ABSTRACT

Title of Dissertation: AUTOMATING HIERARCHICAL TASK NETWORK LEARNING

Ruoxi Li
Doctor of Philosophy, 2024

Dissertation Directed by: Professor Dana S. Nau
Department of Computer Science

Dr. Mark Roberts
Naval Research Lab

Automating Hierarchical Task Network (HTN) learning is essential for reducing the knowledge engineering burden in automated planning systems. Traditional HTN learning techniques, like HTN-MAKER, face challenges related to scalability, efficacy, and the need for manual input. This dissertation fully automates HTN learning from classical planning problems and significantly enhances the capabilities of the learned methods.

The proposed solution leverages curricula to learn simpler methods first and progressively tackling more complex ones. We use landmarks, facts that must be true in any plan solving the problem, as essential benchmarks to generate curricula. Additionally, the recognition of recursive task decomposition patterns allows for the learning of generalized methods, further improving the applicability of the learned methods.

The primary contributions of this dissertation include the development of CURRICULEARN,

an algorithm that enhances HTN learning through the guidance of curricula; CURRICULAMA, an algorithm that automatically generates curricula from landmarks and utilizes CURRICULEARN to learn from those curricula; and METHODGENERALIZER, an algorithm that learns more generalized methods by capturing recursive task decomposition patterns. Some of these algorithms are theoretically analyzed and all of them are empirically evaluated across various domains, including those from the International Planning Competitions. CURRICULEARN is shown to learn fewer methods more efficiently compared to HTN-MAKER, resulting in higher planning success rates and reduced planning times. CURRICULAMA is proven to fully automate HTN learning while maintaining comparable performance compared to HTN-MAKER. METHODGENERALIZER further enhances the applicability of the learned methods, resulting in significantly high planning success rate for problems of various complexities.

In summary, this dissertation addresses the challenges in HTN learning by fully automating the HTN learning process, and significantly enhances the capabilities of the learned methods, contributing to the advancement of automated planning systems.

AUTOMATED HIERARCHICAL TASK NETWORK LEARNING

by

Ruoxi Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Professor Dana S. Nau, Chair/Advisor
Dr. Mark "Mak" Roberts, Co-Advisor
Professor Dave Levin
Professor Pratap Tokekar
Professor Marine Carpuat
Professor Mark D. Fuge

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisors, Dr. Dana Nau and Dr. Mark Roberts, for your invaluable guidance and support throughout my research journey. Your insightful discussions and unwavering encouragement have been instrumental in shaping this dissertation, and I am truly grateful for the knowledge and wisdom you have shared with me.

I am also profoundly thankful for the unconditional love and support from my family. To my mother, father, and sister—your constant encouragement has been a source of strength and motivation throughout this challenging process.

A special thank you to the Zhixing Reading Club, particularly to my beloved mentor, Mr. Hou Wenkai, and all the other peers who have been part of this incredible community. Your spiritual guidance, intellectual conversations, and camaraderie have provided me with much-needed inspiration and companionship during the ups and downs of my PhD journey.

To all those who have contributed to my academic and personal growth, I extend my heartfelt thanks. Your support and encouragement have made this achievement possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

HTN	Hierarchical Task Network
pHTN	Probabilistic Hierarchical Task Network
HGN	Hierarchical Goal Network
IPC	International Planning Competition
PDDL	Planning Domain Definition Language
STRIPS	Standford Research Institute Problem Solver
TLP	Teleoreactive Logic Program
SHOP	Simple Hierarchical Ordered Planner

Chapter 1: Introduction

Automated planning systems play a crucial role in a wide range of applications, from robotics to logistics and beyond. However, these systems typically require extensive domain knowledge, which must be carefully crafted by domain experts. In classical planning, this knowledge involves the semantic descriptions of actions, while in Hierarchical Task Networks (HTNs), it encompasses structural properties and potential hierarchical problem-solving strategies. Writing HTN decomposition methods is particularly challenging, placing a significant knowledge engineering burden on domain experts. To alleviate this burden, various techniques (*e.g.*, HTN-MAKER [1]) have been developed to learn HTN methods by analyzing the semantics of solution traces for planning problems. However, these techniques often struggle with scalability and efficiency, and they also require some manual input, limiting their automation.

Curriculum learning [2], a strategy used in reinforcement learning, has shown promise in improving learning performance by ordering training examples by problem difficulty. This approach has not been explored in the context of HTN learning until now. The idea of curriculum learning here is to enhance an HTN learner by initially learning simpler methods and gradually progressing to more complex methods that build on the simpler ones. This dissertation introduces an innovative algorithm that leverages curriculum learning to improve HTN learning.

Furthermore, by using curricula generated from landmarks — facts that must be true at

some point in every plan that solves a given problem — we provide a way to completely streamline the learning process. Landmarks play a pivotal role in this approach, as they guide the learning process by highlighting essential benchmarks that need to be achieved in any solution. This allows for the creation of a structured curriculum that progressively teaches the learner to handle more complex tasks. The combination of curriculum learning and landmark-based guidance eliminates a substantial part of the knowledge engineering burden and enhance the capability of automated planning systems.

However, the methods learned by existing techniques, including our advancements, often have limitations. Specifically, they tend to be effective primarily for problems that are similar in scale and complexity to the training examples. These learning algorithms often struggle to generalize beyond the training distribution, limiting their applicability of learned methods to larger and more complex problems. To overcome this limitation, it is crucial to develop more robust methods that can generalize to solve much larger and more complex problems. One promising direction provided by this dissertation is to focus on learning generalized methods with recursive solution structures. These methods can be applied to larger problems, as the recursive nature allows the planning system to handle complex tasks through repeated application of the same method, thus enhancing the capabilities of the learned HTN methods.

1.1 Contributions

This dissertation focuses on automating and improving HTN learning. The contributions of this dissertation include:

- CURRICULEARN is a new HTN learning algorithm that learns HTN methods from a cur-

riculum. Unlike HTN-MAKER, which learns HTN methods by examines every subsequence of a solution plan, CURRICULEARN examines only the subtraces specified by the curriculum. Given a planning problem and its solution trace, a curriculum can be manually constructed from a tree that partitions the actions in the solution trace.

- CURRICULAMA is an algorithm that automatically generates curricula, discovers tasks for which methods are learned in each curriculum step, and learn HTN methods following the curriculum steps. Given a planning problem, a solution trace and a curriculum can be generated following the ordered landmarks. This curriculum, along with the solution trace, can then be used by CURRICULEARN to learn HTN methods. CURRICULAMA completely eliminates manual inputs to an HTN learner.

- METHODGENERALIZER is an algorithm that learns generalized methods applicable to a wider range of problems. Some problems have recursive solutions. Learning a set of methods that can solve problems of the same recursive nature, regardless of problem size, can significantly increase the problem coverage of the learned methods.

- Many of the proposed algorithms are theoretically analyzed. They are also implemented and evaluated across a variety of domains and problem sets, including those from the International Planning Competitions. Comparisons are made with state-of-the-art techniques where applicable.

1.2 Dissertation Organization

The chapters in this dissertation are organized as follows:

- **Chapter 2: Background** covers the necessary background, including classical planning, hierarchical planning, and HTN learning, and provides the formal definitions and theoretical foundations.

- **Chapter 3: Learning HTN Methods Using Curricula** describes CURRICULEARN, a new HTN learning algorithm that learns HTN methods through curricula, detailing its theoretical analysis and experimental results.

- **Chapter 4: Learning HTN Methods with Curricula Automatically Generated from Landmarks** explains the development of an algorithm, CURRICULAMA, that uses landmarks to generate curricula to guide HTN method learning.

- **Chapter 5: Towards Learning Generalized Methods by Capturing Recursive Patterns** explores learning generalized methods with recursive patterns to increase problem coverage.

- **Chapter 6: Related Work** reviews related work, providing context and highlighting the unique contributions of this research.

- **Chapter 7: Conclusion** summarizes the contributions, discusses implications, and suggests potential directions for future work.

The dissertation also includes appendices with additional details: an example HTN domain and problem (Appendix A), and theoretical proofs related to CURRICULEARN (Appendix B). The bibliography lists all references cited throughout the dissertation.

Chapter 2: Background

This chapter introduces background including classical planning, hierarchical planning, HTN learning, and landmarks.

2.1 Classical Planning

Problem representations in automated-planning research began with the early work on GPS [3] and the situation calculus [4], and continued with the STRIPS planning system [5], and the widely used classical representations [6, 7]. The well-known PDDL planning language [8] is based on a classical representation but incorporates a large number of extensions. There are also extensions to PDDL such as PDDL+ that enables continuous state [9] and PPDDL that models uncertainties [10].

Constants, variables, and terms are basic concepts in classical representations. A *constant* is a symbol that refers to a specific object, while a *variable* represents an as-yet unspecified object. Both constants and variables are considered *terms*, which are strings of characters. To distinguish variables, their names begin with a question mark. For example, `?a` can represent a variable, whereas `A` can be a constant referring to a specific object. Objects are typed, *e.g.*, `?a - block` indicates that variable `?a` is a `block` type.

Predicates [11] serve as templates for simple statements about the world. They consist

of a predicate symbol and an arity, which is a non-negative number indicating the number of arguments the predicate takes. In the Blocks World domain, the predicate `on`^{2} describes the spatial relationship between two blocks stacked on one another.

Atomic formulas, or *atoms* [12], are specific statements about the world. An atom comprises an opening parenthesis, a predicate symbol, a set of terms equal to the predicate's arity, and a closing parenthesis. For instance, `( on B A )` indicates that block `B` is on block `A`. When an atom contains no variables, it is referred to as ground, meaning it specifies a concrete fact.

Moving on to more complex structures in automated planning, we first define what constitutes a state.

Definition 1 (State)**.** *A state is a finite set of ground atoms, representing all statements that holds true at some particular point in time. Any atom that does not appear in a state is explicitly false (closed-world assumption). e.g.*[1] *,* { `(on-table A), (on B A), (on C B), (clear C), (hand-empty)` } *indicates a state where block* `A` *is on the table, block* `B`*, and* `C` *are stacking on block* `A`*.*

The classical representation can be generalized to let states be arbitrary data structures, and an action template's preconditions, effects, and cost be arbitrary compute-able functions operating on those data structures. Analogous generalizations can be made to the classical representation by allowing a predicate's arguments to be functional terms whose values are calculated procedurally rather than inferred logically [8]. Such generalizations can make the domain models applicable to a much larger variety of application domains.

Next, we define what an action is and how it affects the state of the world.

[1] see listing A.2 in Appendix A for the same example.

```
( :action
  :head !Unstack
  ( ?b1 - block ?b2 - block )
  :precondition
  ( and
    ( on ?b1 ?b2 )
    ( clear ?b1 )
    ( hand-empty )
  )
  :effect
  ( and
    ( not ( on ?b1 ?b2 ) )
    ( not ( clear ?b1 ) )
    ( not ( hand-empty ) )
    ( clear ?b2 )
    ( holding ?b1 )
  )
)
```

Figure 2.1: An example action.

Definition 2 (Action)**.** *A (lifted) action is a four-tuple* $a = (a^{head}, a^{prec}, a^{eff-}, a^{eff+})$*. The head of an action* a^{head} *includes the name of the action (beginning with an exclamation mark by convention) followed by seom parameters. The preconditions* a^{prec}*, negative effects* a^{eff-}*, and positive effects* a^{eff+} *of an action are finite sets of atoms whose parameters all appear in the head of the action.*

For example, action `!Unstack` shown in Figure 2.1 unstacks block `?b1` from `?b2`. It is applicable when block `?b1` in stacked on `?b2`, top of block `?b1` is clear, and the robot hand is empty. The negative effects after applying this action are the negations of the precondition. The positive effects after applying this action are that top of block `?b2` is clear, and the robot hand is holding block `?b1`. This action is ungrounded, or lifted, in the sense that it could be applied to various situations where some block needs to be unstacked from some other block, as

long as those blocks satisfied the preconditions speficied by the action. Ungrounded actions are sometimes referred to as *operators* [11] or *action templates* [12]. Here, we implicitly refer to both ungrounded and grounded actions as actions. When an action is grouded, specific constants are chosen to replace the variables.

Classical planning [12] involves the formulation and solution (i.e., the *plan*) of a problem using logical reasoning about the actions required to transition from an initial state to a goal state. A classical planning problem is formally defined as a tuple: $P = (\Sigma, s_0, g)$, where Σ is a classical planning domain that contains a finite set of constants representing objects in the problem domain, a finite set of predicates representing logical statements about objects, and a finite set of ungrounded actions representing general actions that can be instantiated with specific parameters. The initial state s_0 is a set of ground atoms describing the world at the start of the problem. The goal condition g is a finite set of ground atoms that must hold in the final state for the problem to be solved.

Definition 3 (Classical Planning Problem)**.** *A* classical planning problem P *is a triple* (Σ, s_0, g)*, where* Σ *is the classical planning domain description,* s_0 *is the initial state, and* g *is the goal condition.*

Definition 4 (Plan)**.** *A plan is a sequence of grounded actions:* $\pi = \langle a_0, a_1, \ldots, a_n \rangle$*, where each action is applicable in the state reached by the previous action. A plan* π *is a solution to a classical planning problem* P *if all the actions in* pi *are instances of the set of ungrounded actions in* P*, and executing the actions of* π *in sequence starting from the initial state* s_0 *leads to a final state* s_f *such that the goal condition* g *holds in* s_f*.*

The classical representation schemes are EXPSPACE-equivalent [13]. The time needed to

solve a classical planning problem may be exponential in the size of the problem description. Many planning algorithms, such as BFS, DFS, A*, GBFS and so on [6], work by searching forward from the initial state to try to construct a sequence of actions that reaches a goal state. Some of those planning algorithms may need heuristic functions such as deletion-relaxation heuristics [14] an landmark heuristics. In contrast, some other planning algorithms do a state-space or plan-space [15] search backwards from the goal.

