicable methods would slow down a planner.

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

(a) Weak Generalization

```
( :method
  :head
  (Deliver ?p - package
    ?y - location)
  :vars
  { ?x - location,
    ?t - truck }
  :precondition
  { (package-at ?p ?x),
    (truck-at ?t ?y),
    (not (= ?x ?y)) }
  :subtasks
  < (!Drive-Truck ?t ?y ?x),
    (Deliver ?p ?y) >
)
```

(b) Strong Generalization

Figure 3.5: 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 3.5, which are for the LOGISTICS domain that was first introduced in Section 3.4.4. 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 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. In this case it is not relevant that the starting location of the truck and the destination of the package happened to be the same in the example from which the method was learned, and so the weak generalization version is preferable.

3.5 Properties

In this section we will discuss the soundness and completeness properties of the HTN-Maker algorithm, the computational complexity of the algorithm, and the types of problems that can be expressed using methods learned by HTN-Maker.

3.5.1 Soundness

The notion of soundness for this work is related to the equivalence between a goal and an annotated task, and through that the equivalence between a classical planning problem and an HTN planning problem. Specifically, the annotations on a task should be treated as a contract, with a requirement that all methods learned for an annotated task are guaranteed to abide by that contract (causing the positive effects to become true). Informally, the learning algorithm is sound if it is impossible to use the methods it learns to completely decompose an annotated task without causing the positive effects of that annotated task to become true in the resulting state.

Definition 47. An algorithm for learning HTN methods using annotated tasks is **sound** if and only if, for every classical planning problem in the domain, every solution produced by a sound HTN planning procedure using the learned methods for the HTN-equivalent to that classical planning problem is also a solution to that classical planning problem.

Surprisingly, the generic HTN-Maker procedure described in Section 3.2 is not sound. Given poorly-designed annotated tasks, it can learn methods for a task that

can be used in a way that does not cause the positive effects of that task to become true. This possibility exists because in a series of nonprimitive tasks, a particular valid reduction of a later nonprimitive task may negate a required positive effect of an earlier nonprimitive task. Consider a concrete example illustrating this situation.

```
( :annotated-task
  :head
  (Make-2Pile ?x ?y)
  :precondition
  {}
  :positive-effects
  { (on ?x ?y) }
)
```

(a) Annotated Task A

```
( :annotated-task
  :head
  (Make-3Pile ?x ?y ?z)
  :precondition
  {}
  :positive-effects
  { (on ?x ?y),
    (on ?y ?z) }
)
```

(b) Annotated Task B

Figure 3.6: Alternate annotated tasks for the BLOCKS-WORLD domain

```
< (!Stack B C), (!Pickup A),
  (!Stack A B) >
```

(a) Plan A

```
< (!Unstack B C), (!Putdown B),
  (!Pickup A), (!Stack A B) >
```

(b) Plan B

Figure 3.7: Two plans in the BLOCKS-WORLD domain

Consider the annotated tasks shown in Figure 3.6, 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 3.1 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 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 perfectly valid.

Suppose that the plan shown in Figure 3.7a is executed from the following initial state: {`(on-table A)`, `(on-table C)`, `(clear A)`, `(clear C)`, `(holding B)`}. In

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

(a) Method A

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

(b) Method B

Figure 3.8: Two methods that could be learned from the plans of Figure 3.7 and annotated tasks of Figure 3.6

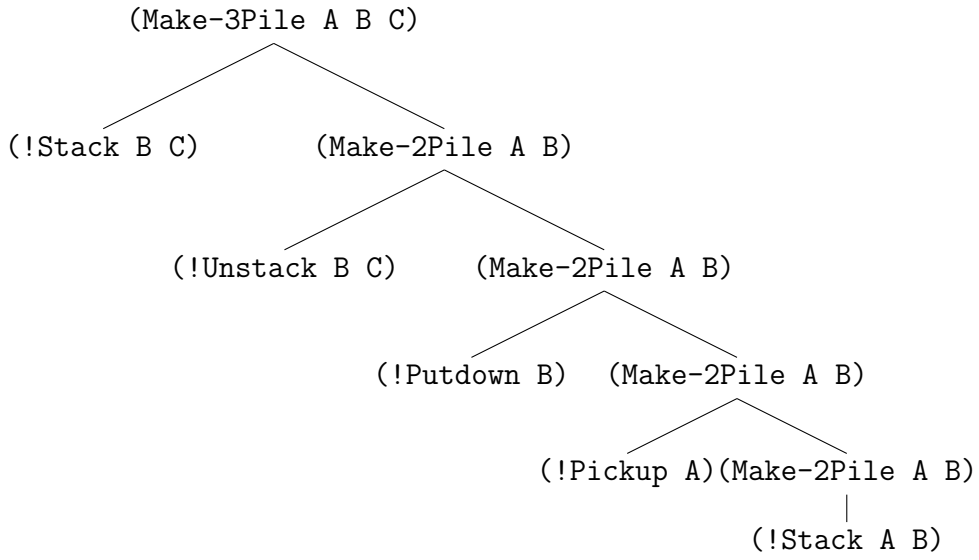

Figure 3.9: An example decomposition tree including the methods of Figure 3.8

addition to many others, HTN-MAKER could learn the method shown in Figure 3.8a from this learning example. Also suppose that the plan shown in Figure 3.7b is executed from the following initial state: {`(on-table A)`, `(on-table C)`, `(on B C)`, `(clear A)`, `(clear B)`, `(hand-empty)`}. In addition to many others, HTN-MAKER could learn the method shown in Figure 3.8b from this learning example.

The method shown in Figure 3.8a seems logical: the first subtask puts the bottom part of the pile together, then the second subtask completes the top part of the pile. Unfortunately, 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, the method shown in Figure 3.8b will do exactly that. Figure 3.9 shows a complete decomposition in which both of these methods are used. Executing the leaves of this tree from the state {`(on-table A)`, `(on-table C)`, `(clear A)`, `(clear C)`, `(holding B)`} does not result in a state where `(on B C)` is true. Thus, this plan is not sound relative to the annotations on the `Make-3Pile` task. While not identical, this issue is closely related to the Sussman anomaly discussed in Section 2.1.2.

Fortunately, there are two relatively straightforward modifications to the HTN-MAKER algorithm, either of which will make it sound. Section 3.4.3 discussed verification tasks and verification methods. I will shortly demonstrate that the inclusion of verification tasks and methods is sufficient to guarantee the soundness of the algorithm. The other idea is to restrict the nondeterministic choices in HTN-MAKER to a greater degree than is discussed in Section 3.4.1. Before providing more detail, I state and prove several useful lemmas.

Lemma 1. *Let $\Sigma[c]$ 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$(\Sigma[c], E, \mathcal{T}, \emptyset)$.

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 method m is the trivial method, which is always created if it does not already exist, as described in Section 3.4.2. □

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 $\Psi[c] = (\Sigma[c], s_0, g)$ be a classical planning problem with $\Sigma[c] = (C, P, O)$, and $\Psi[h] = (\Sigma[h], s'_0, w_0)$ be its equivalent HTN planning problem with $\Sigma[h] = (C', P', O', T, M)$ for some tasks T and methods M. Let $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ be a plan produced by a sound HTN planning algorithm as a solution to $\Psi[h]$.*

Then, π is applicable to the initial state of $\Psi[c]$, which is s_0.

Proof. Because $\Psi[h]$ is the HTN equivalent to $\Psi[c]$, $s_0 = s'_0$, $C = C'$, $P = P'$, and $O = O'$. If π is the empty plan, it is trivially applicable to s_0. If π consists of a single action a_0, then that action must be applicable to s'_0, and thus to s_0. If π consists of multiple actions, then the first is applicable to s_0 by the logic above. The result of applying a_0 to s'_0 will be s'_1. The same logic as above requires that a_1 be applicable to s'_1, and likewise for all (a_i, s_i) pairs. Because $s_0 = s'_0$, all of the above follows for the classical planning problem as well. □

Note that the proof of Lemma 2 is not dependent on any properties of the methods in M. Rather, it follows directly from the definition of an HTN equivalent planning problem and a sound HTN planning algorithm.

Theorem 1. *Let $\Sigma[c] = (C, P, O)$ 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$(\Sigma[c], E, \mathcal{T}, \emptyset)$ *with verification tasks and methods enabled. Let $\Psi[c] = (\Sigma[c], s_0, g)$ be a classical planning problem and $\Psi[h] = (\Sigma[h], s'_0, w_0)$ be the equivalent HTN planning problem with $\Sigma[h] = (C', P', O', T, M)$. Let π be a plan produced by a sound HTN planner as a solution to $\Psi[h]$.*

Then, π is a solution to $\Psi[c]$ (and thus, HTN-Maker *with verification tasks and methods enabled is sound).*

Proof. In order to prove that π is a solution to $\Psi[c]$ 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 $\Psi[h]$ is the equivalent HTN planning problem to $\Psi[c]$, $s_0 = s'_0$, $C = C'$, $P = P'$, and $O = O'$. Furthermore, 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, $g \subseteq s'_0$. Since $s'_0 = s_0$ and there are no actions in the plan, s'_0 is the result of applying π to s_0, 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 the HTN planner 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 resulting from the application of π to s_0. □

In some cases it may not be desirable to use verification tasks. An alternative is to restrict the way HTN-Maker forms methods to disallow potentially unsound constructs from being learned. Specifically, restricting HTN-Maker to only learn right-recursive methods will make it sound. The rules already in use tend to produce right-recursive methods in most circumstances, and it is simple to force HTN-Maker to do so in all cases.

Definition 48. An HTN method is **right-recursive** if and only if any of the following is true:

1. The method has no subtasks.
2. The method has one or more subtasks, all of which are primitive.
3. The method has one or more primitive subtasks followed by a nonprimitive subtask that is a recursive call to the head of the method.

Although right-recursion may seem overly restrictive, it is in fact a natural way (though certain not the only way) to model classical planning domains. In experiments in a variety of standard planning domains, including LOGISTICS, BLOCKS-WORLD, SATELLITE, ROVERS, and ZENO, methods of these forms have been sufficient to model the domain effectively.

Lemma 3. *Let $m = (m^h, m^\phi, m^w)$ be a method learned by* HTN-MAKER *from some classical planning domain $\Sigma[c]$, learning example $e = (\pi, s_0)$, and annotated task $\tau = (\tau^h, \tau^\phi, \tau^+)$, such that m^w consists of one or more primitive tasks. Let s be a state in which m is applicable.*

Then the postconditions of the annotated task, τ^+, hold in the state resulting from the application of the subtasks of m to s.

Proof. This follows from straightforward application of (non-hierarchical) goal regression. Each atom $g \in \tau^+$ must be either a positive effect of a primitive task in m^w and not a negative effect of any following task, or a member of m^ϕ. □

Theorem 2. *Let $\Sigma[c] = (C, P, O)$ 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$(\Sigma[c], E, \mathcal{T}, \emptyset)$ *without the use of verification tasks, but with a restriction that all methods learned must be right-recursive.*

Let $\Psi[c] = (\Sigma[c], s_0, g)$ be a classical planning problem and $\Psi[h] = (\Sigma[h], s_0', w_0)$ be the equivalent HTN planning problem with $\Sigma[h] = (C', P', O', T, M)$. Let π be a plan produced by a sound HTN planner as a solution to $\Psi[h]$.

Then, π is a solution to $\Psi[c]$ (and thus, HTN-MAKER *with a restriction to only learn right-recursive methods is sound).*

Proof. As in the proof of Theorem 1, Lemma 2 demonstrates that π will be applicable in the initial state of $\Psi[c]$. The rest of this proof demonstrates that π, when executed in the initial state of $\Psi[c]$, will result in a state in which g holds.

Because $\Psi[h]$ is the equivalent HTN planning problem to $\Psi[c]$, $s_0 = s_0'$, $C = C'$, $P = P'$, and $O = O'$. Furthermore, 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, $g \subseteq s'_0$. Since $s'_0 = s_0$ and there are no actions in the plan, s'_0 is the result of applying w_0 to s_0, and it has been shown that it satisfies the goals.

Otherwise, π was produced by either a single reduction using a method with only primitive subtasks or by a sequence of one or more reductions using tail-recursive methods followed by a final reduction using a method with only primitive subtasks.

In the former case, Lemma 3 shows that the state resulting from application of the subtasks of m satisfies the positive effects of the annotated task, and thus the goals of the classical planning problem.

In the latter case, $\pi = \pi' \cdot \pi''$, where π' consists of the primitive tasks from each of the reductions using tail-recursive methods and π'' is the subtasks of the non-recursive method, which is used in the final reduction. Let s be the state that results from applying π' to s'_0 and s' be the state that results from applying π'' to s. Lemma 3 shows that s' is a state in which g holds. The actions in π' are irrelevant; all that matters is that goal regression guarantees that if a non-recursive method m is applicable in s', then the positive effects of the annotated task associated with m will hold in s''. Because the only nonprimitive tasks allowed are strictly recursive calls, this is the same annotated task τ. □

While either verification tasks or this restriction are necessary for theoretical soundness, the situations in which unsound behavior could result are very rare in real domains. For the experiments reported in Chapter 6 I chose to use verification tasks. As part of the experiments I logged situations in which the planner would attempt to decompose a verification task, fail and backtrack, and did not record a single instance of this occurring.

3.5.2 Completeness

The notion of completeness used in this work for an algorithm that learns methods is based on the ability of an HTN planner to solve all expressible problems using the methods that are learned.

Definition 49. Given a classical planning domain $\Sigma[c]$ and a finite set of annotated tasks for that domain $\mathcal{T}$, a set of methods M is **sufficient** to model the domain and tasks if and only if, for every classical planning problem $\Psi[c] = (\Sigma[c], s_0, g)$ that has goals g matching an annotated task in $\mathcal{T}$, an HTN planner using M will be able to solve the HTN equivalent of $\Psi[c]$.

Definition 50. An algorithm for learning HTN methods from annotated tasks and traces is **complete** if and only if, for every pair of classical planning domain $\Sigma[c]$ and set of annotated tasks $\mathcal{T}$, there exists a finite set of learning examples E from which the algorithm will learn a set M of methods that is sufficient to model the domain and tasks.

Before demonstrating that HTN-Maker is a complete algorithm, I introduce two necessary lemmas.

Lemma 4. *Let $\Sigma[c]$ be a classical planning domain description, $\mathcal{T}$ be a set of annotated tasks for the domain, $\Psi[c] = (\Sigma[c], s_0, g)$ 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 $\Psi[c]$.*

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 to $\Psi[c]$, $\Psi[h] = (\Sigma[h], s_0, \langle\tau\rangle)$ with $\Sigma[h] = (C, P, O, T, M)$.*

Proof. If the length of π is 0, then $g \subseteq s_0$, or π would not be a solution to P. Then the trivial method described in Lemma 1 will be applicable to s_0, producing the empty task network. Thus, $\Psi[h]$ can be solved.

If the length of π is 1, then it consists of an action $a = (a^h, a^\phi, a^-, a^+)$. Thus, HTN-Maker would learn a method $m = (\tau^h, (g \setminus a^+) \cup a^\phi, \langle a^h \rangle)$. Each member of $g \setminus a^+$ must be true in s_0, because otherwise it would not be true in s_1. Each member of a^ϕ must be true in s_0, because otherwise a would not be applicable to s_0. Therefore, m is applicable in s_0. Thus, an HTN planner could reduce τ^h into a^h, and then apply a to s_0, resulting in the empty HTN and a solution to $\Psi[h]$.

Suppose that the length of π is $n > 1$, and that this lemma is true for all plans of length $n - 1$. Then HTN-MAKER will learn a method $m = (\tau^h, m^\phi, m^w)$ from a call to LEARN-METHOD with $i = 0$, $f = n$, and τ, as well as possibly other methods from calls to LEARN-METHOD with different parameters. Method m will be applicable to state s_0.

Thus, m can be used to reduce $\langle \tau \rangle$ into $m^w = \langle t_0, t_1, \ldots, t_k \rangle$ from state s_0. I will show that m^w can be further reduced (if necessary) with other methods that have been previously learned into a plan that is applicable to s_0.

If t_0 is primitive and corresponds to the action $a = (a^h, a^\phi, a^-, a^+)$, then a must be applicable to state s_0, producing state $s - 1$. If a were not applicable to state s_0, then some earlier action in the plan would have to have produced an effect that is a precondition of a, and this additional action would be represented in a subtask prior to t_0.

If t_0 is instead nonprimitive, then it corresponds to an indexed method instance $x = (x^h, x^+, x^w, x^\phi, x^b, x^e)$. The method $m' = (x^h, x^\phi, x^w)$ of which x is an indexed instance must be applicable to s_0. Because this method was learned earlier, either $x^b > 0$ or $x^e < n$. Thus, the portion of π from x^b to x^e is a plan π' of length $n' < n$. The inductive hypothesis states that m' can be used to solve the HTN planning problem $\Psi'[h] = (\Sigma[h], s_0, x^h)$. Execution of the solution plan to $\Psi'[h]$ results in a state s'.

The same argument holds for the sub-problem in which $\langle t_1, t_2, \ldots, t_k \rangle$ must be decomposed from state s'. Eventually one will reach a sub-problem with no tasks left to accomplish, and the empty plan will complete the solution. Thus, $\Psi[h]$ can be solved using the learned methods.

□

The intuition behind the previous proof is that an HTN planner can replay the decisions that HTN-MAKER observed to re-generate the relevant parts of the plan π, which is already known to be a solution to $\Psi[c]$.

Lemma 5. *Let $\Sigma[c]$ 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 from M.*

Then, if M can be used to solve a problem $\Psi[h]$ then M' can be used to solve $\Psi[h]$ as well.

Proof. If subsumption checking 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 3. *Let $\Sigma[c] = (C, P, O)$ 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$(\Sigma[c], E, \mathcal{T}, \emptyset)$ *can be used to solve the HTN equivalent to every classical planning problem expressible using $\Sigma[c]$ and $\mathcal{T}$.*

Proof. Consider the set S of states representable in $\Sigma[c]$ and the set of goal statements G that have an equivalent annotated task in $\mathcal{T}$. Every solvable problem in $\Sigma[c]$ with an equivalent HTN problem has the form $\Psi[c] = (\Sigma[c], s_0, g)$ where $s \in S$ and $g \in G$. Because the sets C, P, and O are finite, so are the sets S and G, and because the sets S and G are finite, there are 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 4 and 5 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. I 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 by examining a solution to every expressible problem. Although the theoretical result shows only this worst-case behavior, experience indicates that far fewer problems are needed in practice. In one experiment, an HTN planner was 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. The experiments described in Chapter 6 show that in most cases, a few hundred randomly generated learning examples are sufficient.

3.5.3 Expressiveness

One of the advantages of HTN planning over classical planning is that the former is strictly more expressive than the latter [13]. The relationship is analogous to that between context-free languages and regular languages. It is clear that if we restrict ourselves to solving HTN problems that have an equivalent classical planning problem as discussed in Definition 40, then this advantage is negated. Indeed, learning methods that can be used to solve any arbitrary HTN planning problem is quite difficult. Because tasks used in HTN planning problems do not necessarily affect the state in any regular fashion, this would require a formalism based on some idea other than goal regression.

However, there are HTN planning problems that have no equivalent classical planning problem yet can be solved using methods learned by HTN-Maker. We refer to the set of HTN planning problems that can be solved using methods learned by HTN-Maker as classically-partitionable planning problems, because they can be partitioned into a sequence of classical planning problems.

Definition 51. A **classically-partitionable planning problem** is a triple $\Psi[p] = (\Sigma[c], s_0, G)$, where $\Sigma[c]$ is a classical planning domain, s_0 is a state in that domain, and $G = \langle g_0, g_i, \ldots, g_n \rangle$ is a sequence of sets of goal atoms.

Definition 52. A plan π is a **solution** to a classically-partitionable planning problem $\Psi[p] = (\Sigma[c], s_0, G)$ with $G = \langle g_0, g_1, \ldots, g_n \rangle$ if and only if π may be partitioned

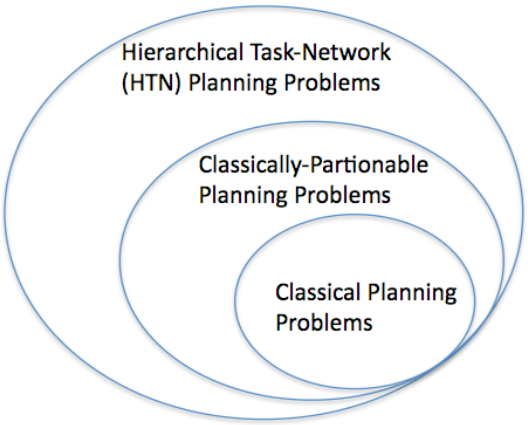

Figure 3.10: Relationships between classes of planning problems

into a sequence of subplans $\Pi = \langle \pi_0, \pi_1, \ldots \pi_n \rangle$ such that π_0 is applicable to s_0, producing a state s_1 that satisfies the goals g_0 and each successive π_i is applicable to state s_i, producing a state s_{i+1} that satisfies the goals g_i.

Definition 53. Two planning problems Ψ and Ψ' are **analogous** if and only if every solution to Ψ is also a solution to Ψ' and every solution to Ψ' is also a solution to Ψ. Ψ and Ψ' may be classical planning problems, classically-partitionable planning problems, or HTN planning problems.

Lemma 6. *For each classical planning problem $\Psi[c] = (\Sigma[c], s_0, g)$, there exists an analogous classically-partitionable planning problem $\Psi[p]$.*

Proof. Let $\Psi[p]$ be $(\Sigma[c], s_0, \langle g \rangle)$. □

Lemma 7. *There exists a classically-partitionable planning problem $\Psi[p] = (\Sigma[c], s_0, G)$ that does not have an analogous classical planning problem.*

Proof. Let $\Sigma[c]$ be the description of the BLOCKS-WORLD domain that we have been using as an example, with `A` as the only constant. Let s_0 be {`(on-table A)`, `(clear A)`, `(hand-empty)`}. Let $G = \langle g_0, g_1 \rangle$, such that g_0 is {`(holding A)`} and g_1 is {`(on-table A)`}.

There are 9 different sensible goal combinations expressible in this domain, and thus 9 sensible planning problems that use $\Sigma[c]$ and s_0. (I ignore goal states that are impossible; since no plan will solve them and since $\Psi[p]$ does have a solution, none of them can be analogous to it.) Of the 9 sensible goal combinations, 8 of them involve only the three atoms in the initial state. The empty plan is a solution to each of these problems, but it is not a solution to $\Psi[p]$. The remaining sensible goal combination is {`(holding A)`}. The plan ⟨ `!PickUp( A )` ⟩ is a solution to this problem, but not to $\Psi[p]$. Since we have exhaustively examined all classical planning problems that could possibly be analogous to $\Psi[p]$ and ruled each one out, there is none. □

Lemma 8. *For each classically-partitionable planning problem $\Psi[p] = (\Sigma[c], s_0, G)$ and finite set of annotated tasks $\mathcal{T}$ such that each goal set g_i in $G = \langle g_0, g_1, \ldots, g_n \rangle$ has an equivalent annotated task in $\mathcal{T}$, there exists an HTN planning problem $\Psi[h]$ that is analogous to $\Psi[p]$.*

Proof. Let $\Sigma[c] = (C, P, O)$ and $\Sigma[h] = (C, P, O, T, M)$, where T contains the heads of the tasks in $\mathcal{T}$ and M is some sufficient set of methods. Let $\Psi[h] = (\Sigma[h], s_0, w_0)$, with $w_0 = \langle t_0, t_1, \ldots, t_n \rangle$, such that each t_i is the equivalent annotated task of g_i. □

Lemma 9. *There exists an HTN planning problem $\Psi[h] = (\Sigma[h], s_0, w_0)$ that does not have an analogous classically-partitionable planning problem.*

Proof. Let $\Sigma[h]$ be a domain with two actions, `(!First)` and `(!Second)`, one task, `AnBn`, and two methods for that task. The first method has no preconditions or subtasks. The second method has no preconditions and the following subtasks: ⟨ `(!First) (AnBn) (!Second)` ⟩. Let s_0 be the empty set and w_0 be ⟨ `(AnBn)` ⟩.

Since there are no predicates in this domain, the only possible goal statement is the empty set. Any classically-partitionable planning problem with each of its goal statements being the empty set could be solved by the plan ⟨ `(!First)` ⟩, but this does not solve $\Psi[h]$. Thus, there is no analogous classically-partitionable planning problem. □

Theorem 4. *The set of classically-partitionable planning problems is strictly more expressive than the set of classical planning problems, but strictly less expressive than the set of HTN planning problems.*

Proof. This follows directly from Lemmas 6 through 9. □

Theorem 5. *Let $\Sigma[c] = (C, P, O)$ 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$(\Sigma[c], E, \mathcal{T}, \emptyset)$ *can be used to solve an HTN analogue to each classically-partitionable planning problem expressible using $\Sigma[c]$ and $\mathcal{T}$.*

Proof. Theorem 3 guarantees that a set of learning examples exists such that M will be able to solve the HTN equivalent of any classical planning problem. A classically-partitionable planning problem can be partitioned into a sequences of classical planning problems, and a solution to the sequence of their HTN equivalents will also be a solution to the classically-partitionable planning problem. □

3.5.4 Complexity

There are many potential ways to measure the size of the input to HTN-MAKER, which makes complexity analysis difficult. It is clear that the worst-case, average-case, and best-case running time is directly proportional to the number of learning examples, since each learning example is processed exactly once. Thus, we could say that HTN-MAKER is a $\Theta(|E|)$ algorithm, but this is not very illuminative.

Alternatively, we could consider the length of the longest plan in E, which we will call Π, and the number of annotated tasks in $\mathcal{T}$. Based on these, in the worst case the number of times the LEARN-METHOD subroutine is called is bounded by $O(|E| * |\Pi|^2 * |\mathcal{T}|)$. The main loop of the LEARN-METHOD subroutine will execute at most $|\Pi|$ times per call to LEARN-METHOD. The inner loop of LEARN-METHOD that checks each indexed method instance to determine whether or not it is potentially useful runs once for each method that has previously been learned from

this trace, which is bounded by $|\Pi|^2 * |\mathcal{T}|$. All other operations are constant with respect to the number of learning examples, length of longest plan, and number of annotated tasks. Thus, we could say that the worst-case time complexity of HTN-MAKER is $O(|E| * |\Pi|^5 * |\mathcal{T}|^2)$.

What about the number of predicates in the domain, the numbers of preconditions, negative effects, and positive effects in the largest action(s), the numbers of preconditions and positive effects in the largest annotated task(s), and the number of variables used in various and sundry constructs? All of these control to some extent the runtime of several lines in the algorithm that are constant with respect to the measures of input size that we had previously discussed. An in-depth analysis of these factors would be tedious and of limited value, so we will be satisfied with the knowledge that the runtime of the generic HTN-MAKER algorithm is polynomial in those factors that would have the most impact.

Most of the implementation details discussed in Section 3.4 do not impact the complexity of the algorithm, but checking methods for equivalence or subsumption as explained in section 3.4.4 is worthy of a closer look. This check occurs once for each call to the LEARN-METHOD procedure, and involves comparing the new method against all existing methods (twice with subsumption). The number of existing methods is bounded by $|M| + |E| * |\Pi|^2 * |\mathcal{T}|$. The subsumption checking problem is analogous to the Associative-Commutative Unification problem, which is constant with respect to the input size measures that we are considering, but has been shown to be NP-complete with regard to the number of variables in the formulas (methods) [36]. As the experiments described in Chapter 6 show, this is not generally a problem.

Chapter 4

Learning in Nondeterministic Domains

Because ND-SHOP2 uses standard HTN methods as its knowledge artifacts, some methods learned by HTN-MAKER can be used to solve problems in nondeterministic domains. However, some particular method structures are poorly suited to solving these sorts of problems. Thus, I have developed an algorithm HTN-MAKERND that extends HTN-MAKER to specifically learn methods that will be useful in nondeterministic domains.

In this section the examples will be based on a version of the BLOCKS-WORLD domain in which several actions have multiple possible outcomes. In the deterministic version, the `(!Pickup ?a)` operator adds `(holding ?a)` to the state and removes `(on-table ?a)`, `(clear ?a)`, and `(hand-empty)` from the state. In the nondeterministic version this is one of two possible outcomes; the other is that the state is unchanged. This might represent, for example, the failure of a robotic controller to successfully grasp the block. The `(!Putdown ?a)`, `(!Stack ?a ?b)`, and `(!Unstack ?a ?b)` operators will similarly have a second possible outcome in which the state is unchanged. Additionally, the `(!Stack ?a ?b)` operator will have a third outcome, with the same effects we would normally expect from `(!Putdown ?a)`, and `(!Unstack ?a ?b)` will also have a third outcome, with the same effects we would

normally expect from `(!Unstack ?a ?b)` followed by `(!Putdown ?a)`.

4.1 The HTN-MakerND Algorithm

The algorithm for learning methods that will be effective in solving problems in nondeterministic domains consists of a pre-processing step, a version of HTN-Maker that has its choices further restricted, and a post-processing step.

In HTN-MakerND, a learning example continues to consist of a state and a plan, although a plan is not a solution to a problem in a nondeterministic domain. Instead, from one state and policy many different execution traces may be generated, each representing what "really happened" during one use of the policy. Although it has a different origin, an execution trace is functionally equivalent to a plan, and a learning example can be created for each one.

Consider, for example, the state { `(on-table B)`, `(on-table C)`, `(on A C)`, `(clear A)`, `(clear B)`, `(hand-empty)` }. If the goal is to have block `A` on block `B`, the policy shown in Figure 4.1 is a strong-cyclic solution to this problem. The states are given symbols to ease reference to an individual state, but these are not part of the policy. The first row of the table states that when in the Γ state (which is the initial state described above) the agent should take the action `(!Unstack A C)`. Because of the definition of the nondeterministic action `(!Unstack A C)`, this will result in one of three states: either Γ, Δ, or Θ. The second row of the table states that when in the Δ state the agent should take the action `(!Stack A B)`. This will result in either the Δ state, the Θ state, or the Λ state. The third row of the table states that when in the Θ state the agent should take the action `(!Pickup A)`. This will result in either the Δ state or the Θ state. The fourth row states that the policy is not defined for the Λ state. This is fine, because this state satisfies the goals.

Figure 4.2 shows the execution structure of this policy applied to the initial state Γ. An infinite number of different execution traces might be generated by following this execution structure. Among them are ⟨ `(!Unstack A C)`, `(!Stack A B)` ⟩ and

Symbol	State	Action
Γ	(on-table B) (on-table C) (on A C) (clear A) (clear B) (hand-empty)	(!Unstack A C)
Δ	(on-table B) (on-table C) (clear B) (clear C) (holding A)	(!Stack A B)
Θ	(on-table A) (on-table B) (on-table C) (clear A) (clear B) (clear C) (hand-empty)	(!Pickup A)
Λ	(on-table B) (on-table C) (on A B) (clear A) (clear C) (hand-empty)	

Figure 4.1: A policy in the nondeterministic BLOCKS-WORLD domain

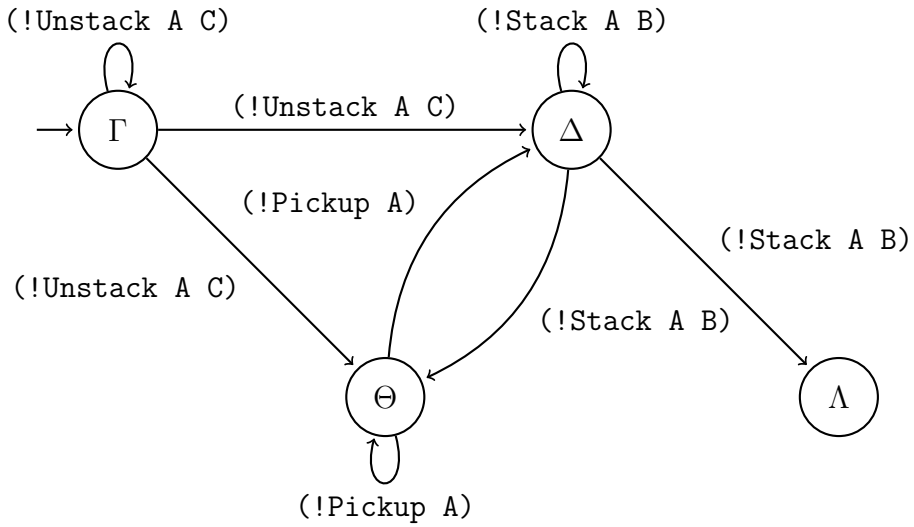

Figure 4.2: The execution structure of the policy shown in Figure 4.1

⟨ (!Unstack A C), (!Unstack A C), (!Stack A B), (!Pickup A), (!Pickup A), (!Stack A B) ⟩.

The HTN-MAKER algorithm has no reasoning mechanism for considering multiple potential outcomes of an action. Rather than creating one, the system pre-processes each nondeterministic operator into a set of deterministic operators with similar names. Thus, the 4-operator nondeterministic BLOCKS-WORLD domain becomes a 10-operator deterministic BLOCKS-WORLD domain. When generating a plan from a policy, it is known which possible outcome actually occurred, and thus an additional pre-processing step replaces the nondeterministic actions in the learning examples with their deterministically partitioned counterparts. Thus, the latter plan listed in the previous paragraph would become ⟨ !Unstack2 A C), (!Unstack1 A C), (!Stack3 A B), (!Pickup2 A), (!Pickup1 A), (!Stack1 A B) ⟩. The learning procedure works with these determinized operators and plans, and produces methods whose primitive tasks are determinized operators. The final post-processing step after learning is to replace each determinized primitive task in the subtasks of a method with the nondeterministic version from which it was split.

I have previously mentioned that not all methods are equally useful for solving planning problems in nondeterministic domains. One obvious way in which a method could be unfit is to have two or more consecutive primitive subtasks, when at least one of the possible outcomes of the first subtask is a state in which the second subtask is not applicable. When using such a method, ND-SHOP2 will create several pairs of state and task network, and from some of them it will not be able to continue. Thus, it would need to backtrack and select a different method.

This is the most egregious example of a broader category of trouble: the more subtasks a method has, the less likely it will be that all possible outcomes of its early subtasks lead to situations in which its later subtasks can be processed. I have identified, however, situations in which these sorts of problems cannot occur. To accomplish this, when learning in nondeterministic domains the choice made in Line 15 of Algorithm 7 is even more restricted than as described in Section 3.4.1. Methods that meet this new restriction are *nd-friendly*. This requirement is similar, but not equivalent, to the right-recursive property of Definition 48, in that only one

primitive task is allowed per method, but nonprimitive tasks do not necessarily need to be recursive. Thus, it is possible for a method to be right-recursive, nd-friendly, both right-recursive and nd-friendly, or neither.

Definition 54. An HTN method is **nd-friendly** if and only if any of the following is true:

1. The method has no subtasks.
2. The method has exactly two subtasks; the first is primitive and the second is nonprimitive.

When ND-SHOP2 uses an nd-friendly method, one of a small number of possible scenarios occurs. If the method had no subtasks, then it must have been a trivial method as described in Section 3.4.2, and no further work needs to be accomplished. Otherwise, application of the first subtask produces a set of pairs of a state and a task network consisting of the second subtask (followed by any later top-level tasks that were in the initial problem). In each of those states, HTN-MAKERND may now select an applicable method to what had been the second subtask. In the ideal case, there will be at least one applicable method for each state. If there is not, this is due to a lack of knowledge or a dead-end in the domain, not because of a failure in representation.

4.2 Properties

As is the case when planning in deterministic domains, I show that the use of methods learned by HTN-MAKERND is sound, and that it is possible to learn a set of methods sufficient for solving all expressible problems in a domain that have solutions.

Theorem 6. *Let $\Sigma[c] = (C, P, O)$ be a nondeterministic planning domain, and $\Sigma[c]' = (C, P, O')$ be its determinization by splitting each nondeterministic operator in O into a set of deterministic operators in O', one for each possible outcome. Let E be a finite set of learning examples for $\Sigma[c]'$ and $\mathcal{T}$ be a finite set of*

annotated tasks for $\Sigma[c]'$. *Let* M *be the result of* HTN-MAKER$^{\text{ND}}(\Sigma[c]', E, \mathcal{T}, \emptyset)$, *after post-processing of determinized primitive tasks back into their nondeterministic versions. Let* $\Psi[c] = (\Sigma[c], s_0, g)$ *be a nondeterministic planning problem and* $\Psi[h] = (\Sigma[h], s_0, w_0)$ *be the equivalent nondeterministic HTN planning problem with* $\Sigma[h] = (C, P, O, T, M)$, $\tau = (\tau^h, \emptyset, g)$, *and* $w_0 = \langle \tau^h \rangle$. *Let* Π *be a policy generated by a sound HTN planner for nondeterministic domains as a strong solution to* $\Psi[h]$.

Then, Π *is a strong solution to* $\Psi[c]$ *(and thus,* HTN-MAKER$^{\text{ND}}$ *is sound with regard to strong planning).*

Proof. Suppose that Π is not a strong solution to $\Psi[c]$. There are three ways this could be true:

First, consider the possibility that the execution structure of Π includes a cycle. This is not possible, because if it were true then Π could not be a strong solution to $\Psi[h]$.

Alternatively, consider the possibility that there exists a state s reachable by following Π from s_0 such that there is no state s' such that $g \subseteq s'$ and s' is a Π-descendant of s in the execution structure of Π. That is, there are one or more states in the execution structure of Π from which it is impossible to reach a state that satisfies the goals. This could only be true if there existed a state that does not satisfy the goals but in which the preconditions of the trivial method for τ^h held, which is impossible.

Finally, consider the possibility that there exists a state s reachable by following Π from s_0 that does not satisfy the goals and for which the policy specifies no further action. Again, the only way this could be true of a policy that is a strong solution to $\Psi[h]$ is if there were a state that does not satisfy the goals but does contain the preconditions of the trivial method for τ^h, which is impossible.

□

Theorem 7. HTN-MAKER$^{\text{ND}}$ *is sound with regard to strong-cyclic planning, following the declarations of Theorem 6 with the exception that* Π *is merely a strong-cyclic solution to* $\Psi[h]$ *rather than a strong solution.*

Proof. There are only two ways in which Π could fail to be a strong-cyclic solution to $\Psi[c]$, corresponding to the second and third possibilities in the proof of Theorem 6, and the same arguments apply here to show that they are impossible. □

Theorem 8. HTN-MAKER$^{\text{ND}}$ *is sound with regard to weak planning, following the declarations of Theorem 6 with the exception that* Π *is merely a weak solution to* $\Psi[h]$ *rather than a strong solution.*

Proof. If Π is not a weak solution to $\Psi[c]$, then there does not exist a path from s_0 to a state that satisfies g in the execution structure of Π. Since Π is a weak solution to $\Psi[h]$, this must mean that there exists a state that does not satisfy g but does satisfy the preconditions of the trivial method for τ^h, and as before this is impossible. □

Theorem 9. *Let* $\Sigma[c] = (C, P, O)$ *be a nondeterministic 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$^{\text{ND}}(\Sigma[c, E, \mathcal{T}, \emptyset)$ *can be used to create a weak solution for the HTN equivalent to each nondeterministic planning problem expressible using* $\Sigma[c]$ *and* $\mathcal{T}$ *that has a weak solution.*

Proof. Theorem 3 already shows this for deterministic planning domains, and a plan that solves a deterministic planning problem can easily be converted into a policy that is a weak solution to the nondeterministic version. □

Theorem 10. *Let* $\Sigma[c] = (C, P, O)$ *be a nondeterministic 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$^{\text{ND}}(\Sigma[c], E, \mathcal{T}, \emptyset)$ *can be used to create a strong-cyclic solution for the HTN equivalent to each nondeterministic planning problem expressible using* $\Sigma[c]$ *and* $\mathcal{T}$ *that has a strong-cyclic solution.*

Proof. There are a finite number of planning problems in each nondeterministic planning domain, following the same reasoning as in Theorem 3. As in the proof

of that theorem I will show that HTN-MAKERND is capable of learning sufficient information from a solution to each of these problems to solve that problem, and thus that if it learns from a solution to each problem it will be able to solve all problems. In the case of deterministic planning domains, the solution to a planning problem was a plan and the information contained in that plan could be extracted by HTN-MAKER into a series of methods that would reproduce its fundamental properties. In the case of nondeterministic planning domains, the solution to a planning problem is a policy. Thus, I must show that for each policy Π there are a finite number of learning examples E such that the methods learned by HTN-MAKER from E will be sufficient to reproduce the policy Π.

The execution structure of a policy has a finite number of states and edges. For each such state, at most one applicable method needs to be learned (none for states with no children). Thus, at most one learning example will be needed per state in the execution structure of the policy. □

Chapter 5

Estimating HTN Method Values

The methods learned by HTN-Maker and HTN-MakerND are sufficient for solving HTN planning problems in deterministic and nondeterministic planning domains, respectively, but they do not necessarily find *high quality* solutions. As planning technology has improved, the quality of solutions has become an increasingly important metric by which planning systems are judged [72].

High-quality can have several different meanings when applied to plans. The most common way of thinking about solution quality is a strict number of actions: shorter plans are preferred over longer ones. With this metric it is easy to define an **optimal** solution to a planning problem: one such that no shorter plan is also a solution.

In some domains there are actions that are clearly more complicated than others. For example, we have previously discussed the Logistics domain. If this were used to produce plans that would be executed in the physical world by humans and machines, we might argue that driving a truck between two locations is more expensive in both time and materials than loading a package onto a truck, and that driving a truck between two locations is less expensive (but slower) than flying an airplane between those same locations. Thus, we might assign a numeric *cost* to each action and prefer plans that take many low-cost actions over those that take few high-cost actions.

There are other, more complicated ways of measuring plan quality as well.

Rather than making the truck driving operator have a fixed cost, some systems would calculate a cost based on the distance it is travelling, or perhaps even the current weight of the vehicle, traffic levels, or other factors. In *overconstrained planning* it may be impossible to accomplish all goals, and a high-quality plan would be one that achieves as many goals as possible. In other cases a plan might be considered high-quality if it adheres closely to standard operating procedures, even if a novel plan would solve the problem with lower cost.

Because plan quality is such a nebulous concept, I have chosen to focus on cost-based quality metrics, but have attempted to design a framework that is flexible enough to accommodate some other ideas of plan quality as well.

Learning methods that produce higher-quality plans than the methods that HTN-Maker already learns would require either re-engineering the system or modifying the methods that it learns through some sort of post-processing. In either case, this would be quite difficult. Fortunately, HTN-Maker already learns methods that can be used to produce high-quality plans in many cases. The problem is that these "good" methods are mixed with a great number of other methods that are not so effective.

The default heuristic used by the HTN-Solver planner is that, when multiple methods are applicable to the current state and task network, the one that was learned first should be explored first, with the other options saved for possible backtracking. This heuristic works reasonably well in that it encourages solutions to simple sub-problems to be preferred over solutions to more complex problems, because of the way HTN-Maker orders the subplans that it analyzes from a single learning example. Because HTN-Maker typically uses many learning examples presented in an arbitrary order, however, there are many cases in which this heuristic causes suboptimal choices.

Thus, the objective is to design a way that the HTN planner can make an informed choice as to which of many potentially applicable methods is most likely to lead to a high-quality plan, and to update its expectations about methods based on the experience that it gains when using them. Reinforcement learning provides a natural framework for exploiting and improving such knowledge, so it forms the

basis of this system.

5.1 A Model For Learning Valued Methods

Recall Algorithm 2, which is a straightforward implementation of an HTN planner, similar to SHOP. In order to model the problem of finding a high-quality solution to an HTN planning problem as a reinforcement learning problem, the reinforcement learning agent is the procedure that makes a decision (Line 12) when multiple methods are applicable. The rest of the planning system is the environment.

Automated planning and reinforcement learning use much of the same terminology, but describing this system requires the lexicon of both, not used in the same way. In reinforcement learning, the *state* is the current situation in which the agent finds itself able to make a decision. If the decision to be made is what method should be used to reduce a certain task, then this state must consist of both the current planning state (s_0 in Algorithm 2) and the current task network (w_0 in Algorithm 2). The *actions* in reinforcement learning are the decisions that the agent may make. In this case, they refer not to actions in the planning sense, but to the HTN methods that are applicable to the current planning state and task network.

Decomposing tasks is an episodic activity; once a task has been fully decomposed no more task reductions are necessary. It is also a recursive activity in that the decomposition of a top-level task typically requires the decomposition of one or more subtasks as well. In cases where a task network contains more than one nonprimitive task this is treated as multiple distinct episodes, even though decisions made when decomposing the first task will affect the state from which the second task must be decomposed. After each task reduction, the agent receives a reward from the environment. The *return* for reducing a nonprimitive task t with a method $m = (m^h, m^\phi, m^w)$ is the sum of immediate reward r and the returns from the reductions of each of the nonprimitive tasks in m^w, as in Equation 5.1. The return for applying an action to a primitive task is 0 in this formalism, although it would be equivalent to move the rewards associated with a reduction to there. Because

task decompositions are reasonably short episodes, this formalism does not discount future rewards as some reinforcement learning systems do.

$$R(t) = r + \sum_{t_i \in m^w} R(t_i) \tag{5.1}$$

A precise model of exactly what planning states and task networks will be reached as a result of each possible method application and the possible rewards that could be received is not immediately available, and generating one would be quite difficult. However, generating experience is quite easy; we simply provide the HTN planner with a problem and allow it to search for a solution. Thus, this formalism uses a Monte Carlo technique for solving the reinforcement learning problem.

At all times (continuing from one episode to the next) the agent maintains an estimated value of using a particular method to reduce a task. This **method-value function** has the signature $Q : M \to \mathbb{R}$, and the value of $Q(m)$ is called the **Q-value** of m. Each method is applicable only to tasks sharing one particular task template, and because these methods do not contain any constant terms, they will be equally useful for any task following that template. Thus, it is only necessary to maintain a single value for each method, which will be used for all tasks matching the head of that method. Note also that the planning state is not included as part of this value calculation. Instead, there is an assumption that a method will be equally effective in all situations in which it is applicable. This assumption is most certainly false, but it drastically constricts the space of values that must be estimated and makes the problem tractable.

The Q-value of a method represents the total return that the agent expects to receive by using the method to reduce a task, assuming that it makes future decisions within the episode sensibly. Thus, if the Q-values are accurate estimates of the true values of the methods, the agent will maximize its return by always selecting the highest-valued method among applicable candidates. If the rewards given to the agent are carefully matched to a particular notion of plan quality, then the agent will also maximize the quality of plans that it produces by always choosing the method with highest Q-value.

To ease the process of using and updating Q-values, with each method is stored the Q-value itself, $Q(m)$, and a count of the number of times that method has been observed or used, $k(m)$.

5.2 Q-Maker: Learning Methods & Initial Method Values

Rather than beginning the reinforcement learning process with the Q-values of methods assigned to equal values or arbitrarily, it is possible to compute an initial estimate of the value of a method as that method is being learned. The Q-MAKER procedure is an extension of HTN-MAKER that computes this initial estimate with each new method that it learns. In order to do so, Q-MAKER must have a way of knowing or guessing what rewards the planner would have received if it generated the plan in the learning example that it is analyzing. Because rewards in this formalism will be closely tied to the quality of the plan being generated, this is not a problem.

Algorithm 8 shows the basic Q-MAKER algorithm, which is very similar to Algorithm 6 for HTN-MAKER. The first difference is on Line 13, where the LEARN-METHOD subroutine returns both a method (as in HTN-MAKER) and the total return R that it believes would have been generated following the use of that method and other decisions that produce the plan from the learning example. If the newly-learned method is a copy of one already known to the system, the return is averaged into the existing estimate of its value (Line 15) and the count of times the method has been observed is incremented (Line 16). Otherwise, the method is added (Line 18), its initial estimate is set to the expected return from this trace (Line 19), and its count is set to 1 (Line 20).