2.2 Landmarks

A landmark [16] for a planning problem P is a fact that is true at some point in every solution plan that solves P. A landmark graph is a directed graph in which the nodes are landmarks and the edges are ordering constraints. Thus if there is an edge between two landmarks l_i and l_j, then l_i must be true before l_j in every solution to P. There are several types of orderings for landmarks [17]:

- There is a *natural* ordering between ϕ and ψ, written $\phi \rightarrow_n \psi$, if in each plan where ψ is true at time i, ϕ is true at some time $j \leq i$.
- There is a *necessary* ordering between ϕ and ψ, written $\phi \rightarrow_{nec} \psi$, if in each plan where ψ is added at time i, ϕ is true at time $i - 1$.
- There is a *greedy-necessary* ordering between ϕ and ψ, written $\phi \rightarrow_{gn} \psi$, if in each plan where ψ is first added at time i, ϕ is true at time $i - 1$.
- There is a *reasonable* ordering between ϕ and ψ, written $\phi \rightarrow_r \psi$, if a landmark ψ must become false in order to achieve a landmark ϕ, but ψ is needed after ϕ (as otherwise, we

would have to achieve ψ twice).

2.3 Hierarchical Planning

Hierarchically organized planning techniques such as HTN planning [18] are well-established in the AI literature. They have advantages in working out interactions in more abstract plan spaces, thus pruning away large portions of the more detailed search spaces [19, 20]. A drawback of this approach is that it requires the domain author to write and debug a potentially complex set of domain-specific recipes.

In Hierarchical Task Network (HTN) planning, the planning system formulates a plan by decomposing *tasks* (symbolic representations of activities to be performed) into smaller and smaller subtasks until primitive tasks are reached that can be performed directly. The expressive power of HTN methods can be useful for developing practical applications [21].

A task is an activity that can be performed in the world. It is represented by a parenthesis containing a task name and a set of arguments. If a task matches the head of an action, it is considered *primitive*. Otherwise, if the task matches the head of a method, it is considered *compound*.

Definition 5 (Method)**.** *An HTN method*[2] *is a tuple* $(m^{head}, m^{prec}, m^{sub})$*, where:*

- m^{head} *is the task that the method decomposes.*
- m^{prec} *is the precondition, a set of atoms that must be satisfied in the current state for the method to be applicable.*

[2]More examples could also be found in Listing A.1 in Appendix A. We also give each method a unique identifier to distinguish methods that have the same head (see the methods in Figure 4.8).

```
(:method M0
 :head ( Make-On ?b1 - block ?b2 - block )
 :vars ( ?b3 - block )
 :preconditions { ( on-table ?b1 ) ( clear ?b1 )
   ( on-table ?b2 ) ( clear ?b2 ) ( holding ?b3 ) }
 :subtasks < ( !putdown ?b3 ) ( Make-On ?b1 ? b2 ) > )
```

Figure 2.2: An example method.

- m^{sub} *is the decomposition of the task* t*, defined as an ordered set of subtasks that when completed will achieve the task* t*. These subtasks themselves can be primitive or non-primitive, necessitating further decomposition.*

As is shown in Figure 2.2, method `M0` is for decomposing a compound task `( Make-On ?b1 - block ?b2 - block )` — making block `?b1` to be stacked on block `?b2`. `M0` is only applicable when both `?b1` and `?b2` are on the table and clear, and the robot hand is holding a block `?b3`. Method `M0` decomposes the `Make-On` task into two subtasks: 1) a primitive task (an action) to make the robot hand putdown the block that it is holding, 2) a compound task that continues making block `?b1` to be stacked on block `?b2`. The variables in the method that do not belong to its task's parameters, such as `?b3`, will be shown and in its "`vars`" and with designated types.

Just like the definition of actions, the definition of methods highlights the lifted nature of HTN methods, underscoring their generality and applicability across various planning scenarios without the need for grounding in specific object instances. This abstraction facilitates the reusability of methods in diverse contexts where only the types of objects and their interrelations vary.

Definition 6 (Hierarchical Planning Problem)**.** *A hierarchical planning problem* P_h *(e.g., see*

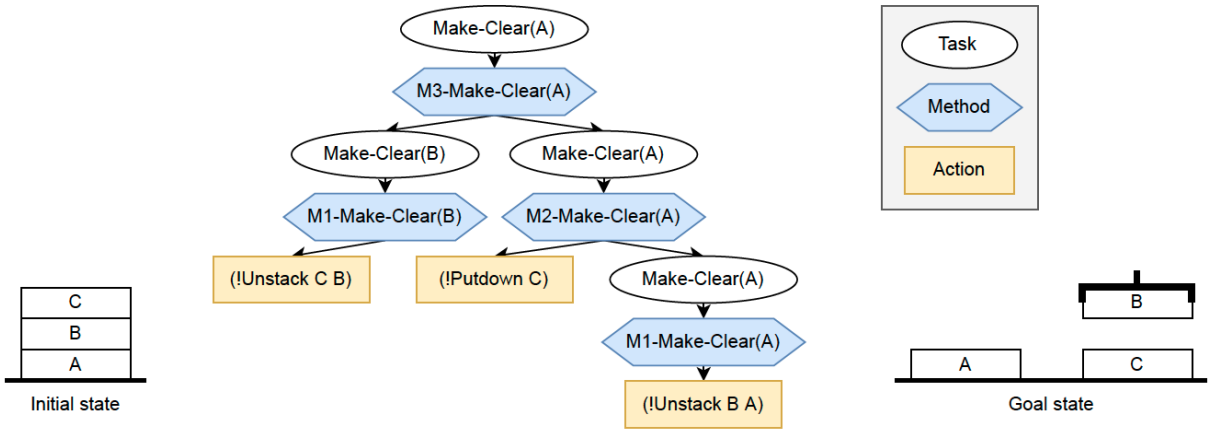


Figure 2.3: An example task decomposition network.

listing A.2 in Appendix A) is a triple $(\Sigma_h, s_0, \langle\tau\rangle)$ *where* Σ_h *is the hierarchical planning domain description (e.g., see listing A.1 in Appendix A) ,* s_0 *is the initial state and* $\langle\tau\rangle$ *is the task list. A hierarchical planning domain description* Σ_h *is a tuple* (Σ, M)*, where* Σ *is the classical planning domain description and* M *is the set of HTN methods.*

HTN planners, such as SHOP [18] and SHOP2 [22], recursively break down a given planning problem into smaller and smaller subtasks using the provided HTN methods until an executable sequence of primitive tasks (actions) is reached. The resulted task decomposition network (see Figure 2.3 for example) is hierarchical and includes orderings among tasks, resulting in an ordered sequence of actions.

The Pyhop [23] planner is a simple SHOP-style planner written in Python. GTPyhop (Goal-and-Task Pyhop) [24] is an extended version of Pyhop that can plan for both goals and tasks, using a combination of SHOP-style task decomposition and GDP-style goal decomposition. It provides a totally-ordered version of Goal-Task-Network (GTN) planning without sharing and task insertion. GTPyhop's ability to represent and reason about both goals and tasks provides a high degree of flexibility for representing objectives in whichever form seems more natural to the

domain designer.

Hierarchical Domain Definition Language (HDDL) is an extension to PDDL (to the needs of hierarchical planning systems), and is the standard input language for the track on hierarchical planning at the IPC 2020. The PANDA [25] framework is a software system to reason about hierarchical planning tasks based on HDDL as input language. Besides solvers for planning problems based on plan space search, progression search, and translation to propositional logic, PANDA also includes techniques for related problems like plan repair, plan and goal recognition, or plan verifcation.

The Action Notation Modeling Language (ANML) [26] provides a high-level, convenient, and succinct alternative to existing planning languages such as PDDL. It is a temporal planning framework with a hierarchical component , designed to address common challenges faced when modeling problems that require sophisticated temporal or resource reasoning within a single language that combines generative and hierarchical task planning; an essential feature of ANML is its ability to embed complex procedures as tasks. The planning system FAPE [27] has implemented a subset of ANML.

The Hierarchical Goal Network (HGN) [28] formalism bridges classical planning and totally ordered task networks by operating over sequences of goals with methods that decompose goals with further subgoals. By attaching goals to methods, HGN planning adds explicit semantics to methods, making it easy to adapt classical heuristics to HGN planning. The Goal-Decomposition Planner (GDP) [28] combines some aspects of both HTN planning and domain-independent planning. For example, it allows the planning agent to use domain-independent heuristic functions to guide the application of both methods and actions. The Goal Decomposition with Landmarks (GoDeL) [29] is sound and complete planner irrespective of whether the

domain knowledge is complete.

The "Actor's View of Automated Planning and Acting" [12] advocates a hierarchical organization of an actor's deliberation functions, with online planning throughout the acting process. The Refinement Acting Engine (RAE) [6] uses hierarchical operational models to perform tasks in dynamically changing environments. The planner, UPOM (UCT-like Procedure for Operational Models) [30], does a Monte-Carlo Tree search in the space of operational models in order to find a near optimal method to use for the task and context at hand.

2.4 HTN Learning

Our work builds on the HTN method learning mechanism of the HTN-MAKER algorithm, which learns hierarchical planning knowledge in the form of decomposition methods for HTNs. HTN-MAKER takes as input the initial states from a set of classical planning problems in a planning domain and solutions to those problems, as well as a set of *semantically-annotated tasks* to be accomplished. The algorithm analyzes this semantic information in order to determine which portions of the input plans accomplish a particular annotated task and constructs HTN methods based on those analyses.

Definition 7 (Annotated Task)**.** *Each annotated task is defined by a head, preconditions, and goals (or effects). An annotated task is a triple* $(\tau^{head}, \tau^{prec}, \tau^{eff+})$*, where:*

- τ^{head} *is the head of the task, specifying the activity to be undertaken.*
- τ^{prec} *represents the preconditions, a set of conditions that must be true before the task can be attempted.*

- τ^{eff+} *denotes the goals (or effects), the set of conditions that will be true once the task is successfully completed.*

```
(:task
 :head (Make-Clear ?a - block)
 :preconditions ()
 :goals ((clear ?a)))
```

Figure 2.4: An Annotated Task Example for the Blocks World domain.

For example, Figure 2.4 shows an annotated task `Make-Clear` for the Blocks World domain. In this example, the head of the task is `Make-Clear` with a parameter `?a` of type `block`. The task has no preconditions, and its goal is to ensure that `?a` is clear. What we call goals, HTN-MAKER calls effects. It does not cause the goals to be true directly; instead, the goals specify what needs to be true after performing the annotated task.

Algorithm 1 A high-level description of the HTN-MAKER procedure. The input includes a classical planning domain description D, an initial state s_0, a solution trace π, a set of annotated tasks T, and a possibly empty set of HTN methods M. The output is an updated set of HTN methods.

procedure HTN-MAKER(D, s_0, π, T, M)
    initialize $X \leftarrow \emptyset$
    let $\vec{S}$ be the state trajectory generated from $\gamma(s_0, \pi)$
    for $e \leftarrow 1$ *to* $|\pi|$ **do**
        for $s \leftarrow e - 1$ down to 1 **do**
            for τ in T **do**
                $M' \leftarrow$ LEARN-METHOD(π, $\vec{S}$, τ, X, s, e)
                $M \leftarrow M \cup M'$
                for $m \in M'$ **do**
                    $X \leftarrow X \cup \{(m^{head}, \tau^{eff+}, m^{sub}, m^{prec}, \text{s}, \text{e})\}$
    return M

Algorithm 1 describes the high-level operation of HTN-MAKER. Its input includes a domain description D, an initial states s_0, an trace π (which can be a plan produced by a planner given a goal), and a set T of *annotated tasks* to be accomplished. HTN-MAKER examines every subtrace of the give plan trace, if the subtrace accomplishes some annotated tasks, HTN-MAKER learns some methods for those tasks.

Algorithm 2 LEARN-METHOD procedure for learning methods from a subtrace.

Input: solution trace π, state trajectory $\vec{S}$, annotated task τ, indexed method instances X, beginning index of the subtrace b, ending index of the subtrace e.

Output: an HTN method.

procedure LEARN-METHOD($\pi, \vec{S}, \tau, X, b, e$)
  $\phi \leftarrow \tau^{eff+}$
  $\omega \leftarrow \langle\rangle$
  $c \leftarrow e$
  while $c > b$ **do**
    $X' \leftarrow \emptyset$
    for $x = (x^{head}, x^{eff+}, x^{sub}, x^{prec}, x^b, x^e) \in X$ **do**
      if $x^e = c \wedge x^b \geq b \wedge x^{eff+} \cap \phi \neq \emptyset$ **then**
        $X' \leftarrow X' \cup \{x\}$
    $a_c \leftarrow$ the c-th action in π
    if $a_c^{eff+} \cap \phi \neq \emptyset$ **then**
      $X' \leftarrow X' \cup \{(a_c^{head}, a_c^{eff+}, \langle\rangle, a_c^{prec}, c-1, c)\}$
    if $X' \neq \emptyset$ **then**
      $x = \texttt{argmax}_{x \in X}(x_e - x_b)$
      $\phi \leftarrow (\phi \backslash x^{eff+}) \cup x^{prec}$
      $\omega \leftarrow \langle x^{head} \rangle \cdot \omega$
      $c \leftarrow x^b$
    else
      $c \leftarrow c - 1$
  return $m = (\tau, \phi \cup \tau^{prec}, \omega)$

The Learn-Method procedure (Algorithm 2) performs hierarchical goal regression over

HTNs. The procedure LEARN-METHOD (line 7)[3] performs the analysis on the subtrace $\pi[s, e]$, and learns some new methods for τ. It also keeps a set of indexed method instances X to identify and reuse previously learned methods as subroutines in a new method (line 10).

Given a plan trace of length k, HTN-MAKER analyzes exactly $\frac{k(k+1)}{2}$ subtraces, which is $\Theta(k^2)$. As a result, HTN-MAKER often learns too many methods, some have undesirable preconditions or decomposition strategies.