5.3 Q-Reinforce: Refining Method Values

Although the initial value estimates produced by HTN-MAKER are useful, they may not be sufficient to produce high-quality plans in some circumstances. Because

these values are based entirely on the plans that were part of the learning examples, they represent the utility of the methods only when used to produce those particular plans. In practice, it will often be possible to use those methods in circumstances that did not appear in the learning examples and that result in very different returns.

The Q-REINFORCE algorithm, which uses reinforcement learning to improve the initial value estimates based on problem-solving experience, is designed to combat this bias. Algorithm 9 shows pseudocode of Q-REINFORCE.

Q-REINFORCE uses recursion in a different manner to ease the propagation of returns, but will produce similar results to Algorithm 2 (other than that it returns more than just a plan). Rather than recursing on the entire new problem after each reduction of a nonprimitive task or selection of an action, it recurses only on the sub-problem created when reducing a nonprimitive task, which makes it easier to treat each decomposition as a distinct, though sometimes recursively related, reinforcement learning episode. The input to Q-REINFORCE is an HTN planning domain, an initial state, an initial task network, and a method-value and method-count function. The output is a plan that accomplishes the tasks, a total return received from generating that plan, and updated method-value and method-count functions. For top-level calls only the method-value and method-count functions are needed, but the plan and return are used to combine the results of recursive calls to solve subproblems.

Q-REINFORCE begins by initializing an empty plan (Line 3), a current state (Line 4), and a return of zero (Line 5). It then processes each task in the initial task network in the order in which they appear (Line 6). If the current task is primitive and an action can be generated that matches it and is applicable in the current state, such an action is generated (Line 7). Applying an action does not involve a choice by the reinforcement learning agent. Thus, no reward is earned and no values are updated. However, the action is applied to the current state and appended to the plan generated thusfar (Lines 9 - 10).

If the current task is instead nonprimitive, Q-REINFORCE selects from among the applicable method instantiations using a uniform distribution (Line 15). This makes Q-REINFORCE an *off-policy* learner [73], since the policy that it is attempting to

improve makes a greedy selection based on the method values but Q-REINFORCE does not follow this policy itself. This is necessary to override any bias that the initial method values may have based on the plans from which they were learned. Q-REINFORCE makes a recursive call to process the subtasks of the selected method (Line 16). The plan that was generated to solve that subproblem is appended to the plan to solve the larger problem, and is applied to advance the current state (Lines 17 - 18). Both the return received while solving the subproblem (R') and the reward received as a direct result of the use of the selected method (r) are added to the total return received while solving the larger problem (Line 19), and both are averaged into the estimated value of the method used (Lines 20 - 21).

5.4 Q-Shop: Planning With Method Values

The Q-REINFORCE algorithm is entirely *exploratory*, but there is an additional program, Q-SHOP, which *exploits* the knowledge that was extracted by Q-MAKER and refined by Q-REINFORCE. Q-SHOP is a straightforward implementation of Algorithm 2 in which the choice made on Line 12 is of the applicable method with highest value. This remains a backtracking point in the case that the planner is unable to continue after making such a decision. Alternatively, Q-REINFORCE could be made into an exploitative planner by changing its selection in Line 15 from using a uniform probability distribution to selecting greedily based on the existing method values.

Algorithm 8: A high-level description of the Q-MAKER procedure. The input includes a classical planning domain description $\Sigma[c]$, a finite set of learning examples E, a finite set of annotated tasks $\mathcal{T}$, a (possibly empty) finite set of HTN methods M, and a method value function Q and method count function k. The output is an updated set M of HTN methods and functions Q and k.

Procedure Q-MAKER($\Sigma[c], E, \mathcal{T}, M, Q, k$)
begin
  foreach learning example $e = (s_0, \pi) \in E$ **do**
    initialize $X \leftarrow \emptyset$
    initialize $\vec{S} \leftarrow \langle s_0 \rangle$
    for $i \leftarrow 1$ to k **do**
      $s_i \leftarrow \gamma(s_{i-1}, a_{i-1})$
      $\vec{S} \leftarrow \vec{S} \cdot \langle s_i \rangle$
    for $f \leftarrow 1$ to k **do**
      for $i \leftarrow f - 1$ down to 0 **do**
        foreach annotated task $\tau = (\tau^h, \tau^\phi, \tau^+) \in \mathcal{T}$ **do**
          if $\tau^\phi \subseteq s_i$ and $\tau^+ \subseteq s_f$ **then**
            $(m, R) \leftarrow$ LEARN-METHOD($\pi, \vec{S}_\pi, \tau, X, i, f$)
            if $m \in M$ **then**
              $Q(m) = \frac{Q(m) * k(m) + R}{k(m)+1}$
              $k(m) = k(m) + 1$
            else
              $M \leftarrow M \cup \{m\}$
              $Q(m) = R$
              $k(m) = 1$
            $X \leftarrow X \cup \{(m^h, \tau^+, m^w, m^\phi, i, f)\}$
  return (M, Q, k)
end

Algorithm 9: An agent that learns from task reduction experience

```
1  Procedure Q-REINFORCE(Σ[h], s_0, w_0, Q, k)
2  begin
3      π ← ⟨⟩
4      s ← s_0
5      R ← 0
6      for t_i ∈ w_0 do
7          if t_i is primitive then
8              if ∃ an o ∈ O and u such that u(o^φ) ⊆ s and u(o^h) = t_i then
9                  s ← (s \ u(o^−)) ∪ u(o^+)
10                 π ← π · ⟨u(o)⟩
11             else
12                 return FAIL
13         else
14             if ∃ an m ∈ M and u such that u(m^φ) ⊆ s and u(m^h) = t_i then
15                 Nondeterministically select such an m and u
16                 (π′, R′, Q, k) ← Q-REINFORCE(Σ[h], s, u(m^w), Q, k)
17                 π ← π · π′
18                 s ← γ(s, π′)
19                 R ← R + R′ + r
20                 Q(m) ← (Q(m)*k(m)+R′+r) / (k(m)+1)
21                 k(m) ← k(m) + 1
22             else
23                 return FAIL
24     return (π, R, Q, k)
25 end
```

Chapter 6

Experimental Evaluation

I have performed a variety of experiments to evaluate the effectiveness of HTN-Maker, HTN-MakerND, and the combination of Q-Maker, Q-Reinforce, and Q-Shop. The first section of this chapter describes in detail the planning domains used in the experiments. The next describes an experiment to measure the rate at which HTN-Maker learns from examples and reports on the results. After that I discuss the speed at which a reimplementation of Shop is able to solve problems using the knowledge learned by HTN-Maker, comparing it to modern classical planners and to itself using specially hand-crafted methods. The fourth section considers the speed at which ND-Shop2 is able to solve nondeterministic planning problems using methods learned by HTN-MakerND, comparing against a benchmark planner for nondeterministic planning domains. Finally, I consider both the quality of plans produced by Q-Shop using methods with values learned by Q-Maker and Q-Reinforce and the time required to produce those plans compared to several different planners.

6.1 Domains

Very many domains have been described and codified over the years to evaluate planning systems. Not all of them, however, are suitable for use in evaluating this work. Most notably, many domains introduced during the last decade have had

multiple variants: one or more that utilize more complex representations such as numeric quantities, durative actions, universal and existential quantifiers, or conditional effects, and one that conforms to the traditional formalism used in this document for classical planning. HTN-Maker is not capable of reasoning about the various extensions to planning languages that are an important part of the complex variants of these domains, so they are not usable. (HTN-MakerND is capable of reasoning about one particular non-classical extension: nondeterministic actions.) HTN-Maker can work with the traditional (Strips) variants, but unfortunately when all of the extra features are removed many of these domains reduce to something that is directly equivalent to another existing domain or is otherwise uninteresting.

Other domains are usable by HTN-Maker, but would not be useful for evaluating it. In order to compare the speed of planning with methods learned by HTN-Maker to non-hierarchical planning, large, complicated planning problems are needed. Typically, the primary way to increase the complexity of a planning problem is to increase the number of goals that must be accomplished. In order to strictly use the notion of equivalence between a classical planning problem and an HTN planning problem as in Definition 40, the equivalent to a planning problem with 100 goals will have an initial task network consisting of one task, and that task will be annotated with 100 goals. This is feasible, but because methods are for accomplishing a specific task, HTN-Maker will not be able to learn to accomplish a 100-goal task without observing plans in which all 100 goals are accomplished. Furthermore, a method that can be used to solve a 100-goal task will be useless for solving 99-goal and 101-goal tasks.

Thus, these experiments do not use an equivalent annotated task to each set of goals that appears in a planning problem. Instead, the many goals of a complex classical planning problem are partitioned similarly to the classically-partitionable planning problems described in Section 3.5.3, but in such a way that once a goal becomes true it will always remain true. The resulting HTN planning problems have not a single task in their initial task networks but instead a sequence of tasks, one for each goal. In those domains reported on here it is fairly easy to create a

serializable partition of goals into discrete tasks; in many others it does not appear to be possible. This does not mean that HTN-Maker is unusable in these other domains, but methods that it learns from small examples would not be usable to solve large problems in those domains.

6.1.1 Blocks-World

The first domain of interest is Blocks-World, which has also been used as an example throughout this text. This domain has been used for evaluating AI systems for a very long time, and was utilized in the second international planning competition (IPC-2) in 2000. As described earlier, the Blocks-World domain contains a number of cubical blocks, a robotic arm, and a flat tabletop with presumably infinite space. At all times, each block is either being held by the arm, sitting on the table, or sitting within a pile of blocks, the bottom of which is on the table. Although other formulations exist, in this work I use one in which there are four deterministic operators, shown in Figure 2.5.

The goal of planning problems in the Blocks-World domain is to move the blocks into a new configuration. The problem generator used produces problems in which the initial and final configurations of all blocks are fully specified, so the difficulty of a problem is directly measured as the number of blocks. Thus, for the deterministic version of the domain two annotated tasks are needed, which together can describe any position in which we might want to store a block. These two annotated tasks are shown in Figure 6.1. The process of serializing these tasks in such a way that once a task has been accomplished it will never be un-accomplished is to first put the towers of blocks that will be built into an arbitrary order, then for each tower to issue the task for the bottom block of that tower, then the block above it, and so forth to the top before moving on to the next tower. This works because the methods learned by HTN-Maker never take actions that do not help to achieve the current task, and there are only two situations in which it can be beneficial to move a block: when that block is mentioned in the current task, or when that block is on top of a block that is mentioned in the current task. Thus,

the goal statement { `(on A B)`, `(on-table B)`, `(on C A)` } has the equivalent annotated task `(Make-3Pile C A B)`, but also the equivalent serialization of tasks ⟨ `(PutOnTable B)`, `(PutOnBlock A B)`, `(PutOnBlock C A)` ⟩.

```
( :annotated-task
  :head
  (PutOnTable ?a)
  :precondition
  {}
  :positive-effects
  { (on-table ?a) }
)
```

```
( :annotated-task
  :head
  (PutOnBlock ?a ?b)
  :precondition
  {}
  :positive-effects
  { (on ?a ?b) }
)
```

Figure 6.1: Annotated tasks in the deterministic BLOCKS-WORLD domain

Because a nondeterministic version of the BLOCKS-WORLD domain is frequently used for comparing planners that work with nondeterministic planning domains, I also used a nondeterministic version of the BLOCKS-WORLD domain when testing HTN-MAKERND. In this version of the domain, each operator has the possibility of having its usual effects or of having no effect at all. In addition, the `(!Stack ?a ?b ?c)` and `(!Unstack ?d ?e ?f)` have the possibility of dropping block `?a` (or `?d`) on the table. Because solving (strong-cyclically) nondeterministic planning problems is much more computationally difficult than solving deterministic planning problems, I was able to get meaningful results from much smaller problem sizes in the nondeterministic version. Thus, I decided to use a compromise between the equivalent HTN problems and the serialized goal partitions described above: instead of one task per block or one task per problem, there was one task per tower of blocks. This meant that I could use the `(Make-2Pile ?a ?b)`, `(Make-3Pile ?a ?b ?c)`, ..., `(Make-8Pile ?a ?b ?c ?d ?e ?f ?g ?h)` style of annotated tasks that were used in earlier examples.

6.1.2 Logistics

The second domain used in these experiments was also introduced earlier, in Section 3.4.4. The LOGISTICS domain models a delivery company trying to get all of its packages to their destinations, using trucks for intracity transport and airplanes for intercity transport. It was first introduced by Voloso [88], and was also used in the first international planning competition (IPC-1) in 1998. Unlike BLOCKS-WORLD, this domain contains constants of varying types and predicates that can only be true of certain types of terms. The operators for the LOGISTICS domain are shown in Figure 6.2. They are used to load/unload packages to/from a truck or airplane and the location of that vehicle, to drive a truck between two locations in the same city, and to fly an airplane between two airports.

The goal of planning problems in the LOGISTICS domain is to deliver the various packages to new locations; thus the number of packages is the natural metric for the difficulty of a problem. For this domain I used a single annotated task, which was shown in Figure 3.4b. Moving a package will never be useful unless the planner is working on a task related to that package, so any serialization of the tasks, one per package, will result in a state where all of the goals hold.

6.1.3 Zeno

The third domain is ZENO, which is somewhat similar to LOGISTICS. Rather than packages, in this domain passengers are transported between cities. There are no locations within cities, and thus no trucks. However, each flight by an aircraft consumes either one unit of fuel (when "flying") or two (when "zooming"). Aircraft can be refueled one unit at a time, but have a fairly small limit to the amount of fuel they can hold. The classical version of the domain uses standard predicate logic to encode the relationships between the integers zero through five to represent fuel levels. The ZENO domain was introduced in the third international planning competition (IPC-3) in 2002. Modelling limited numerical quantities makes the operators somewhat complex; instead of formal definitions there are descriptions in Figure 6.3.

```
( :operator
  :head
  (!UnloadTruck ?p - package
    ?t - truck ?l - location)
  :precondition
  { (in-truck ?p ?t),
    (truck-at ?t ?l) }
  :negative-effects
  { (in-truck ?p ?t) }
  :positive-effects
  { (pkg-at ?p ?l) }
)
( :operator
  :head
  (!UnloadPlane ?p - package
    ?a - plane ?l - location)
  :precondition
  { (in-plane ?p ?a),
    (plane-at ?a ?l) }
  :negative-effects
  { (in-plane ?p ?a) }
  :positive-effects
  { (pkg-at ?p ?l) }
)
( :operator
  :head
  (!Drive ?t - truck ?l - location
    ?m - location ?c - city)
  :precondition
  { (truck-at ?t ?l),
    (in-city ?l ?c),
    (in-city ?m ?c) }
  :negative-effects
  { (truck-at ?t ?l) }
  :positive-effects
  { (truck-at ?t ?m) }
)
```

```
( :operator
  :head
  (!LoadTruck ?p - package
    ?t - truck ?l - location)
  :precondition
  { (pkg-at ?p ?l),
    (truck-at ?t ?l) }
  :negative-effects
  { (pkg-at ?p ?l) }
  :positive-effects
  { (in-truck ?p ?t) }
)
( :operator
  :head
  (!LoadPlane ?p - package
    ?a - plane ?l - location)
  :precondition
  { (pkg-at ?p ?l),
    (plane-at ?a ?l) }
  :negative-effects
  { (pkg-at ?p ?l) }
  :positive-effects
  { (in-plane ?p ?a) }
)
( :operator
  :head
  (!Fly ?a - plan
    ?l - location ?m - location)
  :precondition
  { (plane-at ?a ?l),
    (is-airport ?l,
    (is-airport ?m) }
  :negative-effects
  { (plane-at ?a ?l) }
  :positive-effects
  { (plane-at ?a ?m) }
)
```

Figure 6.2: Operators for the LOGISTICS domain

Operator	Description
(!Board ?p ?a ?c)	Person ?p boards aircraft ?a, both of which must be located in city ?c.
(!Debark ?p ?a ?c)	Person ?p exits aircraft ?a, which must be in city ?c, and is now in city ?c.
(!Fly ?a ?c ?d ?f ?g)	Aircraft ?a flies from city ?c to city ?d, decreasing its fuel level from ?f to ?g. (Level ?f must be exactly one greater than ?g, and may not be zero.)
(!Zoom ?a ?c ?d ?f ?g)	Aircraft ?a flies from city ?c to city ?d, decreasing its fuel level from ?f to ?g. (Level ?f must be exactly two greater than ?g, and may not be zero or one.)
(!Refuel ?a ?f ?g)	Aircraft ?a has its fuel level increased from ?f to ?g. (Level ?f must be exactly one less than ?g, and may not be the maximum fuel level.)

Figure 6.3: Operators for the ZENO domain

The only goal in the ZENO domain is to deliver a passenger to a specific city, so the problem size is the number of passengers to be transported. These experiments use a single annotated task, shown in Figure 6.4. Like in the LOGISTICS domain, HTN-MAKER will not learn methods that would move a passenger unless that passenger were the subject of the current task, so any serialization of these tasks will be safe.

6.1.4 Satellite

The fourth domain is SATELLITE, which models the problem of an array of satellites, each with one or more instruments on them, each of which supports one or more modes. The objective is to collect a variety of images, each of a specific mode and direction. It was first introduced in the third international planning competition (IPC-3) in 2002. The operators in this domain are too complex to show formally on a single page; descriptions of each can be found in Figure 6.5.

```
( :annotated-task
  :head
  (!Travel ?p - person
    ?c - city)
  :precondition
  {}
  :positive-effects
  { (person-at ?p ?c) }
)
```

Figure 6.4: An annotated task for the ZENO domain

Operator	Description
(!TurnTo ?s ?o ?n)	Turn satellite ?s from facing direction ?o to facing direction ?n
(!SwitchOn ?i ?s)	Provide power to instrument ?i on satellite ?s. (No other instrument on that satellite may simultaneously have power.)
(!SwitchOff ?i ?s)	Remove power from instrument ?i on satellite ?s.
(!Calibrate ?i ?s ?d)	Calibrate instrument ?i on satellite ?s, which must be pointed at ?d. An instrument must be calibrated each time that it is powered on before it may be used, and each instrument has a predetermined direction that it must be facing for calibration.
(!TakeImage ?s ?d ?i ?m)	Use instrument ?i on satellite ?s to take an image of direction ?d using mode ?m. The instrument ?i must support the mode ?m and be powered and calibrated. The satellite ?s must be facing direction ?d.

Figure 6.5: Operators for the SATELLITE domain

There is only one type of goal in this domain, only one annotated task is needed. It is shown in Figure 6.6. The metric for the difficulty of problems in this domain is the number of images to take. It is never possible in this domain for an image that had been collected to be lost, so any serialization of goals into a sequence of these tasks will be equivalent to the classical planning problem with all of the goals.

```
( :annotated-task
  :head
  (!GetImage ?m - mode
    ?d - direction)
  :precondition
  {}
  :positive-effects
  { (have-image ?d ?m) }
)
```

Figure 6.6: An annotated task for the SATELLITE domain

6.1.5 Rovers

The most complex deterministic domain used in these experiments is ROVERS, which combines ideas from LOGISTICS and SATELLITE. In this domain, robots on the surface of Mars must traverse the terrain, taking rock and soil samples and astronomical images, and relay their findings to a central lander. What makes this domain much more complex than LOGISTICS is that the graph of locations is not fully connected. Instead, the planner needs to perform path-finding to discover a route from the waypoint a rover is currently in to the one it needs to be in to take a sample or send a result. The ROVERS domain was also introduced at the third international planning competition (IPC-3) in 2002. Figure 6.7 contains a high-level description of the operators for this domain.

There are three similar types of goals in the ROVERS domain: having successfully communicated each of the three types of data to the lander. Thus, there is an annotated task for each type of data, as shown in Figure 6.8. As in the SATELLITE

Operator	Description
(!Navigate ?r ?y ?z)	Move rover ?r from waypoint ?y to waypoint ?z. The latter waypoint must be both visible and traversable from the former.
(!SampleSoil ?r ?s ?p)	Rover ?r, which must be at waypoint ?p, collects a sample of the soil at waypoint ?p and places it in store ?s, which must be empty and be a component of rover ?r.
(!SampleRock ?r ?s ?p)	Rover ?r, which must be at waypoint ?p, collects a sample of the rocks at waypoint ?p and places it in store ?s, which must be empty and be a component of rover ?r.
(!Drop ?r ?s)	Rover ?r disposes of the contents of store ?s, which must be a component of it.
(!Calibrate ?r ?i ?t ?w)	Instrument ?i on rover ?r calibrates itself by pointing at objective ?t while at waypoint ?w. Objective ?t must be visible from waypoint ?w, and must be the predetermined calibration target for instrument ?i.
(!TakeImage ?r ?p ?o ?i ?m)	Instrument ?i on rover ?r takes an image of objective ?o using mode ?m, while the rover is at waypoint ?p. The instrument must support the mode and be calibrated, and the objective must be visible from the waypoint.
(!SendSoilData ?r ?l ?p ?x ?y)	Rover ?r, which is at waypoint ?x, sends data that it had previously collected about the soil at waypoint ?p to lander ?l, which is at waypoint ?y. Waypoint ?y must be visible from waypoint ?x.
(!SendRockData ?r ?l ?p ?x ?y)	As (!SendSoilData ?r ?l ?p ?x ?y), but for data about rocks.
(!SendImageData ?r ?l ?o ?m ?x ?y)	As (!SendSoilData ?r ?l ?p ?x ?y), but for an image of objective ?o that it had previously taken using mode ?m.

Figure 6.7: Operators for the ROVERS domain

domain, once one of these goals has been accomplished there is no action that can undo it, so any serialization of the tasks will be acceptable. Because of the nature of the problem generator used, the measure of the size of a problem in the domain is not the number of goals directly. Rather, it is the number of waypoints, each of which has a $\frac{1}{3}$ probability of having interesting soil and a $\frac{1}{3}$ probability of having interesting rocks.

```
( :annotated-task
  :head
  (GetSoilData
    ?x - waypoint)
  :precondition
  {}
  :positive-effects
  { (comm_soil_data ?x) }
)

( :annotated-task
  :head
  (GetRockData
    ?x - waypoint)
  :precondition
  {}
  :positive-effects
  { (comm_rock_data ?x) }
)
```

```
( :annotated-task
  :head
  (GetImageData
    ?o - objective ?m - mode)
  :precondition
  {}
  :positive-effects
  { (comm_image_data ?o ?m) }
)
```

Figure 6.8: Annotated tasks in the ROVERS domain

6.1.6 RobotNavigation

Finally, I used the inherently nondeterministic ROBOTNAVIGATION domain [34] when evaluating HTN-MAKERND, because it is a common benchmark for systems that plan in nondeterministic domains. The ROBOTNAVIGATION domain consists of a building with eight rooms arranged linearly, with seven doors connecting them. Several rooms contain packages that a robot needs to deliver to different locations.

The robot can open doors, more through open doorways, and pick up and put down packages, though it can hold at most one package at a time. Complicating the robot's objective is an independent agent that runs through the building, opening and closing doors at random. Rather than the actions of a separate agent, these openings and closings are modeled as a set of nondeterministic effects for each action that the robot takes. The basic functions of the operators are shown in Figure 6.9, but in addition each operator has optional effects of opening a closed door and vice versa. In order to reduce the size of the search space, the robot may also choose to temporarily focus on a door (meaning that it cares whether that door is open or not), or to unfocus from it. The only type of goal in this domain is for an object to be in a specific room, so there is a single annotated task that accomplishes this. As in domains discussed earlier, HTN-MAKER would never learn a method to move an object that is not mentioned in the current task, so any serialization of goals is possible. As usual, the number of objects to be delivered will be the measure of problem size.