2.5 Experimental Domains

The International Planning Competition (IPC) [31] provides a diverse set of benchmark domains designed to evaluate the performance of planning algorithms. Each domain presents unique challenges that require sophisticated planning strategies to address. While HTN-MAKER was evaluated in the Blocks World domain, Logistics domain, Zeno Travel domain, Satellite domain and Rovers domain, we also included the Depots domain, and Minigrid domain (with counters). Below are descriptions of the experimental domains used in our study:

Logistics. The Logistics domain involves the transportation of packages between various locations in a city using trucks and airplanes. The objective is to efficiently route the vehicles to deliver all packages to their destinations.

Blocks World. The Blocks World domain includes a number of blocks sitting on a table (possibly on top of each other) and a robotic hand that can grasp one block at a time. The task is to rearrange the blocks to achieve a specific goal configuration.

[3]The code implementation of the `Learn-Method` procedure can actually learn more than one method because of the existence of alternative ways to bind variable names to object names for an annotated task. Therefore, we made a correction of the pseudocode accordingly.

Depots. The Depots domain is a combination of the Logistics domain and the Blocks World domain. The logistics element of the task is to move crates from one depot to another using trucks, while the blocks world element involves stacking and unstacking crates within the depots.

Satellite. The Satellite domain models the problem of an array of satellites collecting a variety of images using observation instruments each of a specific mode and direction. The challenge is to schedule the observations and manage the reorientation of the satellites to maximize the number of useful images collected.

Zeno Travel. In the Zeno Travel domain, passengers are transported between cities by an aircraft that consumes fuel and can be refueled. Standard predicate logic is used to encode the relationships between the integers zero through five to represent fuel levels. The goal is to optimize the transportation routes while managing fuel consumption effectively.

Rover. The Rover domain involves a planetary rover tasked with various scientific objectives such as taking images, analyzing soil samples, and communicating data back to a base station. The domain includes managing the rover's energy, storage, and communication constraints.

Minigrid with Counters. The Minigrid with Counters domain is a grid-based environment where the planner navigates through rooms, with counters representing distances. The objective is to navigate the grid efficiently, often requiring the planner to reach specific goals or collect items while considering the constraints imposed by the counters.

Chapter 3: Learning HTN methods using Curricula

When HTN-MAKER creates HTN methods from observing plan traces to solve hierarchical planning problems, it often generates many extraneous methods. This issue can significantly impact the effectiveness and efficiency of HTN planning.

To address this problem, we adopt the concept of curriculum learning [2]. Research in this area has shown that learning performance often improves when training examples are ordered by problem difficulty. A curriculum can potentially enhance an HTN learner by teaching it to learn simpler methods first, before gradually progressing to more complex ones that build on previously learned methods.

In this chapter, we describe CURRICULEARN[1], an HTN learning algorithm that learns HTN methods through a curriculum-based approach. More specifically:

- We introduce CURRICULEARN, a new HTN learning algorithm that learns from curricula. Instead of examining every subsequence of a solution plan, CURRICULEARN examines only the subtraces that we tell it to examine in an order that we specify.
- We prove theoretically that the methods learned by CURRICULEARN can be used by a hierarchical planner to solve an HTN planning problem that is equivalent to the classical planning problem used to learn those methods.

[1] Portions of this work was published in the HPLAN workshop of ICAPS 2022 [32].

- We compare CURRICULEARN and HTN-MAKER on five benchmark planning domains: 1) moving a stack of blocks in the Blocks World domain, maintaining their order (which requires moving the stack twice); 2) delivering packages in the Logistics domain; 3) moving crates in the Depots domain; 4) taking images in the Satellite domain; and 5) flying passengers in the Zeno Travel domain. In our experiments, CURRICULEARN learns fewer methods that result in better planning performance than HTN-MAKER, and does so with less running time. A planner using the methods learned by CURRICULEARN solves more problems with a shorter runtime than with the methods learned by HTN-MAKER.

- We provide experimental results for the Logistics domain demonstrating further improvements when an ordering constraint is added to the annotated task.

3.1 CURRICULEARN: Learning HTN methods using Curricula

Suppose that we want to teach an HTN method learner how to solve some task τ. An ideal curriculum would focus the learner on the simplest subtasks of τ first, then build more and more complex subtasks until all of τ is learned. If the learner learns from plan traces, then the plan traces for the subtasks of τ will be subtraces of the plan trace that solves τ. More specifically, if π is a plan trace for τ, then the plan trace for each annotated subtask τ_i is a subtrace $\pi[b_i, e_i]$ of π. Thus we can represent our curriculum as a sequence of triples of the form (b_i, e_i, τ_i).

Definition 8. *Given a plan trace π, a* curriculum *C is a sequence of k curriculum steps of the form (b_i, e_i, τ_i), where b_i and e_i specify the starting and ending indices of the subtrace to analyze, and τ_i specifies the annotated task to learn from the subtrace for step $i \in \mathbb{N}_k$.*

For example, in the Blocks World domain, suppose that the task is to move a stack of two

blocks (A and B) from block C onto the table, while maintaining their order (i.e. block A is still on block B). Figure 3.1 shows a possible solution: first move A and B from above C onto the table while inverting the order (state s_0 to s_1), then invert the order of B and A back onto the table (state s_1 to s_2). Let π be the following solution trace for that task (each action's name starts with an exclamation mark):

Action 1: `(!Unstack A B)`
Action 2: `(!Putdown A)`
Action 3: `(!Unstack B C)`
Action 4: `(!Stack B A)`
Action 5: `(!Unstack B A)`
Action 6: `(!Putdown B)`
Action 7: `(!Pickup A)`
Action 8: `(!Stack A B)`

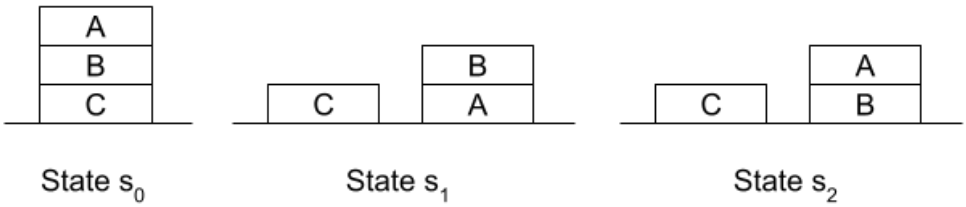


Figure 3.1: To move two blocks A and B from block C to the table while maintaining their order, the plan π inverts their order (state s_1), then inverts it again (state s_2).

Step	Begin	End	Annotated Task
a	1	2	`Make-1Pile`
b	3	4	`Make-2Pile`
c	1	4	`Make-2Pile`
d	5	6	`Make-1Pile`
e	7	8	`Make-2Pile`
f	5	8	`Make-2Pile`
g	1	8	`Make-2Pile`

Table 3.1: A curriculum for a Blocks World problem.

Let C be the curriculum shown in Table 3.1. The curriculum has 7 steps (a through g), each step consists of a subtrace of π determined by the beginning and ending indices and the name of an annotated task. The curriculum steps start with simpler tasks, which progressively combine

into harder ones. We can use the curriculum to ultimately teach how to do `Make-2Pile` in step g.

In principle, for a problem and its solution trace, we give a curriculum that is constructed from a tree that partitions the actions in the solution trace. Specifically, each terminal node of the tree corresponds to one action or a sequence of actions, and each internal node corresponds to the sequence of actions in the subtree below it. Also, we choose one annotated task for each node of the tree to construct a curriculum step. Figure 3.2 illustrates such a tree for the example Blocks World problem.

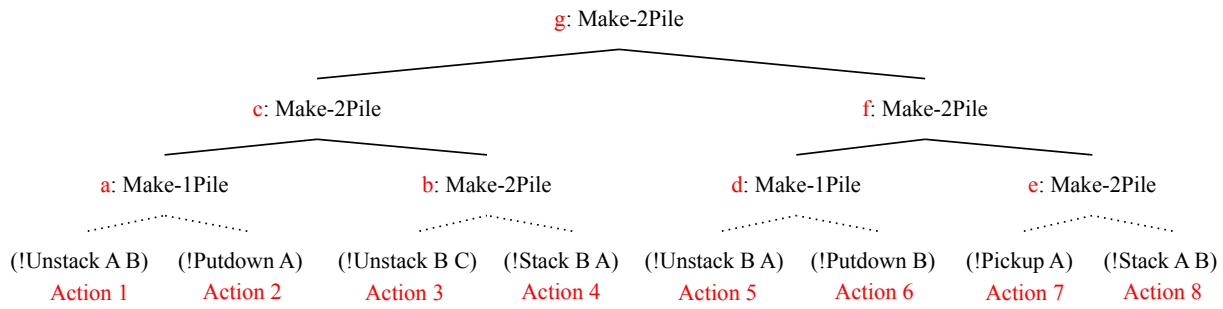


Figure 3.2: The illustration of the hierarchy of the curriculum steps over the 8-action solution trace for moving 2 blocks in the Blocks World domain.

As is shown in Algorithm 3, CURRICULEARN takes a curriculum C as one of the inputs and learns HTN methods according to the curriculum steps. The LEARN-METHOD subroutine (line 5) in CURRICULEARN is same as the one in HTN-MAKER. CURRICULEARN analyzes only those subtraces ordered by the curriculum, rather than analyzing every subsequence of π.

Algorithm 3 A high-level description of CURRICULEARN. The input includes a classical planning domain description D, an initial state s_0, a solution trace π, a curriculum C, and a possibly empty set of HTN methods M. The output is an updated set of HTN methods.

procedure CURRICULEARN(D, s_0, π, C, M)
    initialize $X \leftarrow \emptyset$
    let $\vec{S}$ be the state trajectory generated from $\gamma(s_0, \pi)$
    for $(b, e, \tau) \in C$ **do**
        $M' \leftarrow$ LEARN-METHOD($\pi, \vec{S}, \tau, X, b, e$)
        $M \leftarrow M \cup M'$
        for $m \in M'$ **do**
            $X \leftarrow X \cup \{(m^{head}, \tau^{eff+}, m^{sub}, m^{prec}, b, e)\}$
    return M

3.2 Theoretical Analysis

In appendix B, we prove that the methods we learn using CURRICULEARN are the ones that we want to learn. In this section, we show that theoretically CURRICULEARN can use the learned methods to solve the hierarchical planning problem that is equivalent to the classical planning problem used to learn those methods. We also compare the the number of subtraces that HTN-MAKER and CURRICULEARN each analyzes.

Proposition 1. *Let $P = (\Sigma, s_0, g)$ be a classical planning problem, $\tau = (\tau^{head}, \emptyset, g)$ be an annotated task. Let π be the solution trace for P, and C be a curriculum that has $(1, len(\pi), \tau)$ as its last step. Let M be the set of methods learned by* CURRICULEARN*, P_h be the hierarchical planning problem $((\Sigma, M), s_0, \langle\tau\rangle)$. Then π is a solution to P_h.*

Proof Sketch. If π is empty or g is satisfied in s_0, then CURRICULEARN will learn a method for τ that has empty subtasks, which is sufficient to solve the problem. Otherwise, CURRICULEARN

will learn at least one method from curriculum step $(1, len(\pi), \tau)$. This method must be applicable to s_0 because its preconditions were computed by regressing g through the actions of π [11, the LEARN-METHOD precedure], which is applicable to s_0. Furthermore, the goal regression procedure guarantees that whenever the preconditions of a method are satisfied there must be some way to reduce the subtasks of that method using other methods, because the subtasks of that method were chosen from the indexed instances of other methods. □

Proof. If the length of π is 0, then $g \subseteq s_0$, or π would not be a solution to P. Then CURRICULEARN will learn a method for τ that has empty subtasks and empty preconditions [11, the LEARN-METHOD precedure], which is sufficient to solve the problem. The method will be applicable to s_0, producing an empty task network. Thus, P_h can be solved.

If the length of π is 1, then it consists of an action a = (a^{head}, a^{prec}, a^{eff-}, a^{eff+}). The only step in the curriculum is (1, 1, τ). Thus, CURRICULEARN will learn a method m = (τ^{head}, $(g \backslash a^{eff+}) \cup a^{prec}$, $\langle a^{head} \rangle$). Each member of $g \backslash a^{eff+}$ must be true in s_0, because otherwise it would not be true in s_1. Each member of a^{prec} 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 into a, and then apply a to s_0 as a solution to $P[h]$.

Suppose that the length of π is n ($n > 1$), and that this lemma is true for all plans of length $n - 1$. Then HTN-Maker will learn a method m = (τ^{head}, m^{prec}, m^{sub}) from a call to LEARN-METHOD with b = 1, e = n, and τ, as well as possibly other methods from calls to Learn-Method with different parameters based on the other steps in the curriculum.

The method m learned from π has $m^{sub} = \langle t_0, t_1, ..., t_k \rangle$. If t_k is primitive and corresponds to the index action a = (a^{head}, a^{prec}, a^{eff-}, a^{eff+}, a^b, a^e), then a must be applicable to state

s_{a^b}. If a were not applicable to state s_0, then some earlier action in the plan 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_k is instead nonprimitive, then it corresponds to an indexed method instance x = $(x^{head}, x^{eff+}, x^{sub}, x^{prec}, x^b, x^e)$. 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 P'_h = (Σ_h, s_{xb}, x^{head}), i.e., the method $m' = (x^{head}, x^{prec}, x^{sub})$ of which x is an indexed instance must be applicable to s_{xb} and the execution of the solution plan to P'_h results in a state s_{xe}.