Operator	Description
`(!Pickup ?o ?l)`	The robot, which must be in room `?l` and not holding anything, picks up object `?o`, which also must be in room `?l`.
`(!Putdown ?o ?l)`	The robot, which must be in room `?l` and holding object `?o`, puts object `?o` down in room `?l`.
`(!Move ?r ?s ?d)`	The robot, which must be in room `?r`, moves to room `?s`. Door `?d` must connect rooms `?r` and `?s`, and must be open.
`(!Open ?r ?s ?d)`	The door `?d`, which must connect rooms `?r` and `?s` and be closed, is opened.

Figure 6.9: Operators for the ROBOTNAVIGATION domain

6.2 Learning Rate Experiments

The first set of experiments is designed to measure the rate at which HTN-Maker learns knowledge from examples, and how applicable that knowledge is to novel problems. Theorem 3 states that it is possible to learn a complete set of methods from some finite set of learning examples, but my hypothesis is that a relatively small number of randomly generated examples will be sufficient.

6.2.1 Setup

To test this, I randomly generated 400 problems of low complexity, then began five distinct trials. The size of these problems was between 5 and 10 blocks to reconfigure in Blocks-World, between 1 and 8 packages to deliver in Logistics, between 3 and 8 passengers to transport in Zeno, between 1 and 5 images to collect in Satellite, and between 3 and 6 waypoints that could contain data to collect in Rovers. For each trial, 300 problems were randomly selected as a training set, while the remaining 100 were held out as a test set. Additionally, each trial specified a random ordering among the problems in the training set. By running multiple trials I intended to avoid any bias that would occur from a particularly easy problem being used in the test set, or from a particularly interesting problem appearing early in the ordered training set.

I then used the FastForward classical planner [28] to generate a solution for each of the 400 problems in each domain. For each problem in the training set, the combination of its initial state and the solution to it produced by FastForward formed a learning example. I then ran HTN-Maker on the classical planning domain description, the appropriate annotated tasks, the learning example associated with the first problem in the training set, and an empty set of methods (M_0). The output of HTN-Maker was stored as M_1. Then I ran HTN-Maker on the classical planning domain description, the appropriate annotated tasks, the learning example associated with the second problem in the training set, and M_1, storing the results as M_2. This continued until each of M_1 through M_{300} had been produced.

Each set of methods M_i represents the knowledge learned by HTN-Maker from the first i problems in the training set.

Then I attempted to solve all problems in the test set using M_0, then did so again with M_1, and so forth up to M_{300}. The HTN planner used was HTN-Solver, which is a reimplementation of Shop (which is itself an implementation of Algorithm 2). The HTN-Solver planner is more efficient than Shop for these experiments because it was written in C and optimized with techniques such as a symbol table. It does not support many of the advanced features of Shop such as axioms, numeric computation, lists, or external functions, but none of these are used in the methods learned by HTN-Maker. If HTN-Solver was able to solve the testing problem within 30 minutes using the appropriate M_i, this was reported as a problem that is covered by the method set M_i; otherwise it was reported as unsolvable with those methods.

Because of the several optional features of HTN-Maker described in Section 3.4, there was not a single system to test. Rather, I chose four different configurations of HTN-Maker as being interesting, and repeated this set of experiments for all four configurations. The problems themselves, the partition of problems into training and testing sets, and the ordering of problems within the training set are all the same across different the different configurations of HTN-Maker, so that it is possible to directly compare the results of a specific trial. The four configurations of HTN-Maker that were tested can be characterized along two axes: whether or not subsumption checking was enabled (see Section 3.4.4) and which form of generalization was used (see Section 3.4.5). I did not explicitly require methods to be right-recursive (Definition 48), though the restrictions of Section 3.4.1 caused this to be the case more often than not. For lack of a better organizing principle, I have labeled the four configurations of HTN-Maker with the letters A through D, as shown in Table 6.1.

Configuration	Subsumption	Generalization
A	On	Weak
B	On	Strong
C	Off	Weak
D	Off	Strong

Table 6.1: Configurations of HTN-Maker

6.2.2 Results

This subsection presents the results of these experiments, while the following explains and discusses them. Figure 6.10 shows the results of this experiment in the Blocks-World domain. Although the interesting parts of the data are packed densely into the top-left of the graph, there are several things to notice here. First, regardless of configuration HTN-Maker is able to learn enough information from a single example to solve, on average, more than 40% of the training problems, and only five training examples are needed for 80% coverage. Second, there is no configuration in which all five trials reach 100% coverage, but all come very close. Third, the results in configurations A and C are nearly identical, as are the results in configurations B and D. The choice of generalization strategy has a small but noticeable impact. Interestingly, the configurations that use strong generalization seem to learn most quickly initially (examples 1-12), then are surpassed by those that use weak generalization through a moderate number of examples (12-70), and then regain a very small advantage through a large number of examples (70-300).

The results for the Logistics domain are show in Figure 6.11, and are similar to those from the Blocks-World domain, with two notable differences. In this domain, configurations B and D (those using strong generalization) learn more slowly than configurations A and C throughout, although with sufficient examples they still achieve near-complete coverage of the testing problems. Configurations A and C actually do achieve complete coverage in each of the five trials, and do so after observing at most 62 training examples.

The graph in Figure 6.12 summarizes the results for the Zeno domain. It is

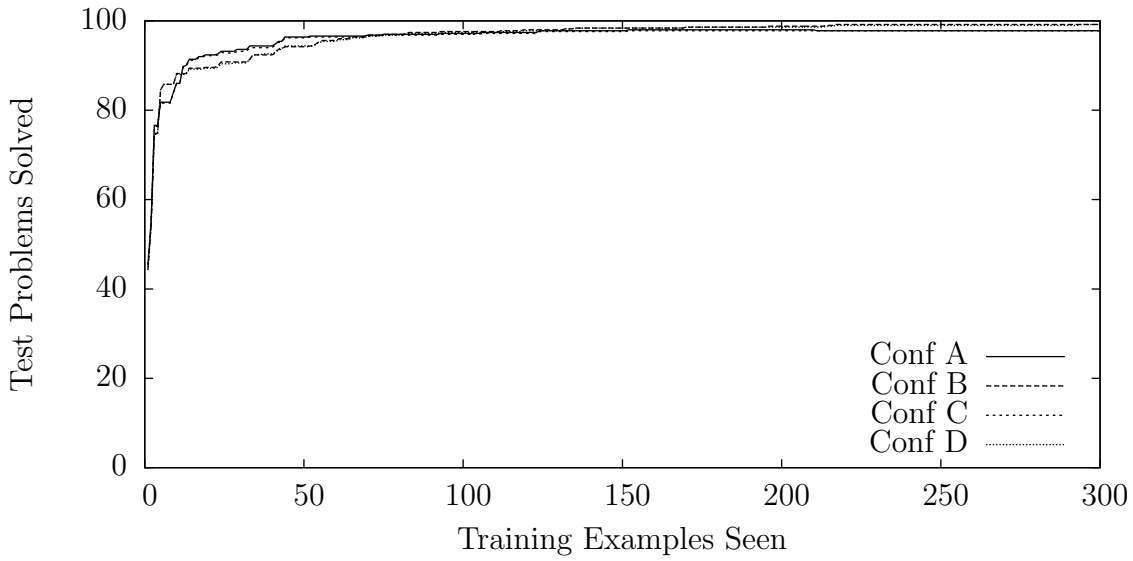

Figure 6.10: Learning rate in BLOCKS-WORLD domain

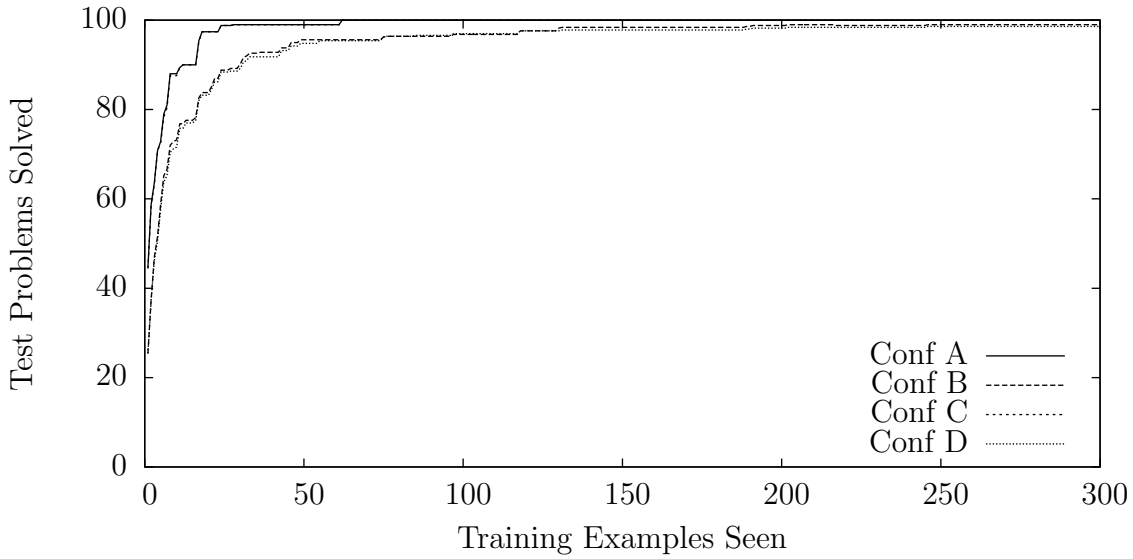

Figure 6.11: Learning rate in LOGISTICS domain

difficult to see much of interest in this graph because HTN-Maker learns so rapidly; in all configurations four learning examples are sufficient to reach 90% coverage of the test set. Configurations A and C reach complete coverage in all five trials after 10 learning examples, while configurations B and D require 18 examples in the worst trial before they produce a set of methods that can solve every test problem.

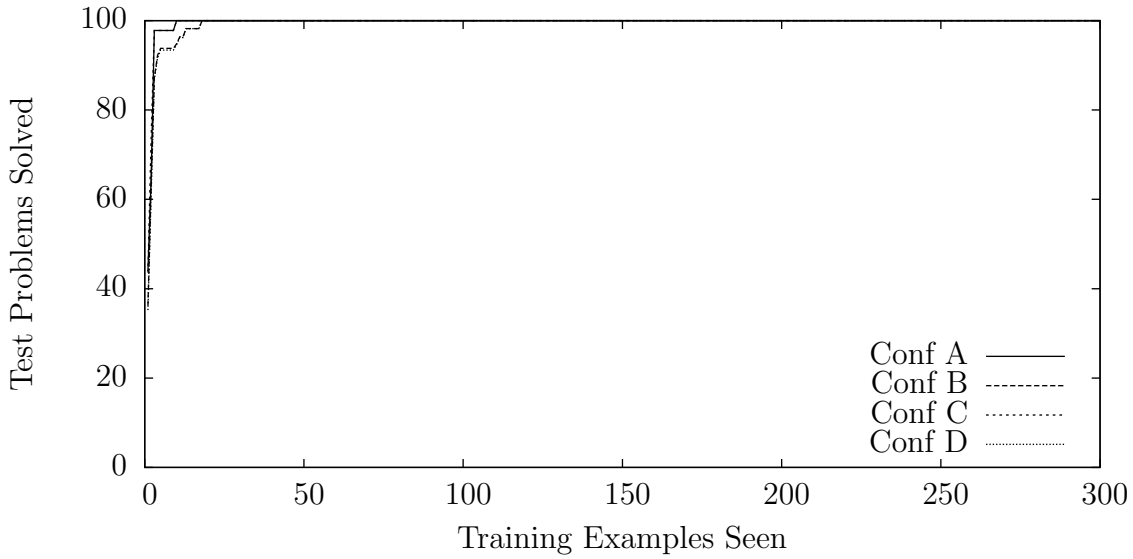

Figure 6.12: Learning rate in Zeno domain

Data from the Satellite domain is shown in Figure 6.13. The results in this domain are similar to Logistics and Zeno, but with one glaring exception. In the Satellite domain configuration A does not perform equivalently to configuration C; instead, it is significantly worse starting after learning example 10. More surprising still, configuration A sees a drop in its performance after learning from training example 182.

Figure 6.14 shows the learning rate data for the Rovers domain. HTN-Maker learns more slowly in this domain than any other test, although it still appears to be slowly converging toward a complete domain. Configurations A and C learn more quickly than B and D.

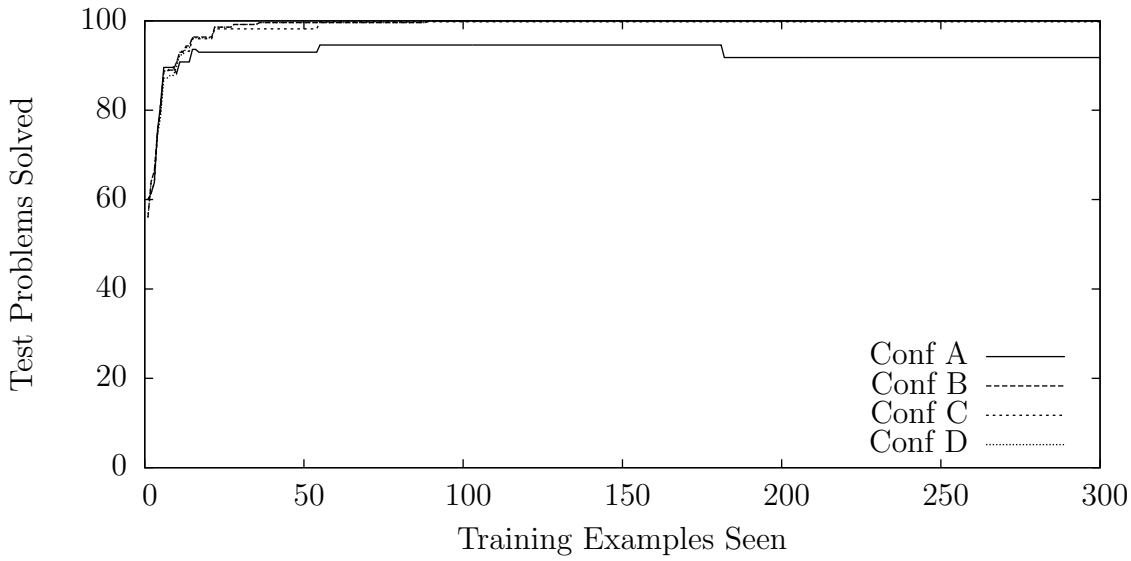

Figure 6.13: Learning rate in SATELLITE domain

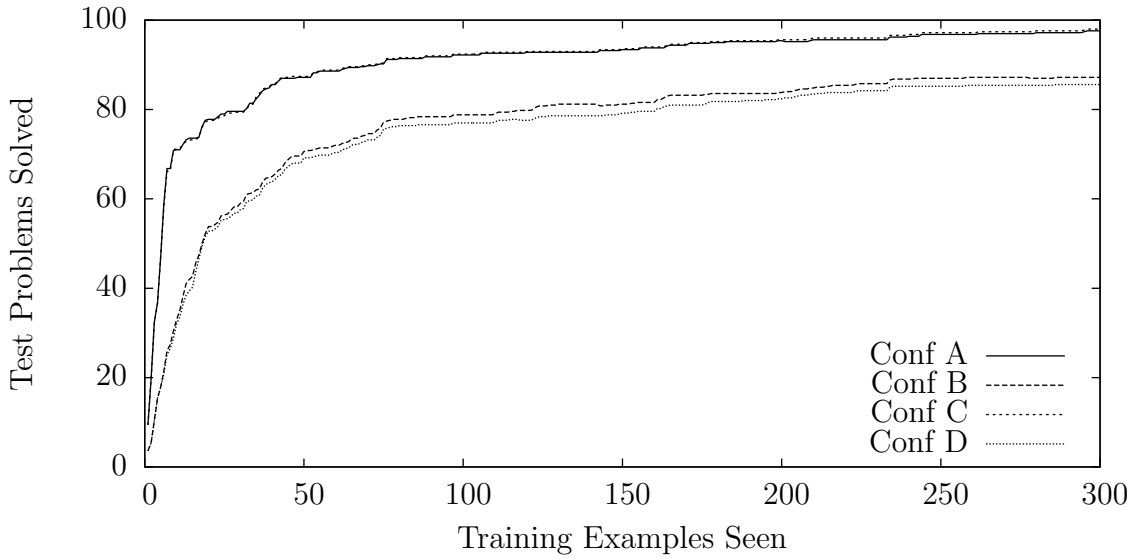

Figure 6.14: Learning rate in ROVERS domain

I also recorded two other pieces of information for each domain and configuration of HTN-Maker: the total number of methods learned from all 300 training examples, and the average time required to learn from a training example. Table 6.2 shows the former, while Table 6.3 has the latter data, both averaged across all five trials.

These experiments used verification tasks and methods, as described in Section 3.4.3, to ensure that plans produced by HTN-Solver as a solution to the HTN equivalent of each testing problem would be solutions to that testing problem, as well as running each plan through a utility program that confirmed that it was a solution. As expected, every plan produced by HTN-Solver was sound with regard to the semantics of the annotated tasks. The system maintained a log of each time that HTN-Solver needed to backtrack during its search process and the reason why. Although a need to backtrack was not uncommon, this was always because the planner solved one of its top-level tasks in a way that resulted in a state from which it did not know how to solve the following top-level task. There was not a single case in which an unsound plan would have been generated if verification tasks had not been used.

Domain	Conf A	Conf B	Conf C	Conf D
Blocks-World	146.0	176.7	1000.6	1056.4
Logistics	42.2	105.6	230.0	829.4
Zeno	11.0	16.2	4211.6	4371.0
Satellite	19.6	24.8	65.8	71.4
Rovers	152.2	419.8	763.6	1957.4

Table 6.2: Average number of methods learned

6.2.3 Analysis

A few broad conclusions may be drawn from this experiment. First and foremost, the methods learned by HTN-Maker, in any configuration and for every domain that was tested, can be used to solve many problems beyond those that formed the

Domain	Conf A	Conf B	Conf C	Conf D
Blocks-World	6.8	15.2	6.4	10.9
Logistics	3.2	13.3	3.3	13.4
Zeno	0.06	0.07	1.61	1.57
Satellite	1.66	0.262	0.262	0.225
Rovers	23.7	116.0	21.7	116.0

Table 6.3: Average number of seconds to learn from one example

examples on which HTN-Maker was trained. In simple domains such as Blocks-World, Logistics, Zeno, and Satellite it is easy to reach complete or near complete coverage of the domain after learning from very few examples. This is less true in a much more difficult domain, such as Rovers.

Most planning problems in each testbed domain can be solved using one of a few common strategies, which is why coverage frequently reaches 80% or higher after learning from a handful of problems. There are, however, a number of special "corner" cases in each domain that will not be solvable until they have been observed and analyzed by HTN-Maker. In the Logistics domain, for example, every package that needs to be delivered falls into one of the cases shown in Table 6.4. Because several of these cases can resolve into one of several other cases depending on the exact situation, there are in total 21 different complete situations. Cases 1, 2, and 3 each account for one situation each, cases 4 and 5 for three each, and cases 6 and 7 for six each. It is possible for HTN-Maker to learn to solve several of these situations by observing only one, but not always. The Rovers domain has many more such corner cases to learn.

In most cases the use of subsumption checking has no effect on the learning rate. This is not surprising, since the only effect of subsumption checking is to remove methods that are provably unnecessary. I did expect to see that the winnowing of methods provided by subsumption checking would speed up planning times, and that perhaps this would be sufficient to make some problems cross the threshold of being solvable in less than 30 minutes, but this does not appear to have occurred in any significant way. (Section 6.3 explores in detail the question of how fast planning

1. The package is already in its destination: nothing needs to be done.
2. The package is in a location in the same city as its destination, with a truck: load into the truck, drive it to the destination, unload it.
3. The package is in a location in the same city as its destination, without a truck: drive a truck to the location with the package, follow #2.
4. The package is in an airport in the wrong city, with an airplane: load into the airplane, fly it to an airport in the proper city, unload it, follow #1, #2, or #3.
5. The package is in an airport in the wrong city, without an airplane: fly an airplane to the airport with the package, follow #4.
6. The package is in a non-airport in the wrong city, with a truck: load into the truck, drive it to an airport, unload it, follow #4 or #5.
7. The package is in a non-airport in the wrong city, without a truck: drive a truck to the location with the package, follow #6.

Table 6.4: Cases for delivering a package in the LOGISTICS domain

is with methods learned from various configurations of HTN-MAKER.)

In all domains the use of subsumption checking significantly decreases the number of methods retained, in the most extreme case (the ZENO domain, with weak generalization) by a factor of 383 and in the least extreme case by a factor of three. This is exactly what I had hoped for and expected when devising the subsumption checking routine, since a smaller set of methods that can solve the same problems as a larger set of methods should be both easier for a human reader to understand and faster for a planner to use. The effect is less pronounced when strong generalization is used than with weak generalization, except in the LOGISTICS domain. This also matches expectations, since strong generalization produces methods that are applicable in fewer circumstances, and thus less capable of subsuming other methods.

In some cases the use of subsumption checking increased the average amount of time required to learn from an example, while in others it had the opposite effect. In most domains the difference is small enough to be within the margin of error, but in the SATELLITE domain when using weak generalization subsumption checking increases the time to learn by a factor of 6 and in the ZENO domain it decreases the time to learn by a factor of 27 (weak generalization) or 22 (strong generalization). There are several ways in which subsumption checking could affect learning times, and apparently different factors are more important in different circumstances. Given two methods, checking whether or not the first subsumes the second or vice versa is often a more expensive operation than checking whether or not the two are equivalent. Thus, I would expect subsumption to increase learning time. However, the use of subsumption decreases the number of methods learned, which means that the number of subsumption checks that need to be performed when it is in use will be less than the number of equality checks that need to be performed when it is not. Indeed, the ZENO domain in which subsumption checking sped up learning by a large factor is the same domain in which subsumption checking decreased the number of methods by a large factor.

In most cases HTN-MAKER learns more quickly with weak generalization than with strong generalization. This is what I would expect, since weak generalization

produces methods that are applicable in all situations in which the methods learned with strong generalization are applicable, and often more. In the Blocks-World domain, however, the relationship is more complicated. The downside of weak generalization is the possibility that methods could be overly applicable, and thus be used by a planner in ways that are legal but not helpful. I believe that this is occurring in the region of the Blocks-World domain graph where the configurations with strong generalization outperform those with weak generalization, and that these unhelpful methods are causing the planner to take too long finding a solution.

Strong generalization produces more methods than weak generalization, but has much less of an effect overall than the presence or absence of subsumption checking. This behavior matches my expectations. With weak generalization HTN-Maker might learn methods from three different examples such that one subsumes the others, or perhaps all are equivalent. With strong generalization HTN-Maker would instead learn three methods, all slightly different from each other in the same way that the examples are slightly different from each other and all necessary to achieve full coverage of the domain.

In the Blocks-World, Logistics, and Rovers domains learning with weak generalization is significantly faster than learning with strong generalization. In the Zeno domain, the choice of generalization strategy has no noticeable impact on learning time, but in the Satellite domain learning with weak generalization is slower than learning with strong generalization, by a wide margin when subsumption checking is used. The behavior in everything but the Satellite domain is easily explained: weak generalization means fewer methods, and fewer methods means fewer equivalence / subsumption checks. Configuration A in the Satellite domain is an anomaly, both in this way and as discussed in the following two paragraphs.

In the Satellite domain, the learning rate of configuration A deviates from configuration C after the tenth learning example, unlike all other domains. I have not been able to find an explanation for this unusual behavior.

In configuration A with the Satellite domain there is a clear decrease in the number of solvable testing problems around learning example 190. This would seem

to be a violation of Lemma 5, which states that all of these rates of solvable testing problems should monotonically increase with the number of learning examples processed. I believe that what is happening here is a failure of a certain bias in HTN-Solver, which does not make truly nondeterministic choices. Instead, at each choice point it uses the first applicable method in the order in which they appear its input file. This is usually a very effective heuristic, but it can result in a situation where the planner infinitely loops, taking a series of earlier methods that do not move toward a solution and ignoring later methods that would do so. Thus, the knowledge necessary to solve the problem still exists, but the planner is ignoring it. This could occur in configuration A if it learns a method that subsumes a method that appears early in the file and thus replaces it (at that early location). Because this new method is more general than the old version, it is now applicable at some point in the process of solving a particular problem when it was not in the past. However, it is not actually useful in that situation and instead leads the planner away from the older, effective problem-solving strategy that is encoded below it.