We know that $(g \backslash x^{eff+}) \cup x^{prec}$ is true in the state before action/method x, t_{k-1} is an action or non-primitive task that corresponds to $x' = (x'^{head}, x'^{eff+}, x'^{sub}, x'^{prec}, x'^b, x'^e)$. Then $(g \backslash x^{eff+}) \cup x^{prec}) \backslash x'^{eff+}$ must be true before action/method x' because otherwise $(g \backslash x^{eff+}) \cup x^{prec})$ would not be true on state s_{xb}. Also, x'^{prec} must be true in state $s_{x'b}$ according to the induction hypothesis. Therefore, $(g \backslash x^{eff+}) \cup x^{prec}) \backslash x'^{eff+} \cup x'^{prec}$ must be true in $s_{x'b}$. This process traces back to m'^{prec}, then we proved that m^{prec} is true in s_0. Therefore, m is applicable in s_0. The plans for solving $t_0, t_1, ..., t_k$ eventually make up π, which makes the state after applying m satisfy g. Thus, P_h can be solved using the learned methods. □

The unmodified HTN-MAKER has an analogous feature that the learned methods can solve the problem that is used to learn them [11, Lemma 4]. However, when considering how many methods are learned, it is easy to see by inspection that given a plan trace of length k it analyzes exactly $\frac{k(k+1)}{2}$ subtraces, which is $\Theta(k^2)$; see Algorithm 1, Lines 4, and 5.

In contrast, CURRICULEARN only needs to analyze $O(k)$ subtraces, assuming that a curriculum, as described in Section 3.1, is generated and provided to CURRICULEARN. More

specifically, suppose that a curriculum C is constructed from a tree in which each terminal node corresponds to a sequence of one or more actions in the solution trace, and each action in the solution trace appears in exactly one of the sequences. Then there are at most k terminal nodes. Suppose each internal node of the tree corresponds to the sequence of actions in the subtrees below it, and every internal node has at least two children. It is well known that in this case, the largest possible number of internal nodes is $k - 1$. Thus the number of steps in the curriculum, i.e., the total number of nodes in the tree, is at most $2k - 1$. Figure 3.2 shows an example tree.

The utility problem, as described by [33], arises when the cost of testing whether learned knowledge is applicable outweighs the benefits of learning more knowledge provides. In the context of learning HTN methods, HTN-MAKER often learns extraneous methods, which results in the utility problem. CURRICULEARN, on the other hand, may reduce the number of learned methods because it involves fewer learning steps compared to HTN-MAKER. Therefore, the next step is to test the effectiveness of the methods learned by CURRICULEARN.

3.3 Experimental Setup

To examine whether a curriculum can improve upon the methods learned by HTN-MAKER, we consider these two questions:

1. Does CURRICULEARN learn fewer methods with less computational effort than HTN-MAKER?

2. Do the methods learned by CURRICULEARN result in more efficient planning than HTN-MAKER when using the same HTN planner?

To assess these questions, we compare CURRICULEARN and HTN-MAKER in five planning

domains that were used in previous HTN-MAKER studies. These domains are HTN variants of the International Planning Competitions benchmarks and include: Blocks World, Logistics, Depots, Zeno Travel, and Satellite. For the first question about method learning, we evaluate the total number of learned methods and the time it took them to learn those methods. For the second question, we evaluate the performance of the same HTN planner, an implementation of the SHOP [18] planning algorithm provided for the original HTN-MAKER studies, in terms of the planning success rate and the planning runtime given the methods learned by HTN-MAKER and CURRICULEARN.

For each domain, we choose a class of problem types to learn HTN methods and evaluate the impact of using curricula. Also, for each domain, we do experiments in a range of problem sizes to provide sufficient evidence for the comparison. The following paragraphs describe each domain and the detailed methodology of our experiments.

Action 1: `(!Drive-To L1S)`
Action 2: `(!Load P1)`
Action 3: `(!Drive-To L1D)`
Action 4: `(!Unload P1)`
Action 5: `(!Drive-To L2S)`
Action 6: `(!Load P2)`
Action 7: `(!Drive-To L2D)`
Action 8: `(!Unload P2)`
Action 9: `(!Drive-To L3S)`
Action 10: `(!Load P3)`
Action 11: `(!Drive-To L3D)`
Action 12: `(!Unload P3)`

Table 3.2: A plan trace in the Logistics domain

Logistics. The objective in the Logistics domain is to deliver packages among locations in various cities, using trucks within cities and airplanes between cities. The class of problems we

studied in this domain involves delivering n (we used $2 \leq n \leq 12$) packages within the same city using a truck. For example, the plan trace shown in Table 3.2 effectively delivers three packages `P1`, `P2`, `P3` from locations `L1S`, `L2S`, `L3S` to `L1D`, `L2D`, `L3D` respectively within a city.

Step	Start	End	Annotated Task
a	1	4	`Deliver-1Pkg`
b	5	8	`Deliver-1Pkg`
c	1	8	`Deliver-2Pkg`
d	9	12	`Deliver-1Pkg`
e	1	12	`Deliver-3Pkg`

Table 3.3: A curriculum in the Logistics domain.

```
(:task
 :head Deliver-1Pkg
  (?o - obj
   ?d - location)
 :preconditions ()
 :goals
  ((at ?o1 ?d)))
```

```
(:task
 :head Deliver-2Pkg
  (?o1 - obj
   ?o2 - obj
   ?d - location)
 :preconditions ()
 :goals
  ((at ?o1 ?d)
   and (at ?o2 ?d)))
```

Figure 3.3: Example annotated tasks in the Logistics domain.

Table 3.3 shows a curriculum to teach how to learn methods from the plan trace. As before, each curriculum entry includes a subtrace of π_2 and an annotated task (see Figure 3.3) that the subtrace accomplishes:

Blocks World. The Blocks World domain includes a number of blocks sitting on a table (possibly on top of each other) and a robotic hand that can grasp one block at a time. The objective is to learn methods to move a stack of n (we used $2 \leq n \leq 25$) blocks on the top using the robotic hand, keeping the top-to-bottom order of the blocks the same as in the original stack.

```
(:task
 :head Make-1Pile
  (?a - block)
 :preconditions ()
 :goals
   ((on-table ?a)
    and (clear ?a)))
```

```
(:task
 :head Make-2Pile
  (?a - block
   ?b - block)
 :preconditions ()
 :goals
  ((on-table ?b)
   and (on ?a ?b)
   and (clear ?a)))
```

Figure 3.4: Example annotated tasks in Blocks World.

Example annotated tasks used in this domain are shown in Figure 3.4. An example problem and the corresponding curriculum are demonstrated in Table 3.1.

Depots. The Depots domain is a combination of the Logistics domain and the Blocks World domain. The logistics element of the task is to move crates from one depot to another using trucks. The Blocks World element arises because of the need to stack and unstack crates. The class of problems we choose in this domain involves delivering n (we used $2 \leq n \leq 25$) crates from different depots to one destination depot and stack them in order.

Satellite. The Satellite domain models the problem of an array of satellites collecting a variety of images using observation instruments each of a specific mode and direction. The class of problems in this domain has a satellite take n (we used $2 \leq n \leq 12$) images in different directions.

Zeno Travel. In the Zeno Travel domain, passengers are transported between cities by an aircraft that consumes fuel and can be refueled. Standard predicate logic is used to encode the relationships between the integers zero through five to represent fuel levels. The class of problems we choose in this domain is to fly n (we used $2 \leq n \leq 12$) passengers from various cities

to a destination city.

For each domain (and the chosen class of problems) and for each problem size, we randomly generated 50 problems. Then we compared the average number of methods learned as well as the time (ms) taken by HTN-MAKER and CURRICULEARN. After that, we evaluate the methods by using them to solve the hierarchical planning problem that is equivalent to the classical planning problem used to learn those methods. For this we used an HTN planner: HTN-MAKER's implementation of the SHOP [18] planning algorithm. Finally, we compared the planning success rate as well as the running time (ms). For each configuration of learning approaches (HTN-MAKER or CURRICULEARN), for each stage of experiments (method learning or planning with the learned methods), for each problem domain among the five domains, as well as for each of the 50 randomly generated problem in that domain, we allowed a limit of 10 minutes of running time.

3.4 Results and Discussion

The results of our experiments, plotted in Figures 3.5 and Figure 3.9, show positive answers to the two questions raised in the beginning of the last section. We will explain and discuss the experiment results in this section.

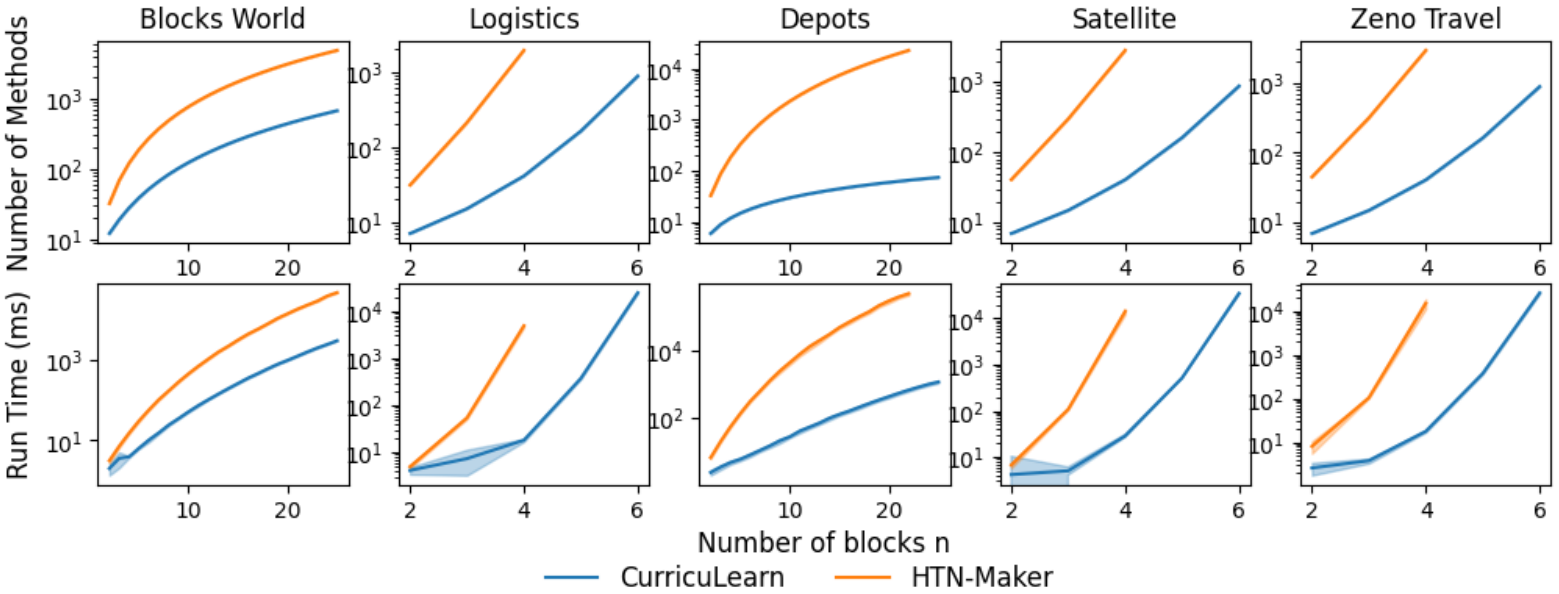


Figure 3.5: These semi-log plots show the average number of methods learned and the running time taken by HTN-MAKER (red color) and CURRICULEARN (blue color) in five domains. Each data point is the average of 50 randomly generated problems of size n ($2 \leq n \leq 25$), where n is the total number of blocks in the problem. On some of the larger problems, no results are shown because our 10-minute time limit was exceeded before those problems yield any results.

3.4.1 CURRICULEARN Learns Fewer Methods in Less Time

CURRICULEARN learns significantly fewer methods in less time than HTN-MAKER. Figure 3.5 demonstrates several trends that make this clear; note that this is a semi-log plot.

Blocks World and Depots. The results in the Blocks World domain and Depots domain are analogous. We set the maximum number of blocks in the Blocks World domain and the maximum number of crates in the depots domain to be 25. Both HTN-MAKER and CURRICULEARN were able to successfully learn methods for all of the problem sizes in the Blocks World domain. CURRICULEARN was able to solve all problem sizes in the range we provided, and it appears we could continue to much larger problems for some domains. However, HTN-MAKER learned more methods, took a longer time, and failed to learn methods for the problems of size greater than 22 in the Depots domain due to the time limit.

To have a clearer understanding of the difference of HTN-MAKER and CURRICULEARN, we examined the methods they learned from the example problem in the Blocks World domain described previously (Figure 3.1).

The HTN-MAKER code distribution[2] tested in the evaluations caused the algorithm to make choices when a method's decompositions is being learned from right to left, such that the right subtask (if any) always corresponded to a previously learned method instance that was indexed over the largest subtrace (if there were multiple qualifying method instances). As a result, HTN-MAKER learned a method in its final step that had `((!Unstack ?a ?b) (Make-2Pile ?b ?a))` as subtasks, where the subtask `(Make-2Pile ?b ?a)` could be right-recursively decomposed into the remaining 7 out of the total 8 actions in the solution trace.

[2]Different from its implementation, the HTN-MAKER pseudocode [11] "nondeterministically" chooses subtask groupings to form methods when there are several possibilities.

In comparison, CURRICULEARN learns `M1` (Figure 3.6) from the final curriculum step. Method `M1` has `((Make-2Pile ?a ?b) (Make-2Pile ?b ?a))` as subtasks. The first subtask effectively moves a stack of blocks `?b` and `?a` from above `?c` onto the table while inverting the order. The second subtask effectively moves the stack of blocks `?a` and `?b` from the table back onto the table while inverting the order again. The method `M1` is semantically more readable as the decomposition of which is how we want to solve the problem — we make a pile of blocks where `?a` is on `?b`, then we make a pile of block where `?b` is on `?a`.

```
(:method M1
 :head Make-2Pile
   (?b - block ?a - block)
 :vars (?c - block)
 :precondition ( (not (= ?b ?a)) (on ?b ?c) (on ?a ?b)
  (not (= ?a ?c)) (not (= ?b ?c)) (clear ?a) (hand-empty))
 :subtasks ( (Make-2Pile ?a ?b) (Make-2Pile ?b ?a) ) )
```

Figure 3.6: The last method learned for the example Blocks World problem demonstrated in Section 3.1.

Logistics, Satellite, and Zeno Travel. The results in these three domains look similar to each other because the classes of problems we chose in the Satellite domain, Zeno Travel domain, and Logistics domain shared similar structure: we need to move objects from different locations to one destination in all three domains (see Section 3.3 for experiment setup). We observe an exponential growth of the number of learned methods in each of the three domains for both CURRICULEARN and HTN-MAKER (Figure 3.5). This is caused by the existence of symmetric ways to bind variable names to object names when learning methods for a given annotated task with lifted variable names.

More specifically, suppose HTN-MAKER learns methods in the Logistics domain for the task `Deliver-2Pkg` for a solution plan that first delivers package `A` then package `B` to the destination. For such a case, there could be four possible bindings to the variable names in the annotated task: 1) `o1` to `A` and `o2` to `B`, 2) `o1` to `B` and `o2` to `A`, or 3) both `o1` and `o2` to `A`, and 4) both `o1` and `o2` to `B`. Cases 1 and 2 are equivalent, while cases 3 and 4 are identical and undesireable because they deliver the same package "twice". But in the absence of guidance about the bindings, the LEARN-METHOD procedure will learn four methods. As a result, the number of learned methods grows exponentially in the problem size and number of variable names in the annotated tasks.

One way to overcome this issue is to include an ordering constraint in the precondition of the annotated task that forces an unambiguous order among the variable names in the precondition (see annotated task `Deliver-2Pkg-Prec` in Figure 3.7 as an example). Such a precondition removes symmetric ambiguity during the variable name binding process, resulting in a large reduction in the number of methods learned.

```
(:task
 :head Deliver-2Pkg-Prec
   (?o1 - obj ?o2 - obj ?d - location)
 :preconditions ( before ?o1 ?o2 )
 :goals ((at ?o1 ?d) (at ?o2 ?d))
```

Figure 3.7: Example annotated tasks with preconditions in the Logistics domain that avoids the symmetry problem potentially caused by ambiguous variable name bindings.

To assess the impact of symmetric orderings like these, we performed another set of experiments in the Logistics domain by adding the ordering constraint in the precondition. The results in Figure 3.8 validates our hypothesis because the `+before-precondition` experiments learn

substantially fewer methods in less time. The effect is so pronounced that the ordering precondition extends the reach of HTN-MAKER and CURRICULEARN to larger problems. However, the time it takes to learn methods for the annotated tasks with "before" preconditions is still exponential; we suspect that it happens because the computational bottleneck is on generating and verifying alternative name bindings. Regardless of whether the symmetry problem is resolved, Figure 3.8 demonstrates that CURRICULEARN always learns fewer methods than HTN-MAKER using less time.

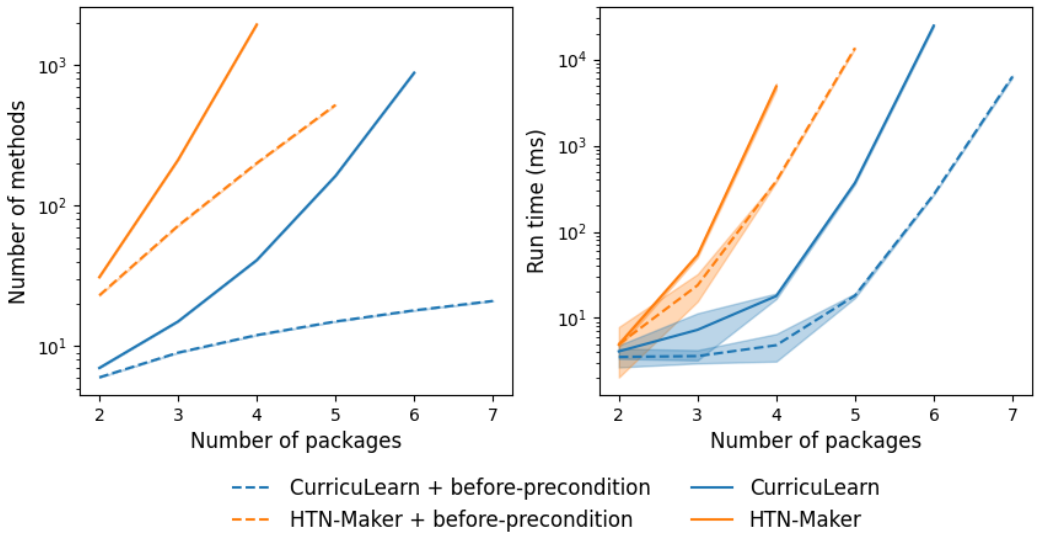


Figure 3.8: The average number of methods learned and the running time taken by CURRICULEARN (blue color) and HTN-MAKER (red color) in the Logistics domain using annotated tasks with (circles and squares) and without (dots and crosses) the "before" preconditions that resolves the symmetry problem. Each data point is the average of 50 randomly generated problems. On some of the larger problems, no results are shown because our 10-minute time limit was exceeded before those problems yield any results.

In all these three domains, neither HTN-MAKER nor CURRICULEARN successfully learned methods for problems larger than a certain size because of the computational bottleneck described above. However, compared to HTN-MAKER, CURRICULEARN always learned methods

for problems in all these three domains of larger sizes and always learned fewer methods in less time if they both learned some methods.

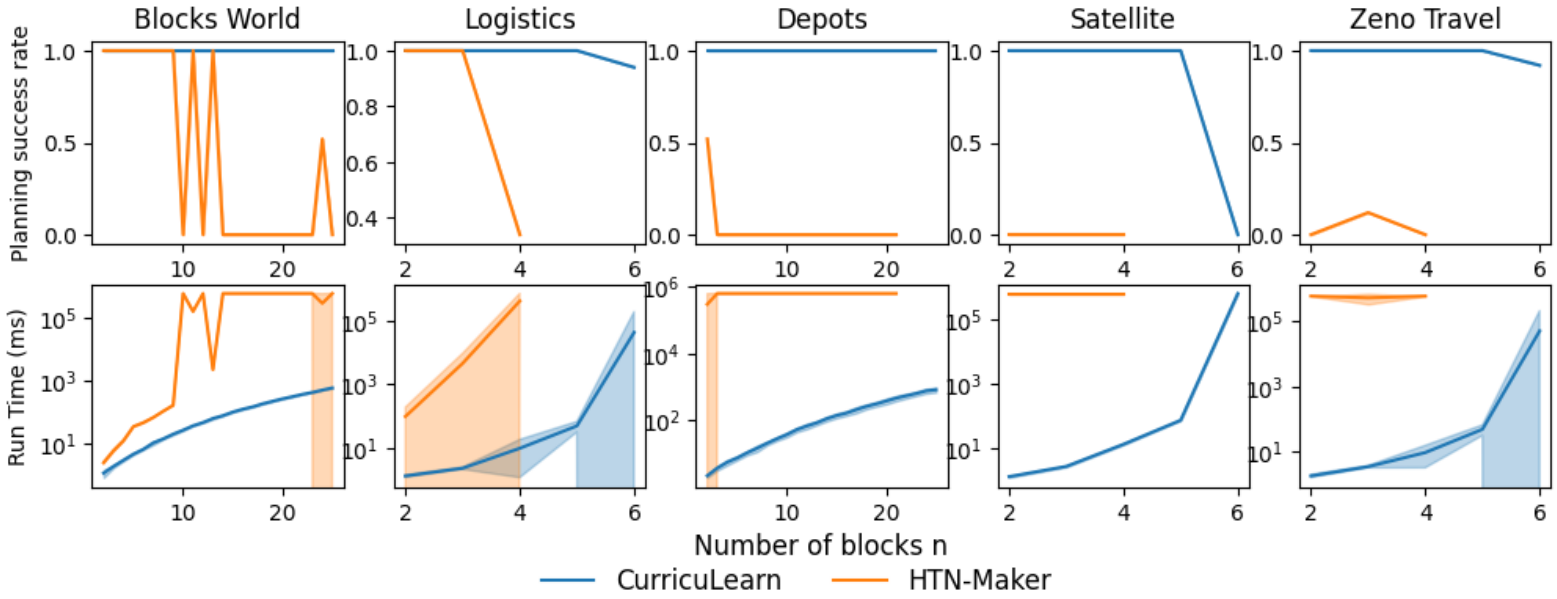


Figure 3.9: The plots show the HTN planner's success rate (linear y axis) and average running time (semi-log y axis) with variance in using the learned methods to solve the hierarchical planning problems that are equivalent to the classical planning problems used to learn those methods. Each data point is obtained from solving 50 hierarchical planning problems.

3.4.2 CURRICULEARN Results in Better Planning

The methods learned by CURRICULEARN result in greater success in planning and faster planning for problems when they are solved. Figure 3.9 shows that the HTN planner takes significantly less time to solve the hierarchical planning problem with a significantly higher success rate using the methods learned by CURRICULEARN. The HTN planner fails to find solutions sooner using the methods learned by HTN-MAKER than CURRICULEARN as the problem size increases because HTN-MAKER learns many more methods.

For example, the evaluation results show that the methods learned by CURRICULEARN can successfully solve all the hierarchical planning problems of any given sizes in the Blocks World domain and Depots domain, while the methods learned by HTN-MAKER can only solve the same problems of small sizes due to time limit.

In the Logistics domain, Satellite domain, and Zeno Travel domain, the evaluation results show that the methods learned by CURRICULEARN can successfully solve most of the hierarchical planning problems until the problems get too large to finish solving within the time limits. However, the methods learned by HTN-MAKER fail to solve most of the problems, especially when they become larger due to time limit.

The very noticeable vertical lines in the plot are error bars that indicate large variance, as the planner sometimes fails very quickly and sometimes runs for almost 10 minutes (time limit) on those data points.

3.5 Summary

We have described CURRICULEARN, an HTN learning algorithm that learns HTN methods using a curriculum. We theoretically proved that the methods learned by CURRICULEARN can be used by a hierarchical planner to solve the hierarchical planning problem that is equivalent to the classical planning problem used to learn those methods. Our experiments in the chosen classes of problems in the Blocks World domain, the Logistics domain, the Depots domain, the Satellite domain and the Zeno Travel domain showed that CURRICULEARN learns significantly fewer methods that result in better planning performance with less computational effort. We have also provided preliminary results for the Logistics domain demonstrating further improvements when an ordering constraint is added to the annotated task.

Chapter 4: Learning HTN Methods with Curricula Automatically Generated from Landmarks

We have demonstrated that using curricula to guide HTN learning offers significant advantages, particularly as the problem size increases. However, the creation of such curricula still requires problem-specific expertise. Because of this limitation, our evaluation of CURRICULEARN in the previous chapter was confined to certain types of problems where we could easily write rule-based programs to generate curricula. In this chapter, we present an approach that fully automates the curricula generation process using landmarks.

Landmarks [16] are facts that must appear in every solution to a planning problem. In the context of learning hierarchical knowledge, methods that achieve landmarks provide a backbone for solving a planning problem. More critically, landmarks also provide a natural way to structure methods automatically.

In this chapter, we describe an approach for building curricula to learn methods that achieve landmarks. This includes the following contributions[1]:

- We introduce CURRICULAMA, which uses landmarks to generate curricula and then utilizes CURRICULEARN to learn HTN methods from those curricula. This approach obviates HTN-MAKER's need for manual annotation of tasks.

[1]Portions of this work was published on FLAIRS 2024 [34].

- We prove that the methods learned by CURRICULAMA can be used by a hierarchical planner to solve an HTN planning problem that is equivalent to the classical planning problem from which the methods were learned.
- Our experimental results show that CURRICULAMA has a similar convergence rate to HTN-MAKER in learning a complete set of methods to solve all the test problems.

4.1 CURRICULAMA

Since a landmark must be true at some point in every solution to a planning problem, we hypothesized that it would be useful to learn methods that reach landmarks. Our algorithm, CURRICULAMA (Algorithm 4) extracts landmarks from a planning problem, generates a curriculum from those landmarks, then utilizes CURRICULEARN to learn methods according to the curriculum.

Algorithm 4 CURRICULAMA: HTN learning using Curriculum Generated from Landmarks

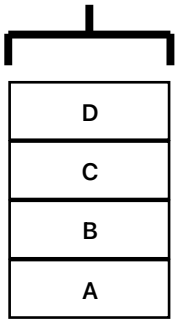
Input: a classical planning problem P

Output: a set of method M

1: $(\Sigma, s_0, g) \leftarrow P$
2: $(V, E_V) \leftarrow$ extract landmark graph from (Σ, s_0, g)
3: add reasonable orders to (V, E_V)
4: $C \leftarrow \langle\rangle$ ▷ initialize the curriculum steps
5: $\pi \leftarrow \langle\rangle$ ▷ initialize the plan trace
6: $s \leftarrow s_0$ ▷ initialize the current state
7: $i \leftarrow 0$ ▷ initialize the plan length
8: **while** $V \neq \emptyset$ **do**
9: select and remove a vertex v in (V, E_V) that has no predecessors
10: $\pi' \leftarrow$ CLASSICALPLANNER(Σ, s, v)
11: $s \leftarrow \gamma(s, \pi)$
12: $i \leftarrow i + length(\pi')$
13: concatenate π' to π
14: $t \leftarrow$ MAKEANNOTATEDTASK(v)
15: **for** k from i to 1 **do**
16: append (k, i, t) to C
17: $M \leftarrow \langle\rangle$
18: $M =$ CURRICULEARN(Σ, s_0, π, C, M)
19: **return** M

CURRICULAMA takes as input a classical planning problem. It first generates a landmark graph for P using h^m Landmarks[2] (Line 2) from the landmark generation method introduced by Keyder et al. [35]; then it adds reasonable orders to the landmark graph (Line 3) from the method described by Hoffmann et al. [16]; and then it iterates through the landmarks by their orderings. For each landmark, CURRICULAMA iteratively obtains a solution trace from the current state to

[2]We use the implementation of h^m landmark generation and reasonable order extraction in the Fast Downward planning system (https://www.fast-downward.org/), configured to only allow for single-atom (conjunctive) landmarks.