6.3 Planning Speed Experiments

Having determined that it is possible to solve problems using methods learned by HTN-Maker, I decided to study how quickly this could be accomplished. Although researchers have begun considering other factors as well, the primary evaluation of planning systems remains the speed at which they are able to produce solutions to planning problems. Because larger, more complex problems require more computation, faster planners are able to solve larger problems in a reasonable amount of time. One of the principle advantages of HTN planning over classical planning is that it can be orders of magnitude faster with a well-written set of methods. If the methods learned by HTN-Maker are to be useful, they will also need to allow faster planning than classical techniques.

6.3.1 Setup

This experiment required planning problems of varying sizes, the exact sizes being dependent on the particular domain. For each domain and size of problem to be evaluated in that domain, I randomly generated 20 classical planning problems and their HTN pseudo-equivalents. I then attempted to solve each of these problems using seven different planning system configurations, shown in Table 6.5. FastForward [28] was selected "distinguished planner" in the second international planning competition in 2000 and remains a common benchmark. The most recent planning competitions have judged systems on metrics other than planning speed, but an earlier version of SGPlan6 [31] won the first prize in the satisficing, deterministic planning track of the fifth international planning competition (IPC-5) in 2006. The hand-written methods in use were freshly written by the author for this purpose; the learned ones were taken directly from the first trial of the experiments described in Section 6.2, and are thus the result of analyzing 300 small learning examples.

Planner	Domain Formalization	Abbreviation
FastForward	classical	FF
SGPlan6	classical	SGPlan
HTN-Solver	classical + hand-written methods	Hand
HTN-Solver	classical + methods learned in configuration A	Conf A
HTN-Solver	classical + methods learned in configuration B	Conf B
HTN-Solver	classical + methods learned in configuration C	Conf C
HTN-Solver	classical + methods learned in configuration D	Conf D

Table 6.5: Planning systems tested

Each planning system was given one hour of CPU time on a compute node with eight 2.8 GHz Intel Xeon MP processors and up to 4GB of main memory available to it. The amount of time required by each planner to solve each problem was recorded.

Planner	Logistics	Blocks-World	Satellite	Rovers	Zeno
FF	95.6%	94.0%	51.2%	100.0%	94.8%
SGPlan	100.0%	99.0%	100.0%	100.0%	100.0%
Hand	100.0%	100.0%	100.0%	100.0%	100.0%
Conf A	93.6%	99.0%	100.0%	99.8%	99.2%
Conf B	92.8%	97.0%	98.3%	92.6%	100.0%
Conf C	89.2%	99.0%	100.0%	99.8%	100.0%
Conf D	88.1%	94.0%	98.3%	92.4%	100.0%

Table 6.6: Success rates for each planner in each domain

6.3.2 Results

Not every competitor was able to solve every problem. In some cases this was because the planner was still working after one hour had passed, while in others it was because it had, within the time limit, proven that it lacked knowledge necessary to solve the problem. I have not distinguished between these two failure modes. When comparing different planners, I have included data only about those problems that were solvable by all competitors, and only included those problem sizes for which there were at least 10 such problems. The percentage of problems solved by each planner, for each domain, is shown in Table 6.6. Other than FastForward in the Satellite domain, all did quite well.

Figure 6.15 shows the times for each planner in the Blocks-World domain. The only information readily apparent in this figure is that FastForward scales very poorly compared to the other competitors and that HTN-Solver using the methods learned in configuration D has significant difficulty at the 15-block problem size. (This is due to a single problem that takes exorbitantly long; the others follow the pattern.) To more clearly see how the other competitors perform I have also provided Figure 6.16, which does not include these curves. From this figure we see that the time required by SGPlan6 is growing exponentially with problem size, while the HTN-Solver curves look nearly linear. (These presumably are also exponential, but with a much lower constant.) HTN-Solver is most efficient when using the hand-crafted methods and least efficient when using the methods learned

by HTN-Maker in configuration C. With all sets of methods it is much more efficient than SGPlan6 for large problems.

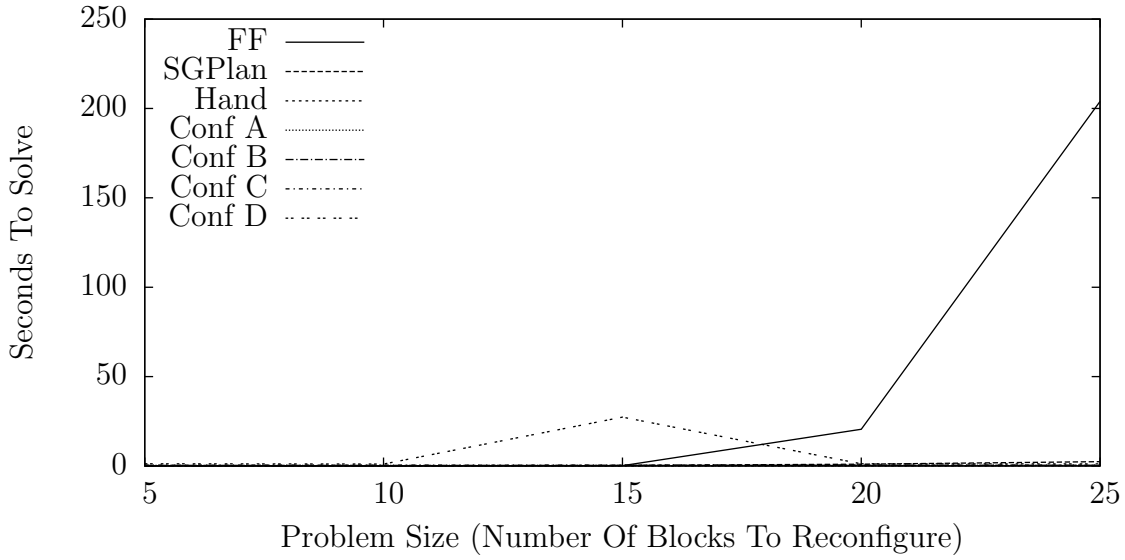

Figure 6.15: Average planning times in Blocks-World domain

Figure 6.17 shows the planning times for all planners in the Logistics domain. The visible results are almost identical to those that Figure 6.15 showed in the Blocks-World domain: FastForward is very slow compared to other planners, and HTN-Solver has a large outlier at one point while planning with methods learned in configuration D. As before, I have created a second graph, shown in Figure 6.18, that excludes these problematic planners. This alternate figure reveals that HTN-Solver also has a few much smaller outliers when planning with the methods learned in configuration A of HTN-Maker. It also shows that as in the Blocks-World domain, the rate of growth of the time required to solve problems with SGPlan6 is much larger than with HTN-Solver, regardless of the methods that it uses. Again, HTN-Solver is most efficient with the hand-crafted methods, and in this case all three sets of learned methods are clustered at requiring approximately twice as long to solve problems as with the hand-crafted methods.

Figure 6.19 shows the planning speeds with all systems in the Zeno domain. As

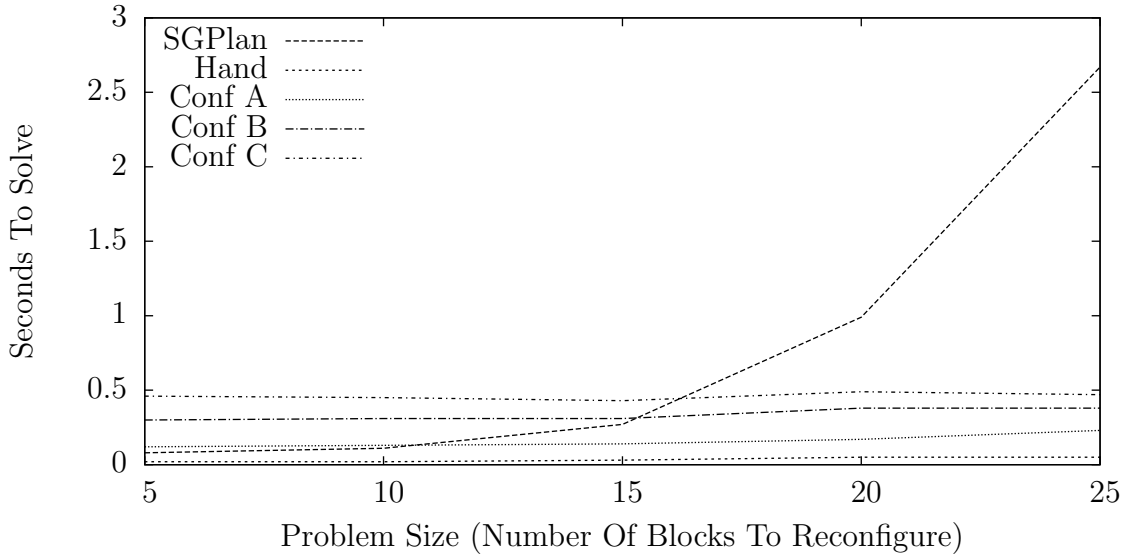

Figure 6.16: Planning times in BLOCKS-WORLD domain, without FF or Conf D

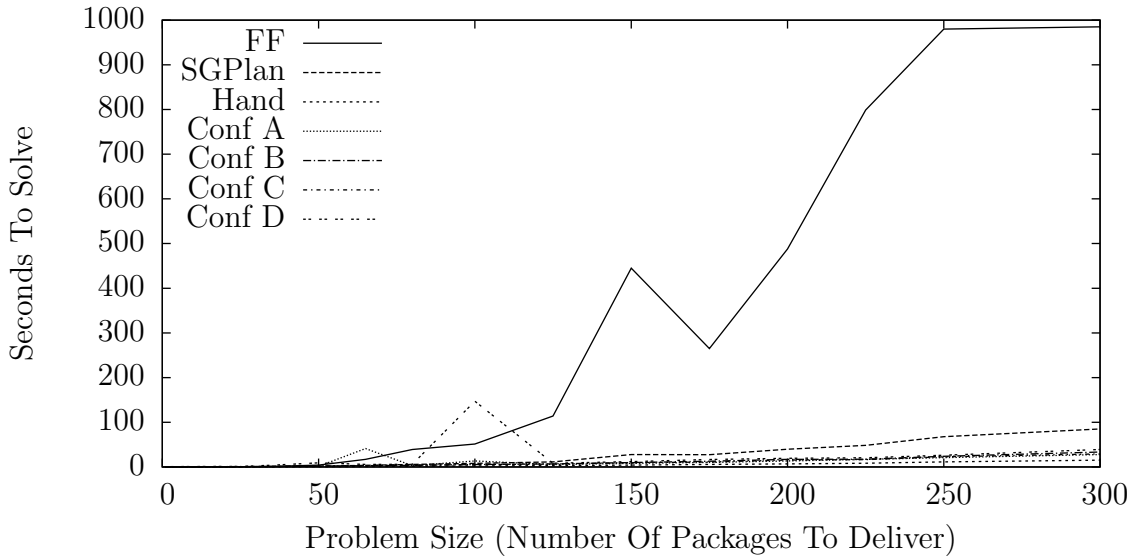

Figure 6.17: Average planning times in LOGISTICS domain

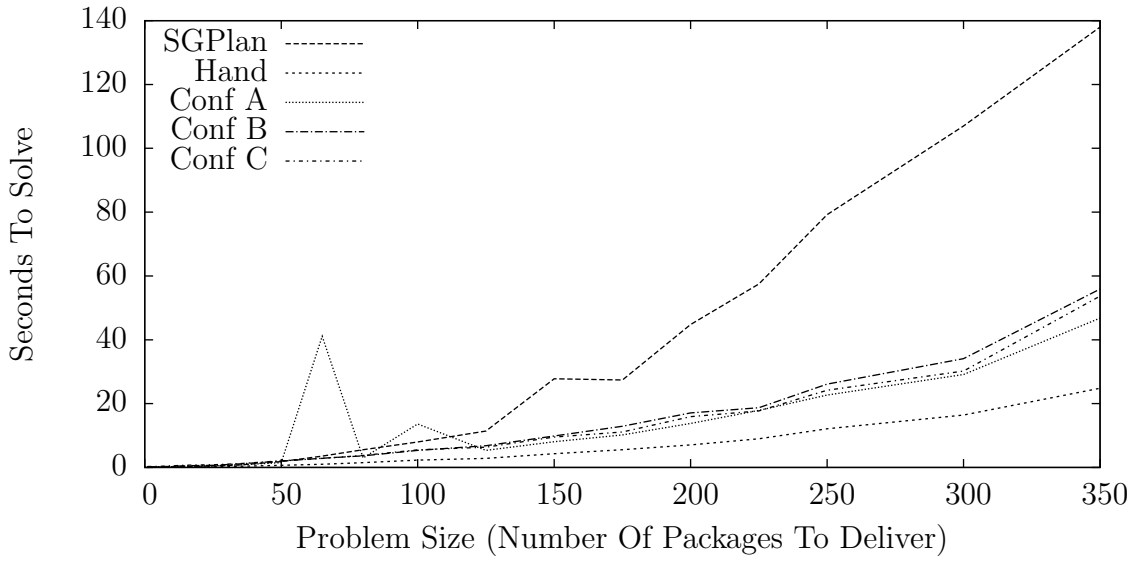

Figure 6.18: Planning times in LOGISTICS domain, without FF or Conf D

in the previous two domains, the times for FASTFORWARD dwarf the other planners. Thus, I have created Figure 6.20 that shows all of the planners other than FASTFORWARD. This figure is rather different from those for the BLOCKS-WORLD and LOGISTICS domains. HTN-SOLVER using the hand-crafted methods is still best, but SGPLAN6 is competitive with HTN-SOLVER using methods learned in configurations B, C, and D, and HTN-SOLVER using methods learned in configuration A performs worst.

Figure 6.21 shows the data for all planners in the SATELLITE domain. As in the ZENO domain, this reveals only that FASTFORWARD is incomparably slower than the other planners. Figure 6.22 shows the same data without FASTFORWARD. This is a similar result to the LOGISTICS domain. That is, HTN-SOLVER using hand-crafted methods is fastest, while HTN-SOLVER with any of the sets of learned methods takes approximately twice as long, and SGPLAN6 is significantly slower on large problems.

Figure 6.23 shows the times for all planners in the ROVERS domain. These results are quite different from those found in the other domains. Most significantly,

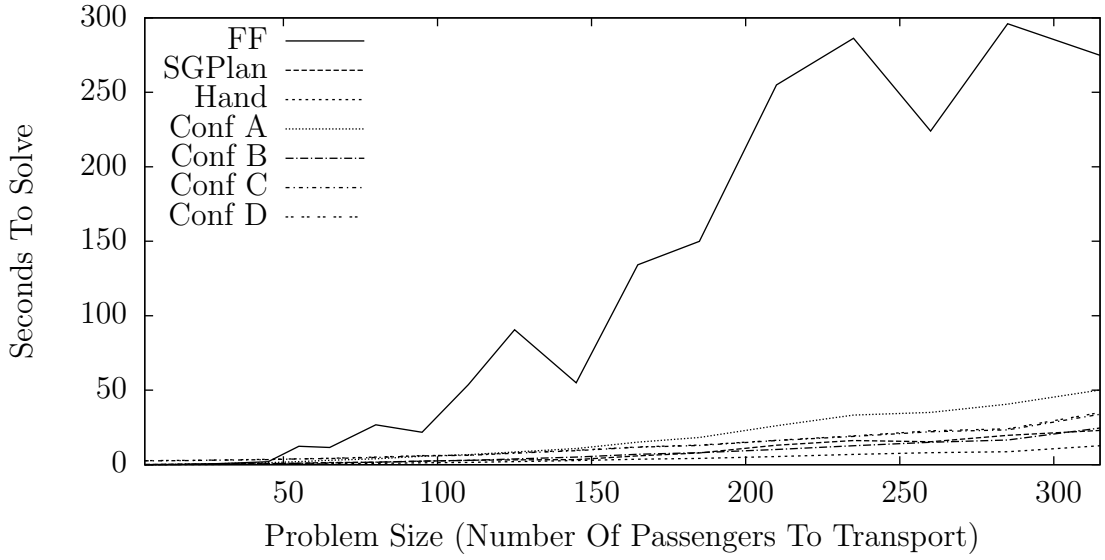

Figure 6.19: Average planning times in ZENO domain

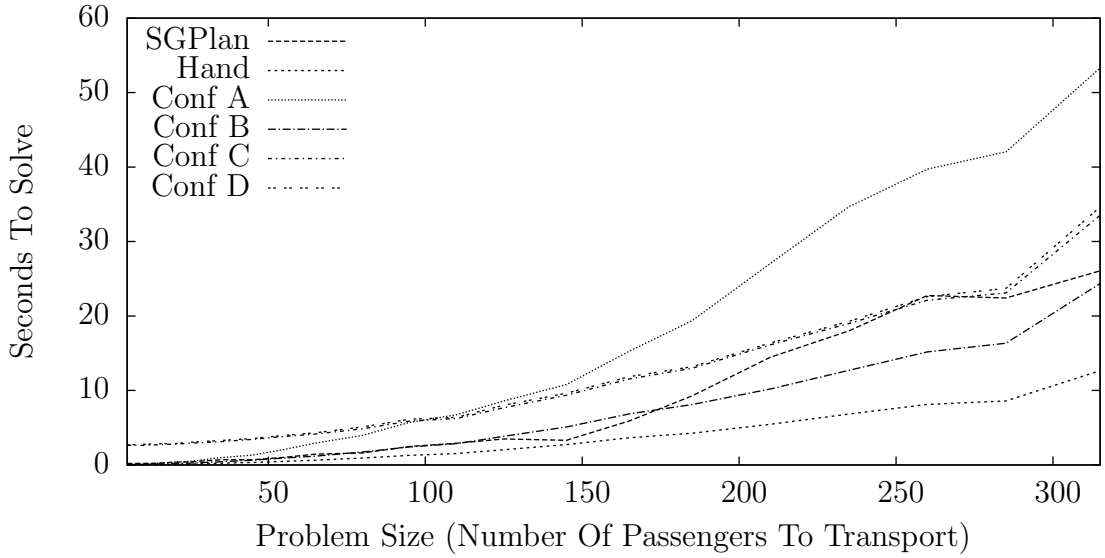

Figure 6.20: Planning times in ZENO domain, without FF

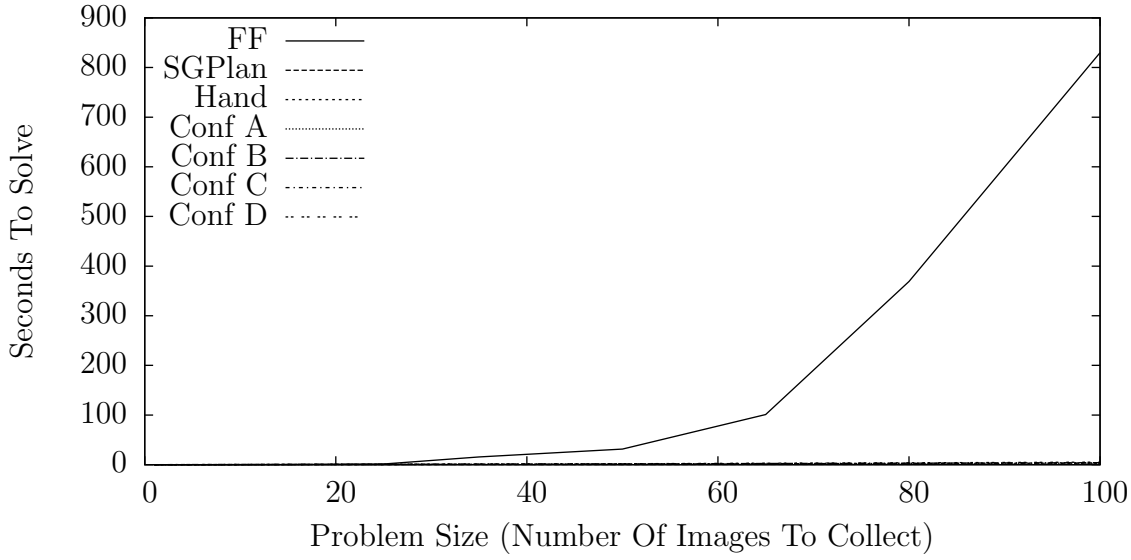

Figure 6.21: Average planning times in SATELLITE domain

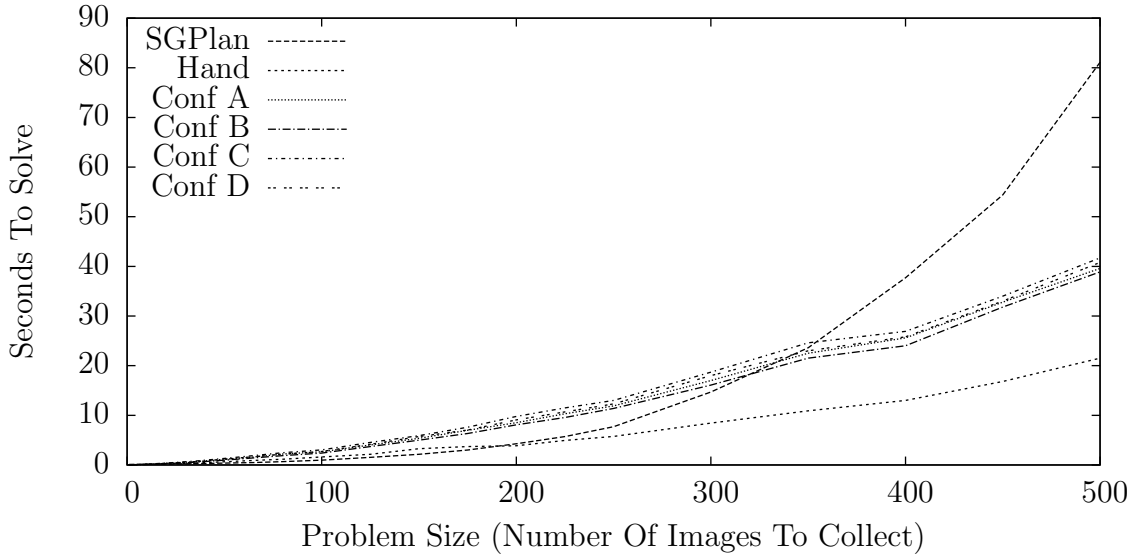

Figure 6.22: Planning times in SATELLITE domain, without FF

FastForward is able to solve problems in the Rovers domain very quickly. It, SGPlan6, and HTN-Solver with the hand-crafted methods are all fast enough to essentially lie along the horizontal axis on this graph. With the methods learned by HTN-Maker, regardless of configuration, HTN-Solver performs comparatively poorly. Among the different sets of learned methods, it is most efficient when using those learned in configuration A and least efficient with those learned in configuration D.