Action 1: `(!Unstack D C)`
Action 2: `(!Putdown D)`
Action 3: `(!Unstack C B)`
Action 4: `(!Putdown C)`
Action 5: `(!Unstack B A)`

Figure 4.1: A Blocks World problem in which the initial state is a stack of 4 blocks. The goal is to make the bottom block A clear. The plan to achieve the goal is shown on the right.

the next landmark using a classical planner and updates the current state by applying the solution plan (Lines 10 and 11). MAKEANNOTATEDTASK (Line 14) takes as input the current landmark and produces an annotated task that has a task name, empty preconditions, and the landmark as its goals. Given the annotated task produced from the landmark, CURRICULAMA generates curriculum steps (Lines 15 and 16) that progressively trace backward to the beginning of the plan to learn methods. Then, CURRICULAMA utilizes CURRICULEARN to acquire HTN methods according to these curriculum steps (line 18) This whole process obviates HTN-MAKER's need for manual annotation of tasks and corresponding plan subtraces.

Example. Consider a Blocks World classical planning problem in which four blocks `A`, `B`, `C`, and `D` stacked on each other and the goal is to have block `A`'s top clear. Formally: initial state s_0 = {`(on-table A), (on B A), (on C B), (on D C), (clear D), (hand-empty)`}; and goal `g` = {`(clear A)`}. A possible solution π_{clearA} is to remove the blocks above `A` one by one through action 1 to 5. see Figure 4.1 for the initial state demonstration and the solution plan.

Excluding the initial state, Figure 4.2 shows the landmark graph from CURRICULAMA for (s_0, g), which consists of 3 landmarks: `(clear C)` $\prec$ `(clear B)` $\prec$ `(clear A)`.

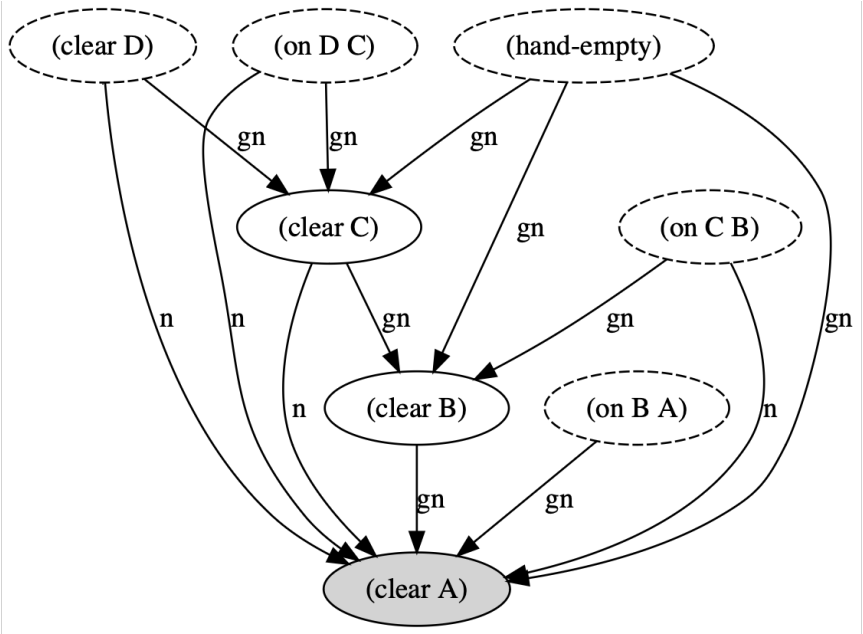


Figure 4.2: A landmark graph for clearing block `A` from blocks `B`, `C` and `D` above in the Blocks World domain. The circled nodes are landmarks, where the dashed nodes are the landmarks that are satisfied in the initial state, and the filled node is the goal. The edges are orderings among the landmarks, where '`gn`' stands for greedy necessary ordering, and '`n`' stands for natural ordering.

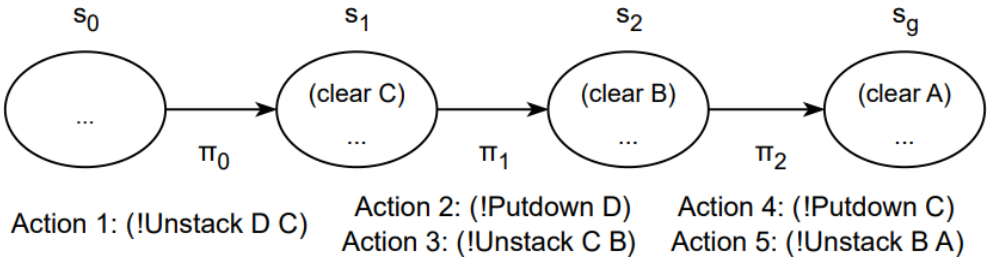


Figure 4.3: The subplans generated from the landmarks.

CURRICULAMA generates subplans to achieve the first, second and the third landmarks (Figure 4.3). For each landmark, it creates an annotated task (in this case, `Make-Clear`) and curriculum steps in which the final indices correspond to the action that achieves the landmark, and the beginning indices go back to the plan trace's start.

The curriculum shown in Table 4.1 comprises nine steps, labeled from a to i. Each step is defined by a specific segment of π, delineated by its beginning and ending indices, along with the

Step	Begin	End	Annotated Task
a	1	1	`Make-Clear`
b	3	3	`Make-Clear`
c	2	3	`Make-Clear`
d	1	3	`Make-Clear`
e	5	5	`Make-Clear`
f	4	5	`Make-Clear`
g	3	5	`Make-Clear`
h	2	5	`Make-Clear`
i	1	5	`Make-Clear`

Table 4.1: A curriculum generated by CURRICULAMA for a Blocks World problem.

name of an annotated task.

We want step a to learn a method m_1 for annotated tasks `Make-Clear` from the first action in plan P, this method tries to clear a block under one block. Then we want step b to learn a method m_2 for annotated tasks `Make-Clear` from the first to the third action in plan P, this method tries to clear a block under two blocks, and would presumably contain a subtask `Make-Clear` that is related to m_1 previously learned from the first action. Then step c learns a new method m_3 for `Make-Clear` that subsumes m_2. So on and so forth.

In general, the curriculum is structured to initiate simpler tasks, gradually progressing to more complex ones. The regressive sequencing of the beginning indices aims at learning methods with varying preconditions for the same annotated tasks.

4.2 Theoretical Analysis

We prove that the methods learned by CURRICULAMA from a classical planning problem can be applied to solve the equivalent hierarchical problem.

Proposition 2. *Let $P = (\Sigma, s_0, g)$ be a classical planning problem, M be the set of methods*

learned from P by CURRICULAMA*, τ be an annotated task that has g as its goals, and P_h be the hierarchical planning problem $((\Sigma, M), s_0, \langle\tau\rangle)$. Then π is a solution to P_h.*

Proof. Given a classical planning problem $P = (\Sigma, s_0, g)$ as a training example, CURRICULAMA produces a solution trace π and a curriculum C. Since g has to be the final landmark in the landmark graph of P, the final curriculum step in C is $(1, len(\pi), \tau)$. Given π, C, and an empty set of HTN methods M, CURRICULEARN will learn some methods. According to Proposition 1, π is a solution to P_h (which is equivalent to the classical planning problem P). □

Therefore, CURRICULAMA is sound for the original problem for which it learned methods. That is, methods learned by CURRICULAMA from a classical planning problem P will solve the equivalent hierarchical problem P_h. However, we also want to know how rapidly it can learn a complete set of methods from the training problems. In the next section, we will empirically evaluate CURRICULAMA to show that it has a comparable convergence rate to HTN-MAKER in learning a complete set of methods.

4.3 Empirical Study

We compared CURRICULAMA to HTN-MAKER experimentally in five IPC (International Planning Competition) domains: Logistics, Blocks World, Rover, Satellite, and Zeno Travel. These domains are used for evaluation in the original papers on HTN-Maker. We assess the efficiency of CURRICULAMA in learning Hierarchical Task Network methods and the effectiveness in solving hierarchical planning problems using the learned methods.

Our evaluation considered how well the methods learned from an incremental set of training problems can solve a set of static testing problems. A single trial for a domain used PDDL-

Generators [36] to generate 150 random classical problems for training and 50 hierarchical problems for testing. Both sets were derived from the same distribution of parameters, with the key difference being that classical problems have goals, while hierarchical problems have tasks. This consistent distribution of parameters ensures that the sizes of the training classical problems and the testing hierarchical problems are comparable. Starting with an empty set of methods M, the procedure iterated through the training problems $(1, 2, .., 150)$, augmented M using either CURRICULAMA or HTN-MAKER, and used HTN-MAKER's version of the SHOP planner [37] to solve the 50 test problems with the current set M. We repeated five trials in each of the five domains and reported on the following metrics:

- Convergence: We measured the convergence rate of the proportion of test problems the planner could successfully solve with the methods learned by each HTN learner. If the methods learned from only a few examples are sufficient to solve most of the testing problems, we say that the set of methods rapidly converges to one that is complete.

- Average Plan Length: We measured the average length of the plans for the successfully solved test problems, which informs us about the efficiency and complexity of the solutions generated by the learning algorithms.

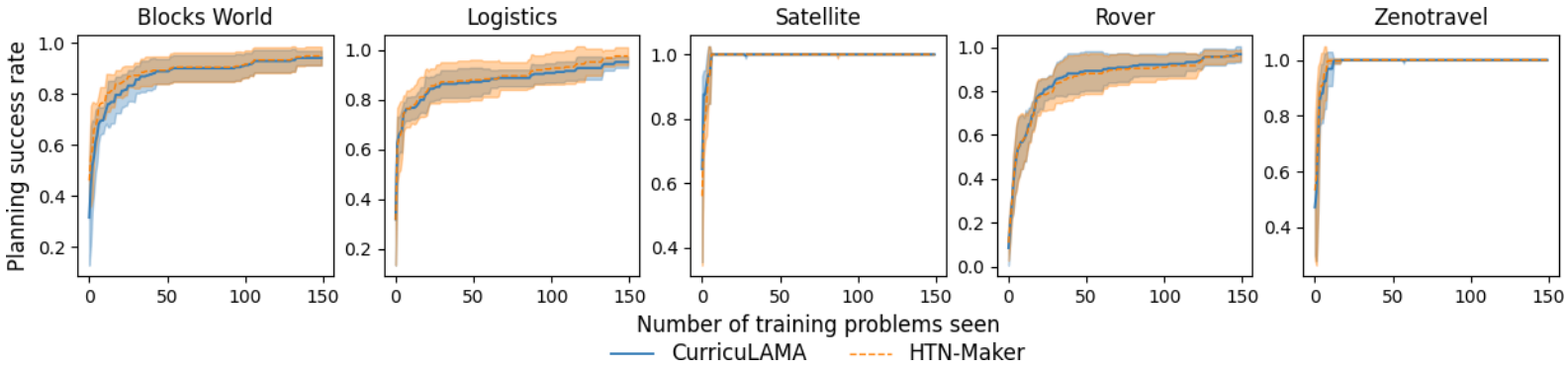
- Average Planning time: We measured the average amount of time that the planner needed to solve the test problems using the learned methods.

- Method Generation: We recorded the cumulative number of methods generated by each of the HTN learners as they saw more and more training examples.

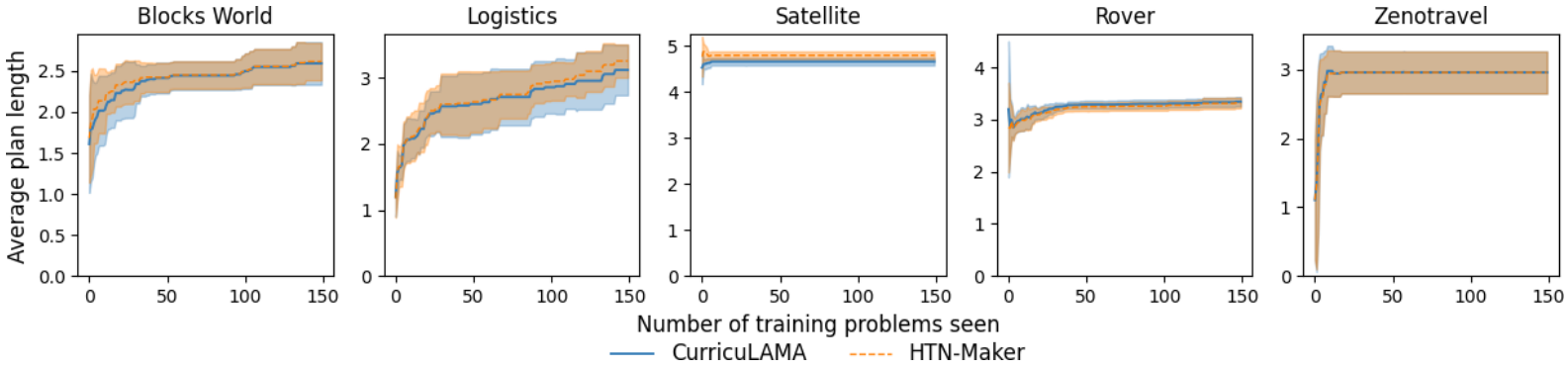
- Running Time: We compared the running time of the HTN learners at different stages of

the learning process.

All experiments were run on an AMD EPYC 7763 (2.45 GHz).