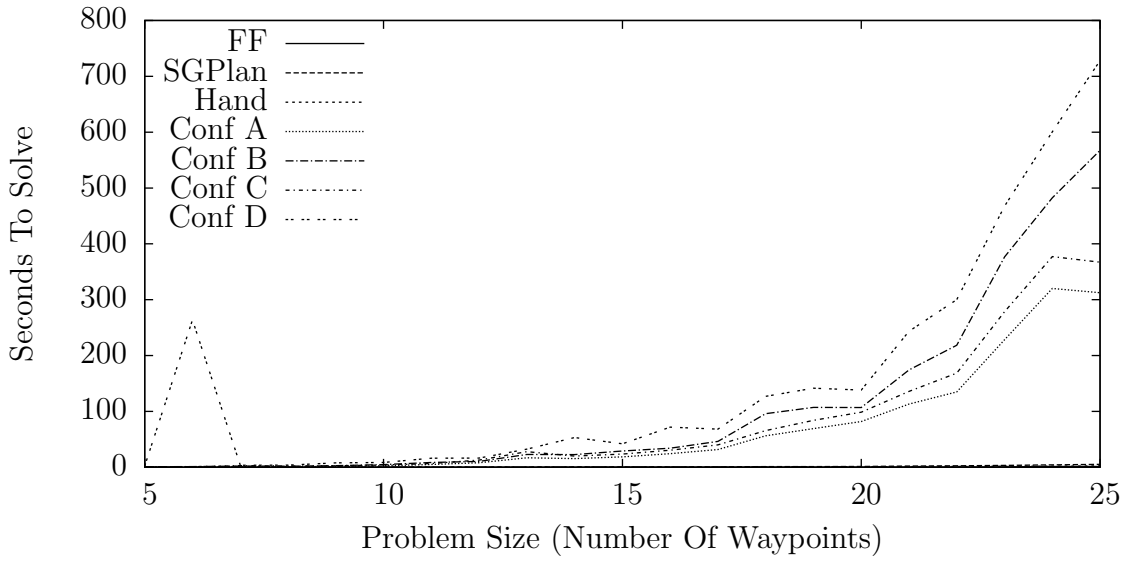

Figure 6.23: Average planning times in Rovers domain

6.3.3 Analysis

In the four relatively easy domains, FastForward is no longer competitive with a more recent planner such as SGPlan6, and is so uncompetitive as to make comparisons with HTN-Solver unnecessary. I have no explanation for the remarkable effectiveness of FastForward in the Rovers domain, where it is actually slightly faster than SGPlan6. As would be expected, HTN-Solver with methods that

have been carefully hand-crafted by a domain expert performs much better than either of the classical planners in four of the domains, and equally well in the remaining one. When using methods that were learned by HTN-Maker, HTN-Solver is not as efficient as when using the hand-crafted methods, but it is still far better than either classical planner in all but the Rovers domain.

It is important to note that HTN-Solver has a significant advantage over the classical planners: it may only consider one serialization of the goals, which is the one that was used when creating a task network from a set of goals. The classical planners are free to consider any serialization, and in fact do not know that a particular serialization will work until it tries it. This feature may be as important to reducing the search space that HTN-Solver sees as the use of methods is.

This same feature, however, can be a significant disadvantage in certain circumstances, and I believe this is why HTN-Solver is incapable of outperforming the classical planners in the Rovers domain. For example, with the chosen serialization of goals into tasks, a rover might need to navigate a series of waypoints to reach a certain location from which it needs a rock sample, take that sample, then navigate to another waypoint from which it can contact the lander module (thus completing a task), then navigate back to where it had taken the rock sample, take a soil sample, navigate back to the waypoint from which it can contact the lander (thus completing a second task), then return to another waypoint that it has passed through four times already to gather another sample, and so forth. The classical planners, which can interleave actions taken to accomplish various goals, will be able to produce much shorter plans. When writing the hand-crafted methods for HTN-Solver I was able to somewhat compensate for this disadvantage by adding additional tasks and using them judiciously to simplify and compact the methods, but to do this automatically would be quite difficult.

The various configurations of HTN-Maker do make a difference in planning times, but not a very significant nor consistent one. Configuration A usually performs best among them, but in Zeno the opposite is true. The methods from configuration D give HTN-Solver an extraordinarily difficult time with a handful of problems, but perform comparably with the other configurations everywhere

else. Without a clear winner, I cannot make a strong recommendation to use or avoid subsumption checking or to prefer one generalization strategy over another. Taking into account the results of the learning rate experiments I can make a weak recommendation to try configuration A (with subsumption checking on and weak generalization) first, but to recognize that depending on the features of the domain any of the others may perform better.

6.4 Nondeterministic Domains

This section describes an evaluation of the effectiveness HTN-MAKERND in learning methods for nondeterministic planning domains.

6.4.1 Setup

The way of evaluating HTN-MAKERND is quite similar to how HTN-MAKER was evaluated, but it uses only two domains: the nondeterministic version of BLOCKS-WORLD and ROBOTNAVIGATION. First, I randomly generated a number of planning problems (500 for ROBOTNAVIGATION, each with a single object to deliver, and 1000 for BLOCKS-WORLD, each with eight blocks to reconfigure). I then used ND-SHOP2 with a set of hand-crafted methods to produce a strong-cyclic solution to each of these problems. (Weak solutions are not very interesting, and strong solutions are not generally possible for these problems). I then simulated execution of these solution policies on their respective problems, producing 50 execution traces from each ROBOTNAVIGATION problem and 100 execution traces from each BLOCKS-WORLD problem. An execution trace is the plan produced by following a particular path through the execution structure of a policy. Thus, this produced 25,000 learning examples in the ROBOTNAVIGATION domain and 100,000 learning examples in the BLOCKS-WORLD domain, although many of them are very similar to each other. Because of the branching nature of nondeterministic planning, many more examples are needed in order to learn to correctly handle all possible outcomes.

I then pre-processed the planning domains and execution traces as described in

Section 4.1 by partitioning the nondeterministic operators into a set of deterministic operators and replacing each nondeterministic action in the execution traces with the determinized version representing which effects actually occurred. Although HTN-MAKERND is usable in all of the configurations that HTN-MAKER is, for all of these experiments it used subsumption checking and strong generalization. Then I ran HTN-MAKERND on these learning examples, collected the learned methods, and post-processed them to restore the nondeterminism.

I then generated testing problems. In the ROBOTNAVIGATION domain this consisted of 25 problems with a single object, 25 problems with two objects, and so forth for three, four, and five objects. In the BLOCKS-WORLD domain this consisted of 50 problems with three blocks, 50 problems with four blocks, and so forth for five, six, seven, and eight blocks. I recorded the time required to solve these problems with two systems: ND-SHOP2 with the methods that had been learned by HTN-MAKERND and MBP with the classical domain description. Both ND-SHOP2 and MBP were described in Section 2.3.2, and MBP is the benchmark against with ND-SHOP2 has been evaluated in prior publications. Each planner was given one hour of CPU time to solve each problem in a virtual machine with a 2.16GHz Intel Core Duo processor and 512MB of main memory, and as before I have not distinguished between failures due to time constraints and failures due to insufficient knowledge.

6.4.2 Results

When there are 100,000 learning examples, testing the ability to solve problems after each one becomes quite tedious. Thus, I have only studied the learning rate in the nondeterministic BLOCKS-WORLD domain, and only for the first 5,000 learning examples. Figure 6.24 shows this data. The test problems used here are the same ones that were generated for evaluating the speed of solving problems with the learned methods, and thus there are 50 problems of each of several sizes. The curve labeled "3-block" represents the percentage of the 50 problems with 3 blocks in them that could be solved after learning from a certain number of examples, and so forth.

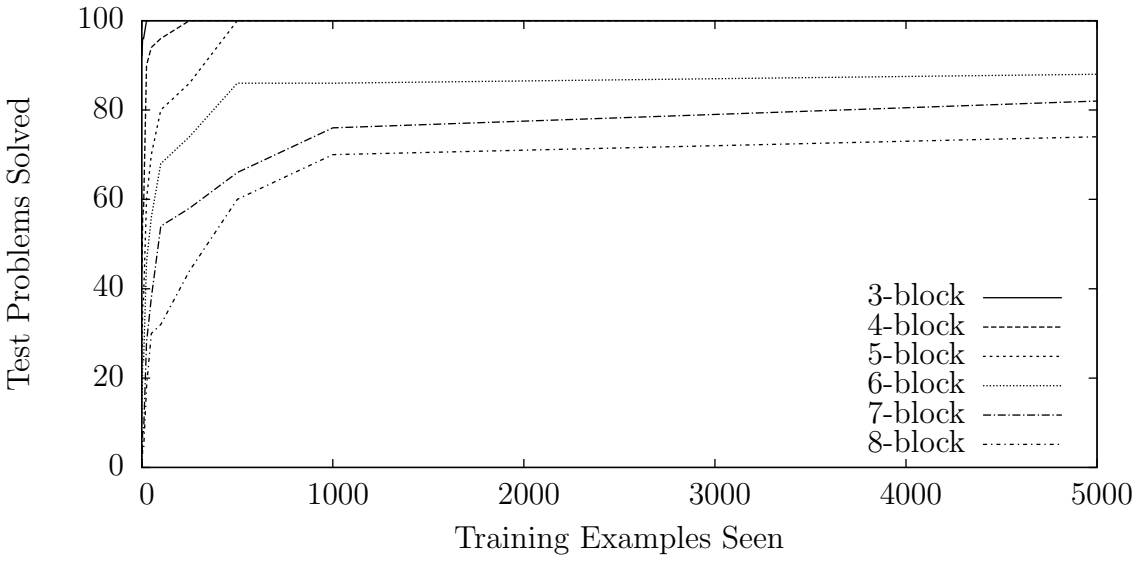

Figure 6.24: Learning rate in nondeterministic BLOCKS-WORLD domain

Planning speed results in the BLOCKS-WORLD domain are shown in Figure 6.25. ND-SHOP2 is able to solve problems of any tested size very quickly. MBP is able to solve problems with 3-6 blocks quickly, but requires much more time for 7-block problems and was unable to solve 39/50 8-block problems within the time limit. ND-SHOP2 solved all of the 3-block, 4-block, and 5-block problems, 45/50 6-block problems, 43/50 7-block problems, and 38/50 8-block problems. Times for problems that could not be solved by one of the planners are not included in the averages for either planner. In those cases where ND-SHOP2 was unable to solve a problem it was able to determine that this would be the case very quickly, averaging 0.04 seconds for the unsolvable 6-block and 7-block problems and 0.03 seconds for the 8-block problems.

Planning speed results in the ROBOTNAVIGATION domain are shown in Figure 6.26. Again, when ND-SHOP2 is able to solve a problem it does so very quickly. HTN-SOLVER failed to solve 1/25 1-object problems, 2/25 2-object problems, 2/25 3-object problems, 2/25 4-object problems, and 6/25 5-object problems. As before, these problems are excluded from the graph. In this domain determining that no

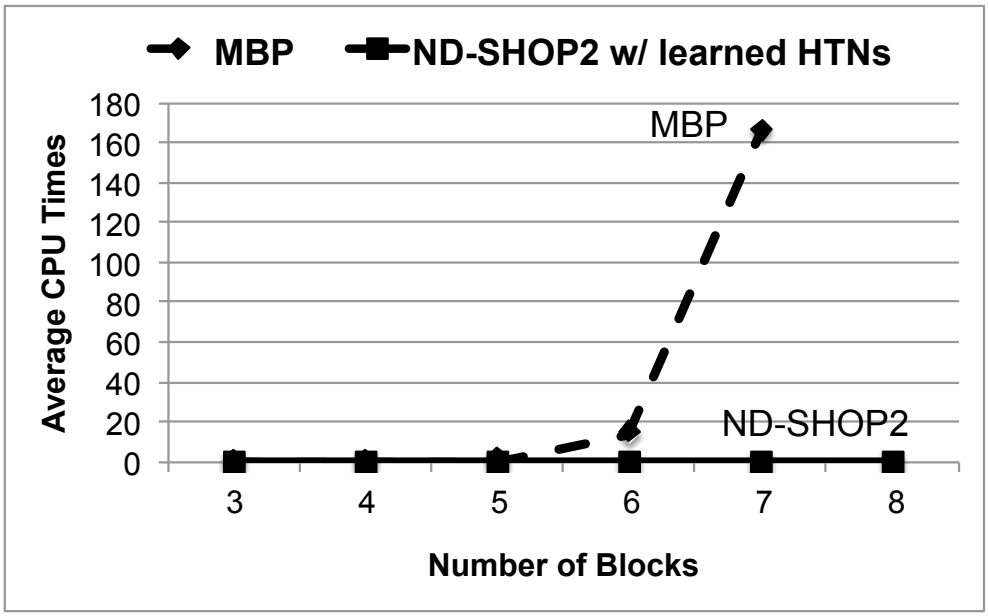

Figure 6.25: Average planning times in nondeterministic BLOCKS-WORLD domain

strong-cyclic solution existed with the provided methods required significantly more time, averaging 5.36 seconds for the 1-object problem, 12.2 seconds for the 2-object problems, 80.8 seconds for the 3-object problems, 200 seconds for the 4-object problems, and 35.6 seconds for the 5-object problems. MBP was again able to solve small problems quickly, but was much slower on 4-object problems and was only able to solve 3/25 5-object problems within the time limit.

6.4.3 Analysis

In spite of the nondeterminism, HTN-MAKERND learns very quickly to solve simple problems in the nondeterministic version of the BLOCKS-WORLD domain. The variety of possible goals in a problem with only three or four blocks is rather small, which explains why only a few examples are needed. As the number of blocks increases, however, the number of unique problems increases exponentially; there are more than 150 billion distinct 8-block problems. Although the learning rate for 6-block, 7-block, and 8-block problems levels off dramatically after the first 1000

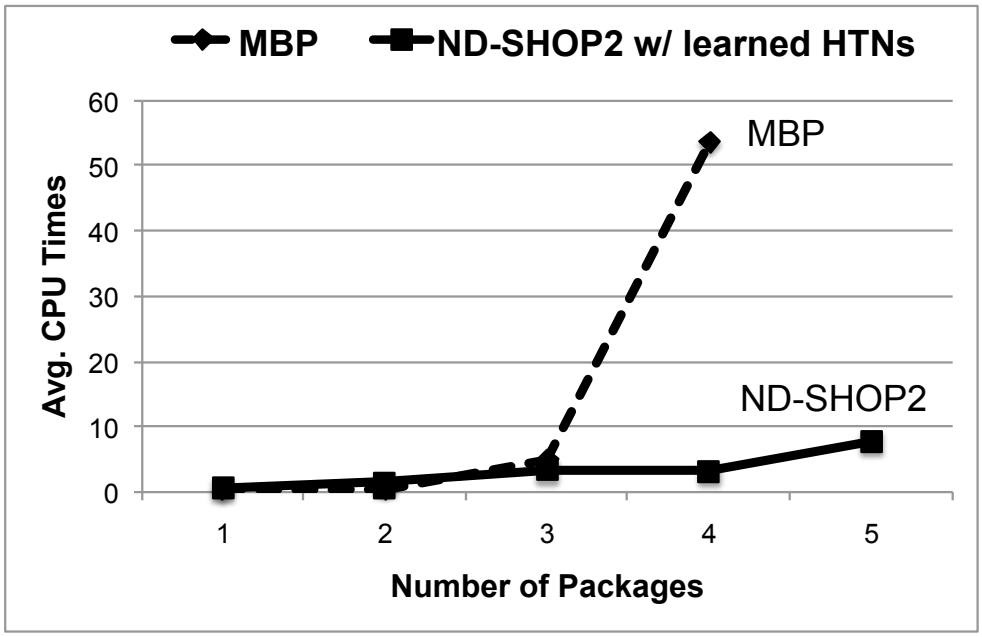

Figure 6.26: Average planning times in RobotNavigation domain

or so traces, the coverage does slowly continue to grow. The speed of planning with HTN-Solver using the learned methods scales much better with problem size than with MBP. Some of the test problems require knowledge that was not demonstrated in the learning examples, but it appears that MBP will be unable to solve any problems of moderate size at all, while HTN-Solver will be able to solve many of them, and do so quickly.

6.5 Method Values

Finally, I have studied the effectiveness of the combination of Q-Maker and Q-Reinforce to learn methods with cost estimates that will allow Q-Shop to quickly find high-quality plans. Most planning systems are designed with one of the following objectives in mind, or perhaps can do either depending on configuration: to find any solution as quickly as possible, or to find an optimal solution as quickly as possible. The former is a *satisficing* planner, while the latter is an *optimal* planner. Finding

an optimal solution to a planning problem is usually very computationally expensive, and thus optimal planners can only be used to solve much smaller problems than satisficing planners. The goal for Q-SHOP is a bit of a middle ground: finding solutions that are reasonably good (though not necessarily optimal) while using no more time and resources than a satisficing planner.

6.5.1 Setup

I decided to use the most straightforward measure of plan quality: shorter plans are better. Thus, the reward received by the reinforcement learning agent when reducing a task t with a method m is the number of primitive subtasks of m multiplied by -1. Thus, when the agent maximizes its returns it will minimize the length of the plans that it generates.

Two domains were used for this experiment: BLOCKS-WORLD (the deterministic version) and SATELLITE. For each of those domains, I randomly generated 600 training problems and solved them using FASTFORWARD [28]. These problems and solutions formed 600 learning examples, which I processed with Q-MAKER to produce a set of methods with estimated method values. Then I randomly generated an additional 600 tuning problems. I used Q-REINFORCE to solve these tuning problems using the methods that were learned by Q-MAKER and updating the value estimates for those methods.

I then randomly generated a third set of problems, 20 each of several sizes, as a testing set. I attempted to solve these testing problems using several different planners, shown in Table 6.7. FASTFORWARD and SGPLAN6 are satisficing planners as used in the earlier experiments, while HSP^*_F [38, 26] is an optimal planner that was the runner-up in the sequential optimization track of the sixth international planning competition (IPC-6) in 2008. Each planner was allowed a maximum of 30 minutes of CPU time for each problem.

Planner	Domain Formalization	Abbreviation
FastForward	classical	FF
SGPlan6	classical	SGPlan
Hsp^*_F	classical	HSP
HTN-Solver	classical + methods without values	NoValues
Q-Shop	classical + methods with unrefined values	Phase1
Q-Shop	classical + methods with refined values	Phase1+2

Table 6.7: Planning systems tested

6.5.2 Results

FastForward, SGPlan6, HTN-Solver, and Q-Shop with both sets of methods were able to solve all of the test problems in less than a second, but this was not true of the optimal planner Hsp^*_F. In fact, for problems of even very modest sizes Hsp^*_F was incapable of finding solutions within the 30 minute time limit.

Figure 6.27 shows the average plan quality achieved by each of the satisficing planners in the Blocks-World domain. The data presented in this graph is the average across all problems of a certain size of the ratio of the length of the plan produced by the planner to the length of an optimal plan, multiplied by 100 to read as a percentage. Thus, a data point of 100 means that the planner produced an optimal plan, while a data point of 200 means that the planner produced a plan that was twice as long as it could have been.

Of these planners, Q-Shop using the methods with value estimates that had been both learned and refined consistently produced the highest-quality plans. In fact, in all but three of the 420 test problems it found an optimal plan. Q-Shop using methods with value estimates that had been learned but not refined performed next best, producing plans on average 13.3% longer than optimal. HTN-Solver using methods with no value estimates also performed well (13.6% longer than optimal), as did FastForward (16.1% longer). SGPlan6 produced the lowest-quality plans, averaging 73.7% longer than optimal. None of the planners seemed to be more suboptimal on larger problems. In fact, the HTN planners seem to perform slightly better on larger problems than smaller ones.

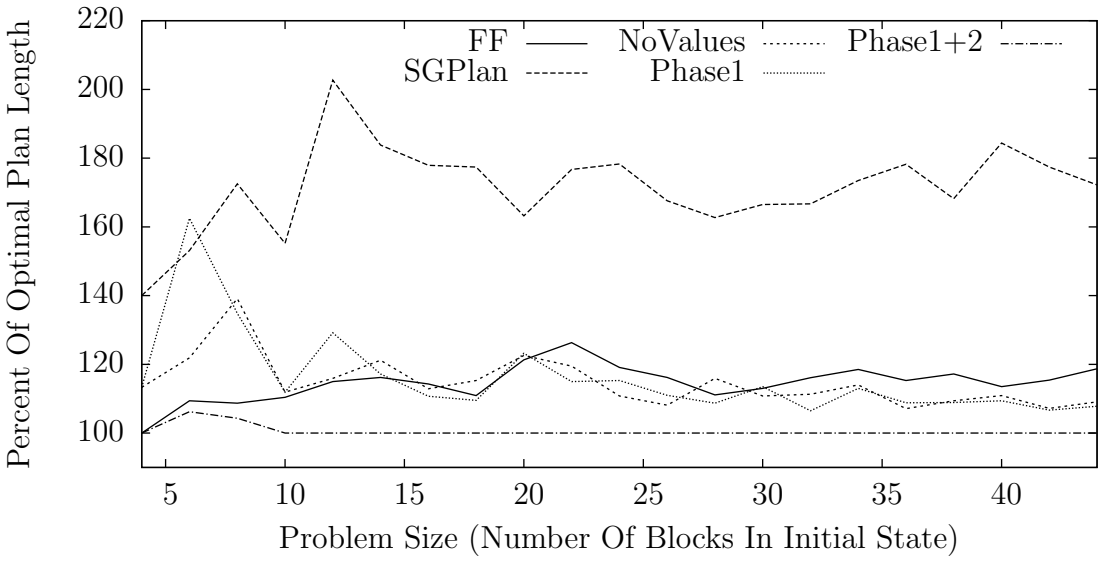

Figure 6.27: Plan quality in BLOCKS-WORLD domain

Figure 6.28 shows the average plan quality achieved by each of the satisficing planners in the SATELLITE domain. Most of the planners did better in this domain, although Q-SHOP continues to produce the highest-quality plans in most circumstances (averaging 7.6% longer than optimal). The refining phase does not appear to alter methods values in way that would change the planner's behavior, as Q-SHOP produced the same plans whether it was using the method value estimates that had been refined or those that were merely learned by Q-MAKER. The worst performer in this domain is HTN-SOLVER with methods that have no value estimates. Each of the planners is trending toward less optimal plans as problem size increases (but see note in Section 6.5.3).

6.5.3 Analysis

Q-SHOP does indeed solve problems at the same order of speed as a satisficing planner (which, in fact, it is). As such, it is far more practical for use where optimality is not a requirement than an optimal planner. The extra work of finding a good

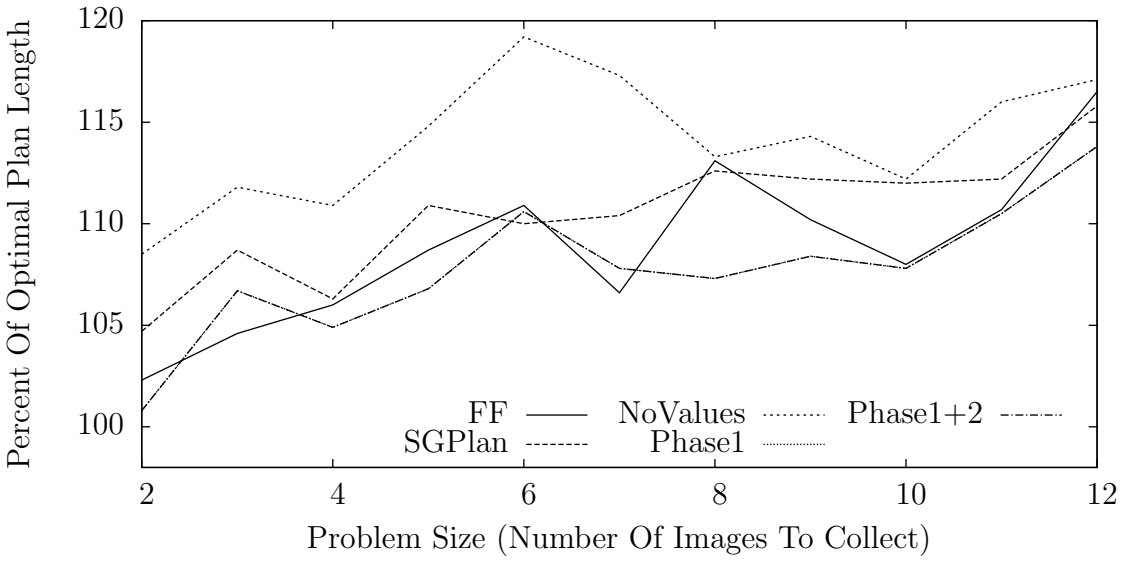

Figure 6.28: Plan quality in SATELLITE domain

plan is offloaded to the learning and refining phases.

While HTN-SOLVER with methods learned by Q-MAKER without any value estimates is already capable of finding solutions of similar quality to classical planners, adding value estimates taken from the learning examples leads to a significant improvement. These value estimates can be further improved through refinement in the BLOCKS-WORLD domain, but apparently not in the SATELLITE domain. This would indicate that the methods learned in the BLOCKS-WORLD domain can be used in ways that did not appear in the training examples, while those learned in the SATELLITE domain can not.

It is surprising that Q-SHOP is able to find optimal plans, and to do so quickly, in a complex domain such as BLOCKS-WORLD. Indeed, the problem of finding a guaranteed optimal solution to a problems in the BLOCKS-WORLD domain is NP-hard, even with a domain-specific algorithm [24]. This difficulty comes from *deadlocks*, when there are a set of blocks, each of which can be moved and needs to be moved so that a different block in the set may be placed in the proper configuration. Deadlocks have multiple ways in which they could be resolved, and there is no faster

way, in general, to determine which resolution will lead to the shortest plan without trying each. Q-SHOP does not guarantee optimality, but did achieve it most of the time in this experiment.

This is because of the way that test problems were generated for this experiment. In other experiments using the BLOCKS-WORLD domain the size of a problem was the number of blocks that exist in that problem, with each block having an explicit initial position and an explicit goal position. In this experiment, the size of a problem remains the number of blocks that exist in that problem, but not all blocks have an explicit goal position. Rather, the goal consists of only a single tower and does not constrain the positions of blocks that are not part of that tower. I chose this because Q-MAKER considers only one task at a time and the method that accomplishes the current task in the fewest number of steps might result in a state that is good for some potential future tasks but bad for others. Thus, more difficult problems will require on average longer solutions, but not much longer. Because there is only one goal tower, there is a deterministic algorithm that will find an optimal solution: for each block in the goal tower, from bottom to top, first move all blocks that are above it to the table, then move it to its goal position. The methods learned by Q-SHOP encode this strategy (as well as others), and in most cases the values learned by Q-SHOP and refined by Q-REINFORCE choose this strategy over other, suboptimal ones.

Chapter 7

Related Work

I have already discussed systems that use knowledge to solve planning problems in Sections 2.1.2, 2.2.2, and 2.3.2. In this chapter I will focus on systems that *learn* knowledge for planning. First, I discuss various work in learning for planning that is not hierarchical in Section 7.1. In Section 7.2 I discuss learning knowledge structures that are equivalent or similar to HTN methods. Section 7.3 discusses a variety of other types of related work.

7.1 Learning For Non-Hierarchical Planning

There is a long history of research integrating automated planning and machine learning [100]. Most of the research using machine learning techniques in the context of planning, at least historically, has been attempts to gain knowledge that would allow classical planners to solve planning problems more quickly.

7.1.1 Case-Based Planning

One of the most obvious ways to speed up planning is to memorize solutions to problems, and when the problem is seen again, use the existing solution rather than performing a computationally expensive search. Because many problems differ from each other only in the names of constants or have extraneous details that are

irrelevant to their solutions, a single plan may be generalized into a version that will be a solution to many different planning problems. The influential STRIPS planning system was extended to do exactly this [17]. Although they do not use the same terminology, the authors consider a generalization strategy equivalent to what I call strong generalization, but reject it in favor of a system equivalent to what I call weak generalization.

The idea of storing and reusing solutions is fundamental to the field of *case-based reasoning* (CBR) [41]. A case-based reasoning system maintains a library of *cases*, which represent previously seen problems and solutions. In order to solve a new problem, a case-based reasoner searches its case library for a a case that is similar to the new problem and then adapts the solution in that case (or set of cases) to solve the new problem. In case-based planning, cases consist of planning problems and solutions [58].

CHEF is one of the first and most important case-based planners [25]. CHEF stores in its case library not only information about solutions found, but also information about planning failures. When a new problem is similar to previous problems that have resulted in failure, additional goals are added to the problem to help avoid those failure modes. Then CHEF searches its case library for solutions to problems with similar goals to the new problem and selects a solution to a problem that shared as many goals as possible with the new problem. The planner then modifies this solution to be a solution to the current problem by adding and removing individual actions, and saves its experience as a new case documenting its success or failure. CHEF has only been used in a single domain, Szechwan cooking, in which it appears the initial state is either always the same or is of little importance in determining the applicability of a plan. It is not clear that the adaptation procedure used by CHEF would be effective in other domains.

The ADJ system, on the other hand, uses domain-independent replanning algorithms [20]. ADJ begins with a plan that is not quite a solution, determines what parts of it should be retained, and repairs the other parts to be relevant to the current problem. It does so by identifying *planning windows*, which represent problematic subplans, and attempting to modify them in a way that has no effect

outside the planning window.

There are HTN and pseudo-HTN planners that utilize case-based reasoning as well. BioPlanner does so in the domain of biological pathways, using domain-specific rules to adapt plans in cases to new problems. Darmok does so in the domain of playing real-time strategy games, by analyzing traces of a human player's decisions to learn *plan snippets* [68]. Traces must be annotated with explanations of why each action was taken. 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.

Other case-based planning systems, such as Prodigy/Analogy, do not attempt to modify and reuse existing plans, but rather store information about why certain decisions were made as cases [88]. When solving new problems and facing a decision with several alternatives, Prodigy/Analogy searches its case library for a time when a similar decision needed to be made and attempts to apply the same reasoning.

7.1.2 Learning Macro-Operators

Instead of retaining entire solutions, a planning system could instead store sequences of only a few actions taken from a plan that could be combined in different ways to solve different types of problems and, during planning, select an entire stored sequence rather than an action. These plan pieces are called *macro operators*, since they give a name to a sequence of traditional operators [43]. In the learning extension to Strips discussed above, plans were stored in such a way that any contiguous subplan could be reused rather than the entire plan, making each plan a set of several macro operators [17]. The idea of macro operators is that if a sequence of three operators is frequently used, a new pseudo-operator could be created that has the combined effects of the three operators. When planning, the system could

select this macro operator instead of any traditional operator. Since the selection of a macro operator could move the system closer to a solution than the selection of any traditional operator, this would speed the system up significantly. While this can be a significant advantage, systems that produce a great many macro operators will find that the branching factor of search is dramatically increased, which in turn slows down planning. This is known as the *utility problem* [54, 29], and thus systems that learn macro operators must limit themselves to only those macro operators that will provide a clear benefit.

The MORRIS system learned macro operators only if they had one of two characteristics: they represent very frequently used operator sequences, or they represent non-obvious solutions to particularly difficult problems [53]. A strict limit was set on the number of the first type of macro operators retained, and beyond this limit a new one was only added if a less frequently used one could be removed. The latter type of macro operators are found when a sequence of operators is judged not very useful by the search heuristics used within MORRIS but in reality do lead to a solution more quickly than other operators that the heuristic would prefer.

Macro operators could be made more useful by generalizing not just constants to variables, but also the order of operations within the macro operator [57]. In some cases the order of operators in a sequence is arbitrary, but in others it is essential to the success of the plan. Analysis of the plan to determine relationships between atoms added to or deleted from the state by one operator and required by another can be used to find essential ordering constraints and relax others. Thus, macro operators could be used not only in state-space planning, but plan-space planning as well.

MACRO-FF learns two types of macro operators, one from an analysis of the domain itself and one from an analysis of solutions to training problems [6]. In order to generate macro operators of the first type, MACRO-FF looks for static facts in the domain, meaning atoms that are either true or false in the initial state of a problem and can not be altered by any actions. The BLOCKS-WORLD domain has no static facts, but many others do. For example, in the LOGISTICS domain each location is within one and only one city, and this can not change within a problem. MACRO-FF

creates *abstract components* by clustering constants that are linked by static facts, and thus might create one per city in the LOGISTICS domain, containing the city itself and all of its internal locations. It then generates macro operators through search such that each macro operator uses only pieces of a single abstract component. A small number of simple training problems are then solved, and only those macro operators that proved most useful in solving these problems are retained. The other type of macro operator learned by MACRO-FF is designed to exploit information information in domains that utilize more complex representations.

The works on learning macro operators from analyzing plans are generally based on a formalism known as *explanation-based learning* (EBL) [56]. In EBL, a system is given a concept (such as *is a solution to planning problem X*) and an example (such as a plan), and formulates an explanation as to why the example demonstrates the concept. In a classical planning context, this essentially means goal regression, as described in Section 3.2.1. EBL is used in many other contexts as well, such as learning artificial neural networks [48].

7.1.3 Learning Control Rules

In addition to determining what actions should be collected together into meaningful macro operators, explanation-based learning can be used to learn control rules that prioritize or eliminate entirely some choices that a planner could make, to help it reach a solution more quickly. PRODIGY/EBL is one of the earliest and most well-studied systems that learns and plans with control rules [54]. There are four concepts that PRODIGY/EBL can learn: that a particular choice led to success, which results in a rule giving that choice preference over other alternatives; that a particular choice led to failure, which results in a rule pruning that choice; that a particular choice was the only option that resulted in success, which results in a rule that automatically selects that option; and that a particular choice resulted in an undesirable interaction between the current goal and a goal that had previously been accomplished, which results in a rule that other alternatives be given preference over this option. After learning control rules from observation of traces, PRODIGY/EBL

attempts to *compress* rules into ones that are semantically equivalent but easier to evaluate. While planning with control rules, the system constantly maintains statistics about the effectiveness of each control rule, and deactivates those that require more time to evaluate than they save.

The STATIC system, by contrast, uses *partial evaluation* to learn similar control rules [15]. Partial evaluation does not use training examples, and instead learns by directly processing the domain description. After selecting an arbitrary lifted atom that could be a goal in a planning problem, STATIC chains backwards through operators, generating a *problem space graph*, which is similar to a planning graph [5] but backwards-chaining and uninstantiated. Analysis of a problem space graph may yield rules similar to those learned by PRODIGY/EBL, but without the bias of specific training examples.

Rather than control rules, a control policy may be used, which specifies for each state and set of goals what action should be taken next. This is very similar to the policies discussed in Section 2.3.1 and Section 2.4.1, but can also be used to guide a traditional classical planner. The LRW-LEARN system iteratively improves such a policy by generating its own training examples from random walks through the domain [16].

A similar related work is learning *concept-based policies*, which specify which actions to take based on *concepts*, which are Horn clauses over the atoms in the domain and other concepts [51]. First, this system generates a set of all possible concepts in the domain up to a certain complexity. A number of rules of the form "if lifted concept applies to set of objects, apply action to them" are considered and those that would make effective decisions most often in a small number of training examples are retained.

DISTILL learns domain-specific planners from example plans annotated with reasons actions were chosen [92]. These domain-specific planners use structured programming constructs such as branching and repetition as well as planning-specific constructs such as predicates and operators. Simple domain-specific planners are essentially nested if statements in which the presence of lifted atoms in the current state or goals select an appropriate operator or sequence of operators.

7.1.4 Learning Action Models

A different strain of work is using machine learning not to speed up planning, but to make planning possible at all. The systems discussed above, as well as HTN-MAKER and variants, assume that a complete and accurate description of the actions in a domain are given. Other systems attempt to learn these action descriptions.

GIPO is a graphical user interface intended to assist in domain modeling, and has been enhanced with learning to simplify the user's work [52]. The user provides a plan, consisting of operator names and parameters, as well as information about object types and predicates. The system iterates through the plan, after each action asking the user to specify, for each parameter of the action, what predicates the action makes true about that parameter. The system uses this information and heuristics to determine likely preconditions, negative effects, and positive effects for each action.

In another work, a planning agent knows the current state, its goals, and the names of actions [89]. It selects an action and either receives a complete description of the subsequent state or a notification that the action may not be taken in the current state. Based on these experiences, the agent updates its beliefs about the preconditions and effects of the action based on rules such as "if an atom was not in the current state and the action was successful, then it is not a precondition of the action". In an alternative configuration, when the system lacks sufficient information to guarantee a solution to a problem, it requests one from a teacher, which decreases the amount of experience needed to learn complete descriptions.

The ARMS system learns action models from input plan traces whose intermediate states are partially observable, meaning that the results are always probabilistic [98]. 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.

A different approach has been integrated into the SOAR cognitive architecture [94]. This system does not attempt to learn a precise, symbolic action model, but instead learns from experience in a manner similar to case-based reasoning. When the system applies an action to a state and receives information about the following state, it stores this information in an episodic memory. When it wishes to predict the outcome of applying that action to a different state, it searches its episodic memory for a most similar state in which that action was previously applied, analyzes the way the action modified that state, and predicts that it will modify the current state in equivalent ways.

7.1.5 Other

Another related work learning abstractions for systems that formulate and refine abstract plans, such as ABSTRIPS [79]. The ALPINE system learns abstraction hierarchies for planning [40]. The idea of an abstraction hierarchy is that an abstract plan may be found for a problem and then refined through subsequently decreasing levels of abstraction to a concrete plan. HTNs can also be thought of as representing plans at different levels of abstraction, but ALPINE does not use task decomposition. Rather, an abstraction space consists of a relaxation of the original problem in which some of the predicates are ignored. This may make a relaxed version of the plan very easy to solve, but it may not always be possible to efficiently extend an abstract plan as more predicates become relevant. The ALPINE system automatically explores a variety of abstraction spaces and chooses the best for a given domain. The PARIS system learns a more complex set of abstraction spaces than those that can be generated by dropping predicates based on a user-provided generic abstraction theory for the domain [4].

STEPPINGSTONE is another integrated planning system that learns how to solve problems more efficiently [76]. When the reasoner in STEPPINGSTONE encounters

a particularly difficult subproblem, it records as "stepping stones" a list of subgoals that make achieving the subproblem easier. The next time that this difficult subproblem arises, the planner will begin by achieving the subgoals in the stepping stones for that subproblem.

The L2Act system learns production rules in which a conjunction of statements that an atom appears or does not appear in either the current state or the goals yields an action that should be taken [39]. When planning, the system considers each rule in order and applies the first one that matches. The learning component first enumerates all rules that it will consider, then evaluates each potential rule on each triple of state, goal, and action in some example plans. It then selects a rule following some preference criteria (such as the rule with the highest ratio of triples in which it chooses the correct action to triples in which it is applicable), removes the covered triples from consideration, and repeats the process. The resulting series of rules models a strategy for solving problems in the domain that, while not hierarchical, is similar to the sort of strategies that HTN-Maker learns.

7.2 Learning For Hierarchical Planning

One area of research is the development of models of task relationships that are similar to HTN methods through mixed-initiative systems in which users provide examples and explanations or annotations of these examples [86, 19]. There have also been a number of related works that learn hierarchical planning knowledge from examples without human interaction. I have identified six characteristics by which these systems may be classified:

- Does the system learn the relationships between tasks and subtasks, or is this knowledge an input to the system?

- Does the system learn the preconditions of methods or equivalent knowledge structures, or is this knowledge an input to the system?

- Does the system have complete knowledge about the preconditions and effects of actions or the contents of intermediate states within plans, or is this knowledge of the domain physics incomplete?
- Does the system support learning of tasks with rich semantics, or only simple sub-goaling?
- Does the system learn incrementally, or does it need to process a complete set of training examples before producing anything useful?
- Can the knowledge structures learned by the system be used to solve general HTN planning problems, classically-partitionable planning problems, or only classical planning problems?

The CaMeL system is designed to learn the preconditions of Shop-like methods, assuming that the structure of those methods (the head and subtasks of each) is already known [33]. The input to CaMeL consists of traces from an HTN planner, showing the entire decomposition tree for a problem as well as all methods that were applicable but not chosen at each decision point. All states in which a method was selected or reported as applicable are taken as positive examples for that method, while all states in which that method was not applicable (but its head matched the current task) are taken as negative examples for that method. CaMeL then uses the candidate elimination algorithm to extract, for each method, the set of preconditions that best explains the positive and negative examples. Because the task-subtask relationships are provided, I believe CaMeL should be able to learn preconditions for methods that can be used to solve general HTN planning problems. The system was later extended to incrementally learn approximate preconditions [32].

The DInCAD system also assumes that knowledge of method structures is known but method preconditions are not [95]. It does not explicitly learn preconditions for methods, but instead builds this sort of knowledge implicitly through a case library in which each case stores a situation in which a method was used to decompose a task. In order to adapt these cases to similar circumstances, DInCAD requires an ontology of the types of objects in the domain and induces preferences

System Name	Task-Subtask Relationships	Method Preconditions	Domain Physics
HTN-Maker	Learned	Learned	Input
CaMeL	Input	Learned	Input
DInCAD	Input	Learned	Input
Light	Learned	Learned	Input
X-Learn	Learned	Learned	Input
Learn-HTN	Input	Learned	Learned
L-HTN	Partial	Input	Input

System Name	Task Types	Learning Style	Learned Expressivity
HTN-Maker	Complex	Incremental	Class-Part
CaMeL	Complex	Nonincremental	HTN
DInCAD	Complex	Incremental	HTN
Light	Simple	Incremental	Classical
X-Learn	Simple	Incremental	Classical
Learn-HTN	Complex	Nonincremental	HTN
L-HTN	Complex	Nonincremental	HTN

Table 7.1: Hierarchical learning systems at a glance

for which a case should only instantiated for either particular constants or particular types of constants. Because the task-subtask relationships are provided, I believe DInCAD with appropriate cases should be able to solve general HTN planning problems.

Teleoreactive logic programs (TRLPs) are a variant of hierarchical task networks using *primitive skills* (analogous to actions), *nonprimitive skills* (similar to methods), and *concepts* (Horn clauses specifying subgoal relationships). Nonprimitive skills have heads, preconditions, and subskills. The head of each nonprimitive skill is either a lifted atom or the head of a concept. The planning component of the Icarus cognitive architecture is based on TRLPs, and it learns nonprimitive skills as necessary [47]. Icarus operates in discrete cycles of observing the state of the world, formulating a plan, and executing the first step of that plan until it observes a world in which its goals hold. When the skills in the knowledge base of Icarus are not sufficient to construct a plan directly, it uses a traditional search based on the application of primitive skills, then learns new nonprimitive skills based on its experience. In particular, at each step it learns a nonprimitive skill whose head is the goal that the problem solver was trying to achieve, and whose subskills are either primitive skills that it executed or subgoals that it achieved through the use of other nonprimitive skills. TRLPs do not encode classically-partitionable problems, though I expect it would be easy to make them do so. The Light system uses similar procedures to learn teleoreactive logic programs by observing traces from an expert [66, 50].

The DLight system is an extension of Light, intended to produce skills that are less generally applicable than those of Light but more generally applicable than the methods learned by HTN-Maker [65]. Specifically, DLight learns the same type of structures as Light while maintaining information about dependencies between goals for which actions may have been interleaved in input plans. To do so, it requires that input plans include annotations for each goal that is accomplished specifying a subplan over which this happens, and when these subplans overlap making subgoals that would have become subskills instead preconditions in such a way that necessary ordering constraints will be preserved.

A different extension to LIGHT is LIGHTNING [42]. LIGHTNING begins by using LIGHT directly to learn nonprimitive skills, which it then refines to add more specific preconditions. It does this by using the skills to solve planning problems and observing both when the use of skill succeeds, producing a positive example, and when the planner must eventually backtrack from the selection of a skill, producing a negative example. An inductive logic programming algorithm produces a new, more accurate set of preconditions based on the existing ones and the positive and negative examples.

The X-LEARN system 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 [74]. X-LEARN has been conceived in the context of bootstrap learning where it assumes that the initial training examples solve simple goals and then more complex examples are given to solve more complex goals. X-LEARN is designed to exploit this iterative growth of knowledge by reducing complex goals into subgoals it has already learned to solve.

LEARN-HTN is an extension of the ARMS [98] system to learn method preconditions as well as action models [99]. Thus, it requires as input full HTN decomposition trees but incomplete and noisy information about the contents of intermediate states and the way that actions modify them. LEARN-HTN uses all the types of constraints already present in ARMS and adds constraints that method preconditions must be true in the state from which they were used and that the preconditions of a method should be related to the preconditions of its subtasks. In addition to allowing the learning of method preconditions, these additional constraints can improve the accuracy of action models as well. Because task structures are provided as input, I believe LEARN-HTN could learn preconditions for methods that could be used to solve general HTN planning problems.

The L-HTN system requires partial decomposition trees that specify what tasks were accomplished at each level of the three but not how the tasks at one level are related to the tasks at another level [97]. It assumes that tasks at each level are not interleaved, and attempts to determine which tasks at a given level are subtasks of each task at the next-highest level, modelling this problem as Markov

decision process. Tasks in this system have no semantics, and are merely names given to sequences of subtasks that frequently appear together. Because the method structures are learned from non-semantic data, it seems likely that the methods learned could be used to solve general HTN planning problems.

7.3 Miscellaneous

The recursive structure of HTN methods is very similar to that of context-free grammars, and so systems that learn grammars are somewhat related. Expectation-maximization techniques have been applied to that problem and extended to learning user preferences over possible task reductions [49].

One approach to fast optimal planning is A* search with an admissible heuristic, but there is no heuristic that is dominant in all cases. Another work evaluates multiple heuristics on a few states in a domain, determining for each state which heuristic produces the best ratio of state-space reduction to time spent evaluating the heuristic, and uses these states as training examples for a classifier [11]. When the system reaches a state during planning, it uses whichever heuristic the classifier chooses as most efficient.

Within the field of reinforcement learning there has been much study of Hierarchical Reinforcement Learning (HRL) [3]. In HRL, reinforcement learning problems are modeled at different levels of abstraction. This is distinct from using reinforcement learning in an inherently hierarchical context, as Q-Maker and Q-Reinforce do.

The most similar type of HRL to HTN-Maker is *options* [84]. An option is a policy in which the agent may decide to take a primitive action or a (nonprimitive) policy, within which the agent would make a further choice. The decision as to what states and actions should be grouped into an option is made by a domain expert in the systems with which I am familiar, rather than learned. Hierarchies of Abstract Machines (HAMs) are a similar approach [69]. A planner that uses HAMs directly composes subpolicies into a complete policy that is a solution for the

original problem.

There are also function approximation and aggregation approaches to hierarchical problem-solving in RL settings. Perhaps the best known technique is MAX-Q decompositions [10]. These approaches are based on hierarchical abstraction techniques that are somewhat similar to HTN planning. Given a Markov decision process, the hierarchical abstraction of the MDP is analogous to an instance of the decomposition tree that an HTN planner might generate. Again, however, the MAX-Q tree must be given in advance and it is not learned in a bottom-up fashion as in this work.

Chapter 8

Conclusions

HTN planning is an effective problem-solving paradigm, and has been successfully used in many domains [61]. This wide adoption of HTN planning technology has come at a high knowledge engineering cost of developing and debugging HTN methods for each new domain. As a result, several researchers have designed systems to learn HTN methods or structures similar to HTN methods, each of which have their own knowledge requirements. I have attempted to reduce this knowledge engineering burden with the HTN-Maker framework, which learns HTN methods from example plans and semantically annotated tasks, which I believe is a lower threshold of knowledge than any other system that has been proposed for learning HTN methods or similar constructs. HTN-Maker incrementally adds to its knowledge base by analyzing example plans to find action sequences that accomplish a task and extracting from the plan knowledge about how the task was accomplished in the form of recursive HTN methods. The preconditions and subtasks of these learned methods are determined through a new, hierarchical form of goal regression.

The HTN-Maker framework is designed to be flexible, and when implementing the algorithm, a number of decisions need to be made. First, a decision must be made regarding nondeterministic selection of possible subtasks for methods, which will ultimately determine the shape of decomposition trees using the learned methods. Allowing the system to backtrack and learn all possible structures would result in the learning of many redundant copies of the same knowledge, but some choices will

result in methods that are inefficient or even dangerous depending on the design of the HTN planner that will use them. The most straight-forward, generally useful structure is one of right-recursion, in which the only nonprimitive task a method may have as a subtask is a recursive call to the head of the method, and it must be the last subtask.