(a) Convergence analysis

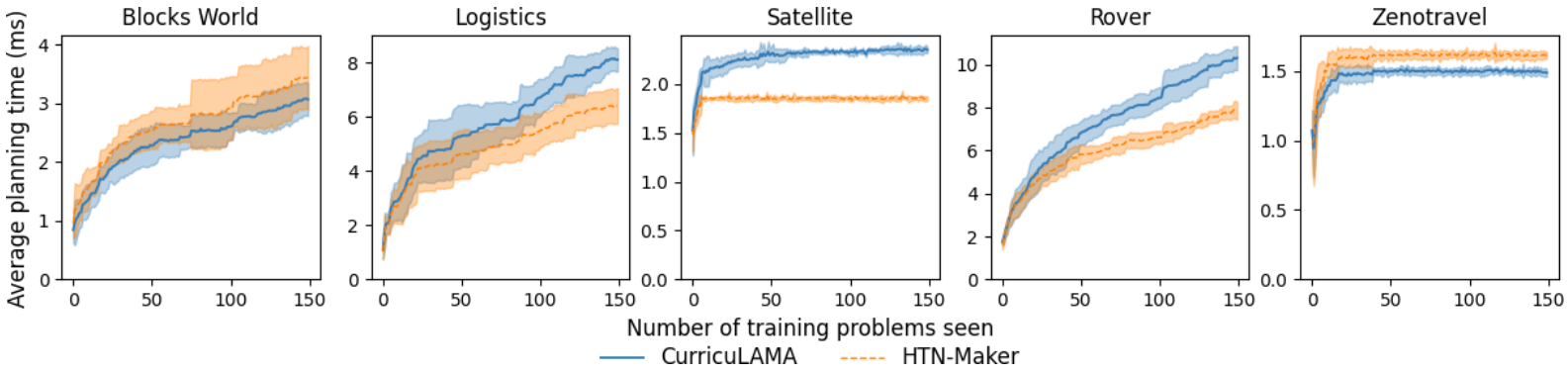


(b) Average plan lengths

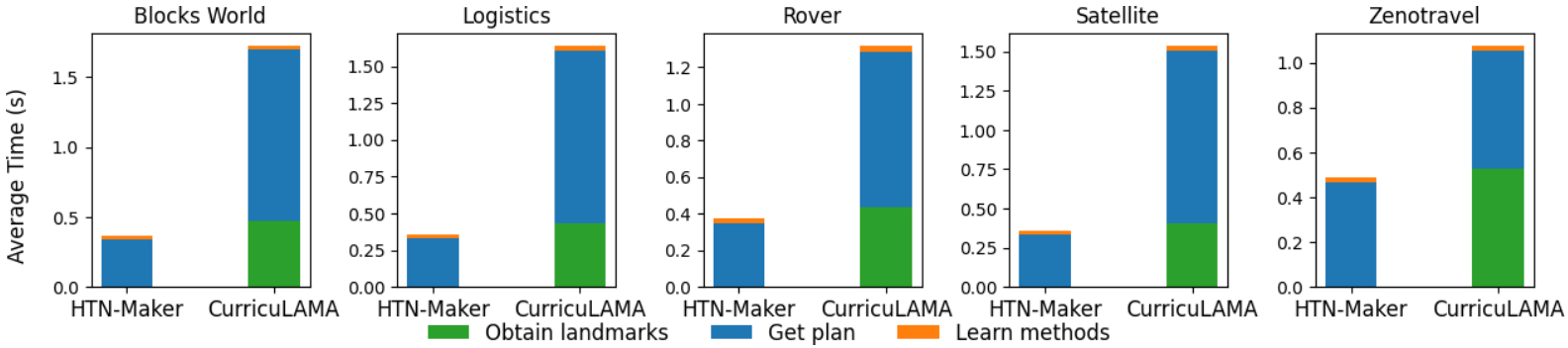
Figure 4.4: (a) The y-axis shows the fraction of problems that the planner could successfully solve using the methods learned by each learning algorithm, and the x-axis shows the number of training problems from which the methods were learned. The shaded regions show the variance in problems solved across five separate trials. (b) The y-axis shows the average length of the plans that the planner produced using the learned methods, and the x-axis shows the number of training problems from which the methods were learned, ranging from zero to 150 training problems. The shaded regions show the variance in plan length across five separate trials.

Convergence Rate. Figure 4.4a shows that the average planning success rate in the test set given the methods learned by each of the HTN learners converges to 100% as the number of training problems seen and the number of methods learned increases. We observe that there are no significant differences in the planning success rate between CURRICULAMA and HTN-MAKER.

Plan Length. The average plan lengths generated by using methods learned by CURRICULAMA and HTN-MAKER are depicted in Figure 4.4b. Both sets of methods produce plans of similar length. The upward trend in plan length for both algorithms suggests that as the number of training samples increases, the methods produced by CURRICULAMA and HTN-MAKER enable the planner to solve more difficult problems that had longer solutions.



(a) Time to solve planning problems using the learned methods.



(b) Running time needed to learn methods.

Figure 4.5: (a) The x-axis gives the number of training problems from which each learning algorithm learned its methods, and the y-axis gives the average planning time over the 50 test problems. The shaded regions denote the variance in planning time across five separate trials. (b) The bars represent the average time that each learning algorithm spent on different parts of the learning process. Green represents the time to obtain landmarks, blue indicates the time to get the plan, and orange shows the time to learn methods.

Planning Time. The average time needed for the planner to solve the testing problems using methods learned by CURRICULAMA and HTN-MAKER is shown in Figure 4.5a. We can observe that the planning time for CURRICULAMA is slightly lower in the Blocks World and Zenotravel domain, but is slightly higher in the other domains compared to HTN-MAKER. Overall, the difference in the average planning time between CURRICULAMA and HTN-MAKER is small, just a few milliseconds.

Running Time. Given a training problem, CURRICULAMA (1) extracts the landmarks, (2) gets pieces of sub-plans by iteratively achieving the landmarks using a classical planner, and (3) learns methods from the concatenated plan trace using a curriculum; while HTN-MAKER (1) gets a plan trace using the same classical planner, and (2) learns methods from the plan trace. Figure 4.5b shows the running time for those phases of each algorithm.

On average, CURRICULAMA incurs a higher cost in terms of running time for learning methods from a training problem compared to HTN-MAKER for its additional step of generating a plan trace aligned with the obtained landmarks. While the time difference is significant, it does not render CURRICULAMA impractical. Notably, the extra time expended by CURRICULAMA is offset by the fact that it *completely eliminates* the need for manual annotations by a domain engineer, which is a substantial advantage in the overall workflow.

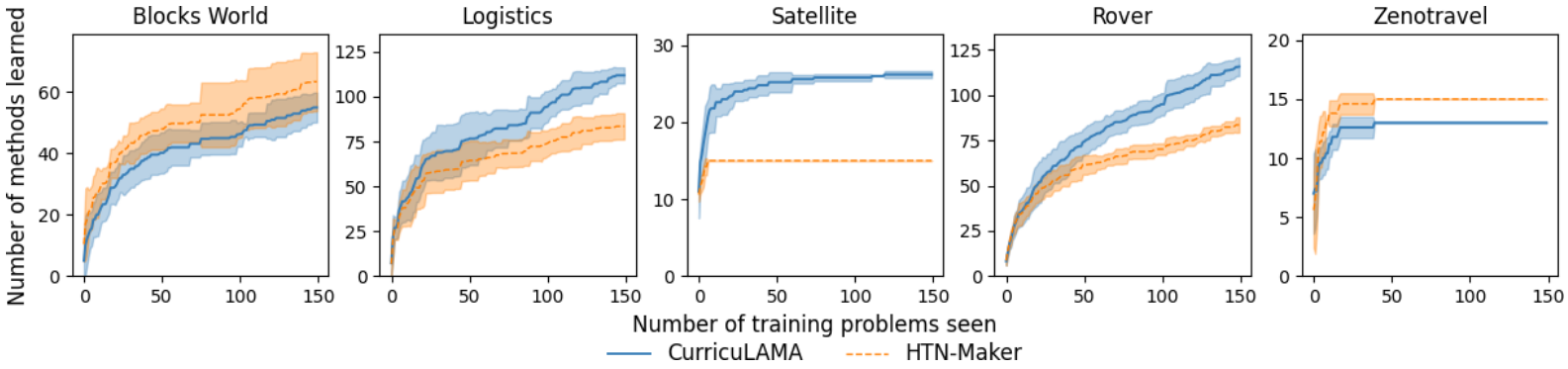


Figure 4.6: *Number of methods learned.* The y-axis shows the total number of methods learned, and the x-axis shows the number of training problems from which they were learned. Both algorithms show increases in the number of methods as they are exposed to more training problems. The shaded areas indicate the variance in the number of methods learned across five trials.

Method Generation. The average number of methods learned by CURRICULAMA and HTN-MAKER is shown in Figure 4.6. In the Blocks World and Zeno Travel domains, CURRICULAMA tends to learn fewer methods than HTN-MAKER, and in the Logistics, Satellite, and Rover domains the reverse is true. This result is correlated with the planning time result, where we can observe that the larger the number of methods learned during training, the larger the time needed by the planner during testing.

CURRICULAMA's planning mechanism may cause it to learn extraneous methods in some domains (*e.g.*, the Logistics, Satellite and Rover domain). While it's possible that this may be an indication of overfitting, we believe this is more likely a result of the partial orders in the landmark graph. The landmark generation algorithms used by CURRICULAMA (Algorithm 4, lines 2 and 3) return only a partial ordering among landmarks given the additional reasonable orders. All reasonable orderings are not determined because determining whether a reasonable order exists between two given landmarks is a PSPACE-complete problem [16]. Thus CURRICULAMA enforces a total ordering to formulate a sequence of subgoals, which is not necessarily the optimal strategy. This often results in CURRICULAMA's derivation of additional methods from extended plan traces, as those methods cover more potential (and sub-optimal) paths to the goal.

For illustrative purposes, consider delivering a package `p0` from location `l2-0` to location `l0-0` in the Logistics domain. The airplane a0 is initially at location `l1-0`. The most efficient strategy after flying the airplane a0 from `l1-0` to `l2-0` (according to the first landmark `(airplane-at a0 l2-0)`) would be to load the package into the airplane followed by a direct flight to `l0-0`. Nonetheless, as depicted in Figure 4.7, CURRICULAMA may adopt a sequence where the airplane first relocates to `l0-0` without the package, resulting in a suboptimal path and thus an increase in the number of methods learned.

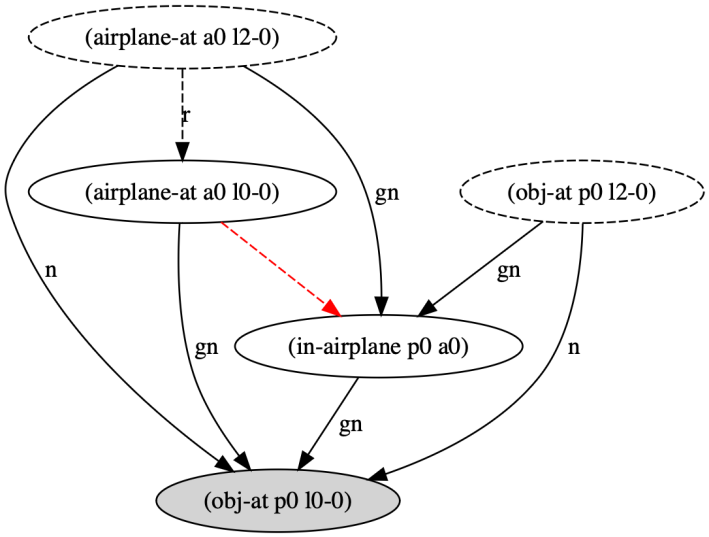


Figure 4.7: Landmark graph illustrating potential suboptimal planning in CURRICULAMA. Landmarks are represented by circles, and edges indicate ordering: '`r`' for reasonable, '`n`' for natural, and '`gn`' for greedy necessary. A missing reasonable ordering would prioritize `(in-airplane p0 a0)` before `(airplane-at a0 l0-0)` to avoid unnecessary airplane movements (marked by the red dashed arrow).

While CURRICULAMA may learn slightly more methods in some domains due to suboptimal landmark orderings, this does not appear to have a detrimental impact on planning success rates or plan lengths. Notably, in domains where CURRICULAMA successfully captures all required landmark orderings, it learns fewer methods than HTN-MAKER, which results in relatively shorter planning time.

4.4 Summary

Given a classical planning problem, CURRICULAMA generates curricula from landmarks and uses them to acquire HTN methods according to these curricula. We have proved that the methods that CURRICULAMA learns from classical planning problems enable an HTN planner to solve equivalent HTN planning problems. In our experiments CURRICULAMA learned com-

parably competent methods to those learned by HTN-MAKER for the same problems, with no requirement for a human to provide methods, curricula, or annotations of tasks. The idea that landmarks are useful for structural knowledge learning, and that curricula can be constructed from those landmarks, may apply to other structural knowledge learning techniques.

Here are some valuable future work directions on this topic:

- Improving CURRICULAMA's strategy for ordering landmarks by incorporating more sophisticated heuristics could help reduce the creation of these redundant methods.

- The number of methods and planning time keep increasing without convergence for both algorithms in the Blocks World, Logistics and Rover domain. We believe that given enough training problems, they will eventually converge. To verify that, we will expand the training set in our future experiments.

- We are also interested in an empirical study that compares manually annotated task and automatically annotated tasks when directly applied to HTN-MAKER without any curricula. This will give an idea of the quality of the tasks generated by CURRICULAMA.

- Last but not least, we will theoretically and empirically analyse CURRICULAMA's time complexity as some measure of task domain problem or solution complexity increases.

```
(:method M2
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block )
 :preconditions { ( on ?b2 ?b1 ) ( clear ?b2 ) }
 :subtasks < ( !Unstack ?b2 ?b1 ) > )


(:method M3
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block ?b3 - block )
 :preconditions { ( on ?b2 ?b1 ) ( holding ?b3 ) ( clear ?b2
    ) }
 :subtasks < ( !Putdown ?b3 ) (Make-Clear ?b1 ) > )


(:method M4
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block ?b3 - block )
 :preconditions { ( on ?b2 ?b1 ) ( on ?b3 ?b2 )
    ( clear ?b3) ( hand-empty ) }
 :subtasks < ( Make-Clear ?b2 ) ( Make-Clear ?b1 ) > )


(:method M5
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block ?b3 - block ?b4 - block )
 :preconditions { ( on ?b2 ?b1 ) ( on ?b3 ?b2 )
    ( clear ?b3 )  ( holding ?b4 ) }
 :subtasks < ( Make-Clear ?b2 ) ( Make-Clear ?b1 ) > )


(:method M6
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block ?b3 - block ?b4 - block )
 :preconditions { ( on ?b2 ?b1 ) ( on ?b3 ?b2 )
    ( on ?b4 ?b3 ) ( clear ?b4 ) ( hand-empty ) }
 :subtasks < ( Make-Clear ?b2 ) ( Make-Clear ?b1 ) > )
```

Figure 4.8: Some methods learned by an HTN learner in the Blocks World domain.