Constants encountered in a plan may be generalized to variable symbols in a way that produces methods applicable only to situations highly similar to those from which a method was learned (strong generalization), or in a way that produces broadly applicable methods (weak generalization). When using the former, more training examples are needed and more methods must be learned in order for an HTN planner to be able to solve all problems in a domain, and in most cases planning with these highly-specific methods is slower than planning with the highly-general methods that would be learned with weak generalization.

HTN-Maker will learn similar methods from similar examples, and will learn identical methods from identical sub-examples or examples that differ only in unimportant ways. Thus, it is important to perform some pruning of learned methods. In the simplest case, a method is discarded if it is identical to any existing method. With subsumption checking enabled, a method is discarded if any other method is provably more general than it and both will produce the same results. The use of subsumption checking can drastically reduce the number of methods learned by HTN-Maker without reducing the number of problems solvable with those methods, and does not have a substantial effect on the running time of HTN-Maker.

When learning only right-recursive methods, HTN-Maker is sound in the sense that the methods it learns cannot be used in a way that is inconsistent with the annotated tasks from which they were learned, even though the planner is not aware of these annotations. If it is desirable for HTN-Maker to learn methods that are not right-recursive, verification tasks can be added to the domain to ensure soundness. HTN-Maker is also complete in the sense that, given a finite number of examples from a domain it will learn a set of methods that can be used to solve every problem in that domain that is solvable and can be expressed using the annotated tasks provided to HTN-Maker. In the worst case, it will need to analyze a solution

to each problem in the domain, but in practice a few hundred randomly-generated examples are usually sufficient.

There exists a class of problems, called classically-partitionable, that cannot be expressed in classical planning, yet can be solved by an HTN planner using only methods learned by HTN-Maker. Such problems consist of a series of sets of goals such that the first set must be achieved, then the second set must be achieved (without necessarily maintaining the first set), and so forth. The HTN-Maker algorithm is polynomial in the number and length of training examples and the number of annotated tasks.

The HTN-Maker algorithm can be extended to learn in nondeterministic domains, where actions have a set of possible outcomes rather than a single outcome. HTN-MakerND is such an extension, which works by initially partitioning nondeterministic actions into a set of deterministic pseudo-actions, learning methods based on these pseudo-actions, and then recombining the pseudo-actions into nondeterministic actions. In order for methods to be useful in a nondeterministic domain they must have a structure that no subtask list includes a nondeterministic action followed by another action. Like HTN-Maker, HTN-MakerND is sound as long as it learns only right-recursive methods, and it is complete.

The objective of learning in HTN-Maker is completeness (the ability to solve as many problems as possible) and to a lesser extent efficiency (the ability to solve problems as quickly as possible). An additional common objective in planning is optimality (the ability to find high-quality solutions to problems). The combination of Q-Maker, Q-Reinforce, and Q-Shop attempts to improve optimality without sacrificing completeness or efficiency. Q-Maker is an extension of HTN-Maker that learns, along with each method, a numerical estimate of the value of that method where higher-valued methods are believed to produce higher-quality plans than lower-values methods in situations where both are applicable. Q-Reinforce uses a Monte Carlo based reinforcement learning algorithm to refine the values of estimates as it uses methods to solve problems and receives rewards for finding high-quality solutions. Q-Shop solves problems by using at all times the method with highest estimated value among all applicable methods, without backtracking.

Thus, it should find a solution of reasonably high quality without expending any more effort than an HTN planner that focuses solely on efficiency. The reward received by Q-REINFORCE and predicted by Q-MAKER can be adjusted for various notions of plan quality, but the simplest approach is that high-quality plans have few actions and the reward for using a method is the negative of the number of primitive subtasks of that method.

An experimental evaluation of HTN-MAKER shows that a small number of examples is sufficient to learn a set of methods that can be used to solve most problems in a domain, but there is a long tail of more complicated problems. Planning with methods learned by HTN-MAKER is slower than planning with methods written by an expert, but this difference appears to be only a constant factor. Planning with methods learned by HTN-MAKER scales much better with problem size than classical planning in domains that are well-suited to the task representation that has been chosen for HTN-MAKER, but poorly in a domain in which planning with hand-crafted methods using the same task structure also performed poorly. Planning with methods learned by HTN-MAKERND scales much better with problem size than the traditional model-checking alternative for nondeterministic domains. Planning with methods learned by Q-MAKER and refined by Q-REINFORCE is incomparably faster than planning systems that guarantee an optimal solution, and at least as fast as classical planners that focus on efficiency. Solutions produced from the methods learned by Q-MAKER and refined by Q-REINFORCE are of higher quality than those produced by classical planners that focus on efficiency, and in many cases are optimal.

8.1 Future Work

Evaluation of HTN-MAKER and related algorithms is currently restricted to domains that can be represented using the relatively simplistic action formalism used throughout this document, but real-world problems may require more complex representations. As a result, many modern planning systems support extensions such

as durative actions, negative preconditions, numeric quantities, type ontologies, existential and universal quantification, preferences, conditional effects, disjunction, derived predicates, and modal constraints to the representation language. Supporting any of these would require extending the reasoning mechanisms on which HTN-Maker is based, and in most cases it is not obvious how to do so. Nevertheless, extending the representation language used by HTN-Maker would enable a more thorough and interesting evaluation, and is probably an essential step before using HTN-Maker outside of benchmark, "toy" domains.

Furthermore, classical planning makes a number of assumptions that may not be valid in real-world domains, among them that the number of states be finite, the the current state of the world be completely known to the system at all times, that actions have one predictable outcome, and that no external agents can affect the world. HTN-Maker has already been extended to HTN-MakerND, which can operate in domains where actions are nondeterministic and, by modeling the actions of outside agents as nondeterminism in the actions chosen by the planner, domains where external agents are able to modify the world. Extending HTN-Maker to work in domains where other classical assumptions are invalid is probably also necessary before HTN-Maker can be used to assist real-world knowledge engineers.

Yet another current limitation to the types of domains in which HTN-Maker can be used effectively is the rigid nature of annotated tasks. As discussed in Section 6.1, a method that has been learned for a task to create a pile of 100 blocks cannot be used to accomplish tasks to create piles of 99 or 101 (or any other number) of blocks unless another method has been learned that has the task of creating a pile of 100 blocks as a subtask. It would be desirable to be able to create a flexible annotated task for creating a pile of a variable number of blocks, but it is not clear how this could be incorporated into the framework.

The objective of HTN-Maker has been to reduce the knowledge engineering burden of extending HTN planning to new, real-world domains in which no methods had yet been developed and doing so manually would be prohibitively expensive. Thusfar, neither HTN-Maker nor any other system for learning HTN methods or

similar structures has been used in this way. As discussed above, real-world problems that are interesting enough to make this a worthwhile endeavor typically require a more complex representation language than that currently supported by HTN-MAKER, and may even require relaxing classical assumptions about the domains to learn.

The use of reinforcement learning to rank methods has only begun to be explored. Monte Carlo techniques were used in this work because they are simple and do not require an explicit model of transitions, but temporal-difference techniques would likely enable Q-MAKER and Q-REINFORCE to more quickly converge toward values that accurately predict the quality of plans produced by methods. The framework of Q-MAKER, Q-REINFORCE, and Q-SHOP has not yet been evaluated with quality metrics other than the inverse of plan length. Although it should generalize very easily to schemes in which some actions have different costs than others, it would also be interesting to see its use in schemes where the quality of a plan is not purely a function of the actions contained within it. One such example is strategy formulation in computer games, where a high quality plan is one that wins, regardless of how it does so.

Although HTN-MAKER can be configured to produce methods that are more general or more specific, a post-processing step like that used in LIGHTNING [42] would probably improve either result. More knowledge-based post-processing steps would be interesting as well, such as replacing several methods that do the same thing in disparate circumstances with a single method applicable in all those situations, in effect creating a method that subsumes the others.

The techniques used in DLIGHT [65] might also be useful here. In some domains, there are preconditions that a human can readily determine do not matter, but it has not been clear how HTN-MAKER could determine this automatically. For example, in the LOGISTICS domain the same actions need to be taken to move a package into the proper city regardless of what will need to be done with that package once it reaches the proper city. The current HTN-MAKER algorithm has no way of knowing this, and thus learns separate methods for each combination of intercity and intracity transport strategy. Even in the best cases HTN-MAKER learns far

more methods for a domain than a human expert would write because it currently lacks the ability to recognize these situations.

The language of annotated tasks permits far more expressive task hierarchies than have been tested thusfar. For example, one possible way to achieve the results described in the previous paragraph without adding further reasoning capabilities to HTN-Maker would be to use separate tasks for intercity and intracity transport by adding preconditions to annotated tasks. Adding additional knowledge through the annotated tasks might result in a more compact and easily understood set of methods that can be used for faster planning than those produced using the absolute minimum knowledge.

No attempt has yet been made to transfer knowledge learned by HTN-Maker for one domain into a different but related domain. Because the methods learned by HTN-Maker can capture structural knowledge about a domain, this knowledge may be useful in other domains that do not use the same predicates or actions, yet require the same type of problem-solving strategies.

A comparative evaluation of HTN-Maker to some of the other systems described in Section 7.2 would be difficult for several reasons. Notably, each system has its own types of input and produces knowledge constructs that are similar in style but could not be used by the same planners. Still, an experimental evaluation of the utility of the knowledge learned by these varying systems would be quite interesting. Also, I have made the claim that producing correct action models and annotated tasks for a domain is a significantly simpler knowledge engineering task than producing, for example, primitive skills and a concept hierarchy for Light. It would also be interesting to perform a study in which knowledge engineers with a basic understanding of planning technology but no training in the specific knowledge requirements of any system were asked to produce the inputs to HTN-Maker and other related works to determine which frameworks in fact have the least arduous knowledge engineering burden.

Bibliography

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[2] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

[3] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1–2):41–77, 2003.

[4] Ralph Bergmann and Wolfgang Wilke. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, 3:53–118, 1995.

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

[6] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[7] Marc Cavazza and Fred Charles. Dialogue generation in character-based interactive storytelling. In R. Michael Young and John E. Laird, editors, *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, pages 21–26. AAAI Press, June 2005.

[8] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.*, 147:35–84, 2003.

[9] Ken Currie and Austin Tate. O-PLAN: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[10] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[11] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. To max or not to max: Online learning for speeding up optimal planning. In Maria Fox and David Poole, editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1071–1076. AAAI Press, July 2010.

[12] Kutluhan Erol, James Hendler, and Dana S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In Kristian J. Hammond, editor, *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 249–254. AAAI Press, June 1994.

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

[14] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

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

[16] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In Shlomo Zilberstein, Jana Koehler,

and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 191–199. AAAI Press, June 2004.

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

[18] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[19] Andrew Garland, Kathy Ryall, and Charles Rich. Learning hierarchical task models by defining and refining examples. In *Proceedings of the 1st International Conference on Knowledge Capture (K-CAP-01)*, pages 44–51. ACM, October 2001.

[20] Alfonso Gerevini and Ivan Serina. Efficient plan adaptation through replanning windows and heuristic goals. *Fundamenta Informaticae*, 102(3–4):287–323, 2010.

[21] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL — the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, October 1998.

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

[23] Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI-69)*, pages 213–240. William Kaufmann, May 1969.

[24] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2–3):223–254, 1992.

[25] Kristian J. Hammond. CHEF: A model of case-based planning. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, pages 267–271. Morgan Kaufmann, August 1986.

[26] Patrik Haslum. Additive and reversed relaxed reachability heuristics revisited. In *6th International Planning Competition Booklet (IPC-6)*.

[27] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical plan representations for encoding strategic game AI. In R. Michael Young and John E. Laird, editors, *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, pages 63–68. AAAI Press, June 2005.

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

[29] Lawrence B. Holder. The general utility problem in machine learning. In Bruce W. Porter and Raymond J. Mooney, editors, *Proceedings of the 7th International Conference on Machine Learning (ICML-90)*, pages 402–410. Morgan Kaufmann, June 1990.

[30] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[31] C. W. Hus and B. W. Wah. The SGPlan planning system in IPC-6. In Malte Helmert, Minh Do, and Ioannis Refanidis, editors, *Proceedings of the Sixth International Planning Competition (IPC-6) at ICAPS-08*, September 2008.

[32] Okhtay Ilghami, Héctor Muñoz-Avila, Dana S. Nau, and David W. Aha. Learning approximate preconditions for methods in hierarchical plans. In Luc De Raedt and Stefan Wrobel, editors, *Proceedings of the 22nd International Conference on Machine Learning (ICML-05)*, pages 337–344. ACM, August 2005.

[33] Okhtay Ilghami, Dana S. Nau, Héctor Muñoz-Avila, and David W. Aha. CaMeL: Learning method preconditions for HTN planning. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, pages 131–142. AAAI Press, April 2002.

[34] Froduald Kabanza, Michel Barbeau, and Richard St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–114, 1997.

[35] Subbarao Kambhampati and James Hendler. A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, 55(2–3):193–258, 1992.

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

[37] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 359–363. John Wiley and Sons, August 1992.

[38] Emil Keyder and Héctor Geffner. Heuristics for planning with action costs revisited. In *ECAI-08*, pages 588–592.

[39] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1–2):125–148, 1999.

[40] Craig Knoblock. *Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning*. Kluwer Academic Publishers, Norwell, MA, 1993.

[41] Janet Kolodner. *Case-Based Reasoning*. Morgan Kaufmann, 1993.

[42] Tolga Könik, Negin Nejati, and Ugur Kuter. Inductive generalization of analytically learned goal hierarchies. In Luc De Raedt, editor, *Proceedings of*

the 19th International Conference on Inductive Logic Programming (ILP-09), pages 65–72. Springer, July 2009.

[43] Richard E. Korf. Macro-operators: a weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[44] Ugur Kuter and Dana Nau. Forward-chaining planning in nondeterministic domains. In George Ferguson and Deborah McGuinness, editors, *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, pages 513–518. AAAI Press, July 2004.

[45] Ugur Kuter, Dana Nau, Marco Pistore, and Paolo Traverso. Task decomposition on abstract states, for planning under nondeterminism. *Artif. Intell.*, 173(5–6):669–695, 2009.

[46] Ugur Kuter, Dana Nau, Elnatan Reisner, and Robert P. Goldman. Using classical planners to solve nondeterministic planning problems. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-08)*, pages 190–197. AAAI Press, September 2008.

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

[48] Geoffrey Levine, Ugur Kuter, Kevin Van Sloten, Gerald DeJong, Derek Green, Antons Rebguns, and Diana Spears. Using qualitative domain proportionalities for learning mission safety in airspace operations. In Héctor Muñoz-Avila and Ugur Kuter, editors, *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge from Observations (StrucK-09)*, July 2009.

[49] Nan Li, Subbarao Kambhampati, and Sungwook Yoon. Learning probabilistic hierarchical task networks to capture user preferences. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1754–1759, July 2009.

[50] Nan Li, David J. Stracuzzi, Gary Cleveland, and Pat Langley. Learning hierarchical skills for game agents from video of human behavior. In Héctor Muñoz-Avila and Ugur Kuter, editors, *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge from Observations (StrucK-09)*, July 2009.

[51] Mario Martin and Hector Geffner. Learning generalized policies in planning using concept languages. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR-00)*, pages 667–677. Morgan Kaufmann, June 2000.

[52] T. L. McCluskey, N. Elisabeth Richardson, and Ron M. Simpson. An interactive method for inducing operator descriptions. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, pages 121–130. AAAI Press, April 2002.

[53] Steven Minton. Selectively generalizing plans for problem-solving. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 596–599. Morgan Kaufmann, August 1985.

[54] Steven Minton. *Learning Effective Search Control Knowledge: an Explanation-Based Approach*. PhD thesis, Carnegie Mellon University, 1988.

[55] Steven W. Mitchell. A hybrid architecture for real-time mixed-initiative planning and control. In *Proceedings of the 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 1032–1037. AAAI Press, July 1997.

[56] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[57] Raymond J. Mooney. Generalizing the order of operators in macro-operators. In John E. Laird, editor, *Proceedings of the 5th International Conference*

on Machine Learning (ICML-88), pages 270–283. Morgann Kaufmann, June 1988.

[58] Héctor Muñoz-Avila and Michael T. Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 23(4):75–81, 2008.

[59] Héctor Muñoz-Avila, Daniel C. McFarlane, David W. Aha, Len Breslow, James A. Ballas, and Dana S. Nau. Using guidelines to constrain interactive case-based HTN planning. In Klaus-Dieter Althoff, Ralph Bergmann, and Karl Branting, editors, *Proceedings of the 3rd International Conference on Case-Based Reasoning and Development (ICCBR-99)*, pages 288–302. Springer, July 1999.

[60] J. William Murdock. *Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation*. PhD thesis, Georgia Institute of Technology, 2001.

[61] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Héctor Muñoz-Avila, J. William Murdock, Dan Wu, and Fusun Yaman. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

[62] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2002.

[63] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 968–973. Morgan Kaufmann, July 1999.

[64] Dana S. Nau, Satyandra K. Gupta, and William C. Regli. AI planning versus manufacturing-operation planning: A case study. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1670–1676. Morgan Kaufmann, August 1995.

[65] Negin Nejati, Tolga Könik, and Ugur Kuter. A goal- and dependency-directed algorithm for learning hierarchical task networks. In Yolanda Gil and Natasha Fridman Noy, editors, *Proceedings of the 5th International Conference on Knowledge Capture (K-CAP-09)*, pages 113–120. ACM, September 2009.

[66] Negin Nejati, Pat Langley, and Tolga Könik. Learning hierarchical task networks by observation. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine Learning (ICML-06)*, pages 665–672. ACM, June 2006.

[67] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.

[68] Santiago Ontañón, Kane Bonnette, Prafulla Mahindrakar, Marco A. Gómez-Martín, Katie Long, Jainarayan Radhakrishnan, Rushabh Shah, and Ashwin Ram. Learning from human demonstrations for real-time case-based planning. In Héctor Muñoz-Avila and Ugur Kuter, editors, *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge from Observations (StrucK-09)*, July 2009.

[69] Ronald Parr. *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley, 1998.

[70] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reither, editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pages 324–332. Morgan Kaufmann, May 1989.

[71] J. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 103–114. Morgan Kaufmann, October 1992.

[72] M. Alicia Pérez and Jaime G. Carbonell. Control knowledge to improve plan quality. In Kristian J. Hammond, editor, *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 323–328. AAAI Press, June 1994.

[73] Doina Precup, Richard S. Sutton, and Satinder P. Singh. Eligibility traces for off-policy policy evaluation. In Pat Langley, editor, *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 759–766. Morgan Kaufmann, June 2000.

[74] Chandra Reddy and Prasad Tadepalli. Learning goal-decomposition rules using exercises. In Douglas H. Fisher, editor, *Proceedings of the 14th International Conference on Machine Learning (ICML-97)*, pages 278–286. Morgan Kaufmann, July 1997.

[75] Raymond Reither. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[76] David Ruby and Dennis F. Kibler. Steppingstone: An empirical and analytical evaluation. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 527–532. AAAI Press, July 1991.

[77] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[78] Earl Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI-75)*, pages 206–214, September 1975.

[79] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI-73)*, pages 412–422. William Kaufmann, August 1973.

[80] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1–3):123–158, 1996.

[81] G. J. Sussman. *A Computational Model of Skill Acquisition*. Elsevier, New York, NY, 1975.

[82] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[83] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1988.

[84] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211, 1999.

[85] Austin Tate. Generating project networks. In R. Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 888–893. William Kaufmann, August 1977.

[86] Gheorghe Tecuci, Mihai Boicu, Kathryn Wright, Seok Won Lee, Dorin Marcu, and Michael Bowman. An integrated shell and methodology for rapid development of knowledge-based agents. In James Hendler and Devika Subramanian, editors, *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 250–257. AAAI Press, July 1999.

[87] Peter van Beek and Xinguang Chen. CPlan: A constraint programming approach to planning. In James Hendler and Devika Subramanian, editors, *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 585–590. AAAI Press, July 1999.

[88] Manuela Veloso. *Planning and Learning by Analogical Reasoning.* Springer-Verlag, 1994.

[89] Thomas J. Walsh and Michael L. Littman. Efficient learning of action schemas and web-service descriptions. In Héctor Muñoz-Avila and Ugur Kuter, editors, *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge from Observations (StrucK-09)*, July 2009.

[90] David Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.

[91] David E. Wilkins and Marie desJardins. A call for knowledge-based planning. *AI Magazine*, 22(1):99–115, 2001.

[92] Elly Winner and Manuela M. Veloso. DISTILL: Learning domain-specific planners by example. In Tom Fawcett and Nina Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 800–807. AAAI Press, August 2003.

[93] T. Winograd. *Understanding Natural Language.* Academic Press, 1972.

[94] Joseph Z. Xu and John E. Laird. Instance-based online learning of deterministic relational action models. In Maria Fox and David Poole, editors, *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1574–1579. AAAI Press, July 2010.

[95] Ke Xu and Héctor Muñoz-Avila. A domain-independent system for case-based task decomposition withotu domain theories. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 234–240. AAAI Press, July 2005.

[96] Qiang Yang. Formalizing planning knowedge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, 1990.

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

[98] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2–3):107–143, 2007.

[99] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Héctor Muñoz-Avila. Learning HTN method preconditions and action models from partial observations. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1804–1810, July 2009.

[100] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 24(2):73–96, 2003.

Vita

Chad Hogg was born in Lancaster, Pennsylvania, USA on May 31, 1982 to Jeffrey and Cynthia Hogg. He earned his Bachelor of Science degree in May 2004 from Ursinus College, where he graduated magna cum laude with interdepartmental honors for his thesis *Computer-Assisted Composition in the 32-bar Jazz Standard Form*. While at Lehigh University, he was a Research Assistant in the SEAL-DB lab from 2004-2006, designing a multi-database system for targeted data sharing among law enforcement agencies; a summer Instructor teaching C programming and the UNIX series of operating systems to undergraduates in 2006; and a Research Assistant in the InSyTe lab from 2006-2010, studying automated planning, machine learning, reinforcement learning, case-based reasoning, and the integration of artificial intelligence research into commercial computer games. During the 2010-2011 academic year he was an Instructor at Mansfield University of Pennsylvania, teaching object-oriented programming, data structures, software engineering, operating system design, and web design and development. He will be returning to his alma mater as a Visiting Assistant Professor in the department of Mathematics and Computer Science at Ursinus College.

Selected Publications

- Chad Hogg, Stephen Lee-Urban, Héctor Muñoz-Avila, Bryan Auslander, and Megan Smith. Game AI for Domination Games. In Pedro A. González-Calero and Marco Antonio Gómez-Martín, editors, Artificial Intelligence for Computer Games, pages 83-101. Springer, 2011.

- Chad Hogg, Ugur Kuter, and Héctor Muñoz-Avila. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In Proceedings of the 24th AAAI Conference on Artificial Intelligence. 2010.

- Hua Li, Héctor Muñoz-Avila, Diane Bramsen, Chad Hogg, and Rafael Alonso. Spatial Event Prediction by Combining Value Function Approximation and Case-Based Reasoning. In Proceedings of the 8th International Conference on Case-Based Reasoning. 2009.

- Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In Proceedings of the 21st International Joint Conference on Artificial Intelligence. 2009.

- Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Héctor Muñoz-Avila. Learning HTN Method Preconditions and Action Models from Partial Observations. In Proceedings of the 21st International Joint Conference on Artificial Intelligence. 2009.

- Bryan Auslander, Stephen Lee-Urban, Chad Hogg, and Héctor Muñoz-Avila. Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning. In Proceedings of the 9th European Conference on Advances in Case-Based Reasoning. 2008.

- Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. HTN-Maker: Learning HTNs with Minimal Additional Knowledge Required. In Proceedings of the 23rd AAAI Conference on Artificial Intelligence. 2008.

- Ian Warfield, Chad Hogg, Stephen Lee-Urban, and Héctor Muñoz-Avila. Adaptation of Hierarchical Task Network Plans. In Proceedings of the 20th FLAIRS International Conference. 2007.