Chapter 5: Towards Learning Generalized Methods by Capturing Recursive Patterns

In hierarchical task network (HTN) learning, a notable disadvantage is that the methods learned from training problems often do not generalize well to larger, more complex testing problems. Specifically, while HTN-MAKER and CURRICULAMA can produce methods that solve many problems within the scope of the training data, these methods often struggle when applied to problems of greater size or complexity, as they are typically tailored to specific scenarios encountered during training. To address this limitation, this chapter introduces a novel approach to learning generalized methods that capture recursive patterns, enabling the solution of problems at any scale.

The key innovation lies in the ability of these generalized methods to abstract away from specific details—such as the exact number of elements involved in a task—while preserving the essential recursive structure needed to solve the problem. For instance, rather than learning separate methods for clearing a block in a stack of three or four blocks, a generalized method can handle stacks of any size by recursively applying the same solution strategy.

The introduction of this generalization technique not only enhances the robustness and scalability of HTN methods but also reduces the total number of methods required, which can lead to more efficient planning. The experimental results presented in this chapter demonstrate

the effectiveness of this approach across various planning domains, showing improvements in the ability to tackle larger, more complex problems.

In the following sections, we detail the algorithmic foundations of this approach, provide illustrative examples, analyze the empirical results that highlight the benefits of learning generalized methods with recursive characteristics, and discuss some limitations of the approach and ideas for future research.

5.1 Understanding the Need for Generalized Methods in HTN Learning

HTN methods are generally applicable across different scenarios because they are ”lifted.” This means that the methods' parameters are not tied to specific objects (constants) but remain as variables. Any specific object that fits the specified type can replace these variables, provided it satisfies the relationships outlined in the preconditions [11].

For example, each of the methods in Figure 4.8 is lifted, and is applicable to situations where there are a certain amount of blocks above the block that this method tries to make clear, and the robot hand could be holding some block or empty. Specifically, for situations where the robot hand is empty, `M2` is applicable to situations where there is only one block above the objective, and `M4` is applicable to situations where there are two blocks above the objective, and `M6` is applicable to situations where there are three blocks above the objective. However, this set of methods is not applicable in situations where there are four blocks or more than four blocks above the objective.

In contrast, `Mg` shown in Figure 5.1 is applicable to situations where there are any number of blocks above the objective. This method operates in two primary steps: 1) it clears the

```
(:method Mg
 :head ( Make-Clear ?b1 - block )
 :vars ( ?b2 - block ?b3 - block )
 :preconditions { ( on ?b2 ?b1 ) ( hand-empty ) }
 :subtasks < ( Make-Clear ?b2 ) ( Make-Clear ?b1 ) > )
```

Figure 5.1: A generalized method that is recursive in nature.

block directly above the objective block no matter how many blocks are on the block, and 2) it subsequently clears the objective block itself. Distinguished from the methods typically learned by HTN learners (as are listed in Figure 4.8), this method does not specify the exact number of blocks above the target block. The first step of the method acts as the recursive component, enabling subsequent calls to the method itself during the task decomposition process. Essentially, it's a strategy for tackling complex problems by reducing them to simpler problems, recursively, until they can be easily managed.

This kind of method is particularly useful when there is a task that can be broken down into smaller versions of itself until the task is simple enough to solve directly. We refer to such methods as *generalized methods* (beyond lifted methods) as they can handle an unlimited number of situations by following a recursive solution. Since tasks that can be broken down into smaller versions of themselves are very common in automated planning domains, this chapter investigates how to learn such generalized methods automatically to boost the robustness of the learned method set.

Learning such generalized methods could potentially be achieved by extracting common structures from non-generalized methods. *E.g.*, method `Mg` is a generalized method that shares the same solution structure as method `M4` and `M6` (they have analogous subtasks) but only keeps

necessary preconditions. The following section lays the foundation for an algorithm that learns generalzied methods like `Mg`.

5.2 Automatically Learning Generalized Methods with Recursive Characteristics

To effectively learn generalized methods, first, we need to identify every pair of methods that accomplishes the same task and has *analogous subtasks*.

Definition 9 (Analogous Subtasks)**.** *Two methods, m_1 and m_2, have analogous subtasks if:*

- *They contain the same number of subtasks.*
- *The subtask names are identical across both methods.*
- *The variables within each corresponding subtask for each method, denoted as V_{m_1} and V_{m_2} (as defined next), are semantically equivalent.*

Two variable lists are equivalent if those variables shares analogue relationships in terms of variable names and preconditions that involve those variables.

Definition 10 (Semantic Equivalence of Variables)**.** *Given a method m, V_m is the list of variables ordered by their occurence in m^{sub}. $V_m[i]$ indicates the i-th variable. Given two methods m_1 and m_2, V_{m_1} and V_{m_2} are semantically equivalent if:*

- *for all pairs (i, j) in the range of the length of V_{m_1}:*

$$(V_{m_1}[i] = V_{m_1}[j]) \iff (V_{m_2}[i] = V_{m_2}[j])$$

$$(V_{m_1}[i] \neq V_{m_1}[j]) \iff (V_{m_2}[i] \neq V_{m_2}[j])$$

- *for each atom* $(p\ V_{m_1}[i]\ V_{m_1}[j]\ ...)$ *in* m_1^ϕ *that has and only has variables in* V_{m_1} *, atom* $(p\ V_{m_2}[i]\ V_{m_2}[j]\ ...)$ ***must exist in*** m_2^ϕ.

After identifying two methods for a certain task that have the same solution structure, we can make a generalized method by extracting those two methods' *common preconditions*.

Definition 11 (Substitution of Variable Names in Preconditions)**.** *Let* V_m *be the variables in method* m*'s subtasks. Give two methods* m_1 *and* m_2*,* $|V_{m_1}|$ = $|V_{m_2}|$*. The substitution of the variable names in* m_2^ϕ *with the variable names in* m_1 *is denoted as* $sub(m_2^\phi, V_{m_1})$.

Definition 12 (Common Preconditions)**.** *Given two methods,* m_1 *and* m_2*, the common preconditions, denoted as "*$m_1^\phi \cap m_2^\phi$*",*

$$m_1^\phi \cap m_2^\phi = \{a \in m_1^\phi \mid a \in sub(m_2^\phi, V_{m_1})\}$$

The following algorithm, METHODGENERALIZER, systematically refines a library of HTN methods by identifying and merging analogous methods based on their task structures and precondition similarities.

Algorithm 5 Curriculum Learning with Landmarks

1: **procedure** METHODGENERALIZER(M) ▷ M is a library of HTN methods
2: Let $M' = \emptyset$
3: **while** $\exists m_1, m_2 : m_1^{head} = m_2^{head}$ and m_1^{sub} is analogous to m_2^{sub} **do**
4: $prec = m_1^{prec} \cap m_2^{prec}$
5: $m_{new} = (m_1^{head}, \text{TRIMVARIABLES}(prec), prec, m_1^{sub})$
6: Remove m_1 and m_2 from M
7: Add m_{new} to M'
8: **return** M'

This algorithm examines each pair of methods in the library to find candidates for generalization by focusing on similarities in their task structures and precondition constraints. The iterative process identifies methods with the same head and analogous subtasks (line 3), then extracts their common preconditions (line 4) to create a new, more generalized method (line 5). The function TRIMVARIABLES constructs the new method's parameters by only keeping parameters that are still present in the new methods' preconditions.

Example Methods `M4` and `M6` have analogous subtasks because both methods' two subtasks first make one block clear, then make the objective block right below the first block clear. Since the variable names in their subtasks' parameters happen to align, there is no need for variable name substitution. Their common precondition is the atom `(on ?b2 ?b1)`. Thus, we obtain the generalized method `Mg` as the result of merging `M4` and `M6`.

By employing this algorithm, the learning process strategically combines methods to produce a streamlined library that maintains functional efficacy while broadening the range of problems each method can address. This approach also reduces the total amount of methods in the library, thereby potentially reducing planning time.

5.3 Evaluating the Generalization Algorithm

To assess the effectiveness of the proposed generalization algorithm, we conducted a series of experiments across five different planning domains. These domains included variations that simulate natural numbers to depict features such as the number of blocks in a stack or distances between rooms. This experimental setup was designed to test the hypothesis that generalized methods can solve a wider range of problems, capturing the recursive solution patterns that are essential for scaling.

The methodology for evaluating the generalization algorithm involved a structured experimental procedure. For each domain, 200 problems were randomly generated, split into sets of 50 training problems and 150 testing problems. The testing problems were further categorized based on their size: 50 problems of the original training size (x), 50 problems twice as large ($2x$), and 50 problems three times the original size ($3x$). This partition helped in assessing the capabilities of the learned methods.

During the training phase, HTN methods were incrementally developed using the training set. The testing phase then evaluated the effectiveness and robustness of these methods by applying them to the test sets with incrementally larger problem sizes. To ensure the reliability of the results, each experiment was repeated five times for each domain. Three configurations of the learning algorithm were evaluated:

- HTN-Maker: Served as the baseline method generation algorithm.
- CurricuLAMA: An advanced method learning approach.
- CurricuLAMA with Generalized Methods: Integrated the generalization technique with

the methods learned by CurricuLAMA to develop recursive methods.

The effectiveness of these configurations was measured using several metrics:

- Planning Success Rate: The percentage of testing problems that were successfully solved.
- Planning Time: The time taken to solve each testing problem.
- Plan Length: The Length of the Plan.
- Total Number of Methods Learned: Recorded as a function of the number of training problems seen, this metric compared the learning efficiency across different approaches.

5.3.1 Analysis of Results

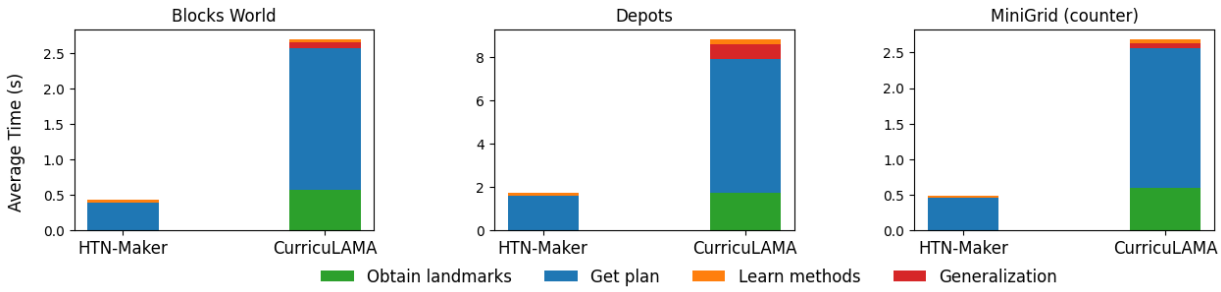


Figure 5.2: *Running time needed to learn methods.* The bars represent the average time that each learning algorithm spent on different parts of the learning process. Green represents the time to obtain landmarks (Alg 4, Line 2 and 3), blue indicates the time to obtain the plan (Alg 4, Line 8 to 16), and orange shows the time to learn methods. When applying the generalization algorithm to CURRICULAMA, some extra running time represent by red color is used.

The results of the experimental evaluation confirm the efficacy of the proposed generalization algorithm in efficiently learning recursive methods. The running time required for the generalization process, as shown in Figure 5.2, is relatively low, typically requiring only between

0.1s to 1s. This brief duration indicates that the additional step of generalization introduces minimal overhead to the overall method-learning process.

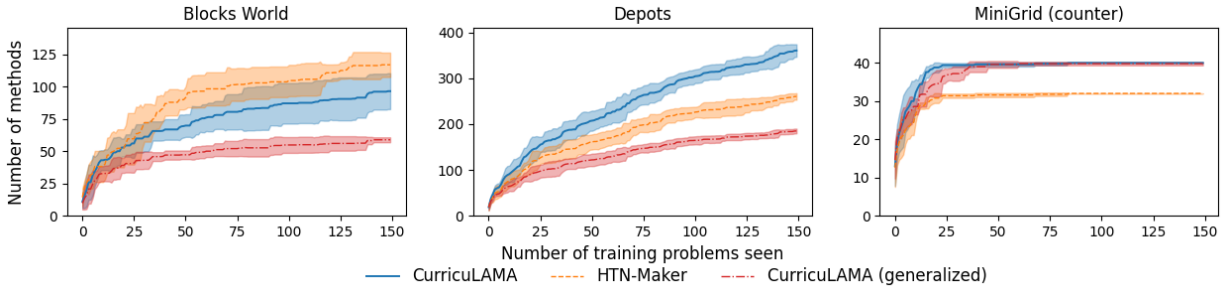


Figure 5.3: *Number of methods learned.* The y-axis is the total number of methods learned, and the x-axis is the number of training problems from which they were learned. All algorithms show increases in the number of methods as they are exposed to more training problems. The generalization algorithm significantly reduces the cumulative number of methods during learning for the Blcks world domain and Depots domain0. The shaded areas indicate the variance in the number of methods learned across five trials.

A significant finding from the experiments is the substantial reduction in the number of methods in the library, evidenced in Figure 5.3. This reduction not only simplifies the method base, potentially decreasing planning time, but also potentially eases the cognitive load when interpreting the methods.

Furthermore, the generalized methods demonstrated a higher planning success rate across all problem sizes (Figure 5.4). In every case, the generalized methods dominates the other approaches. This improvement suggests that these methods are not just applicable to a broader range of problems but are also effective in capturing the necessary recursive patterns for diverse scenarios.

The generalization algorithm also resulted in longer planning times and plan lengths, as shown in Figures 5.5 and 5.6. This outcome is a reflection of the robustness and versatility of the generalized methods, which can solve a wider range of more complex problems. In other

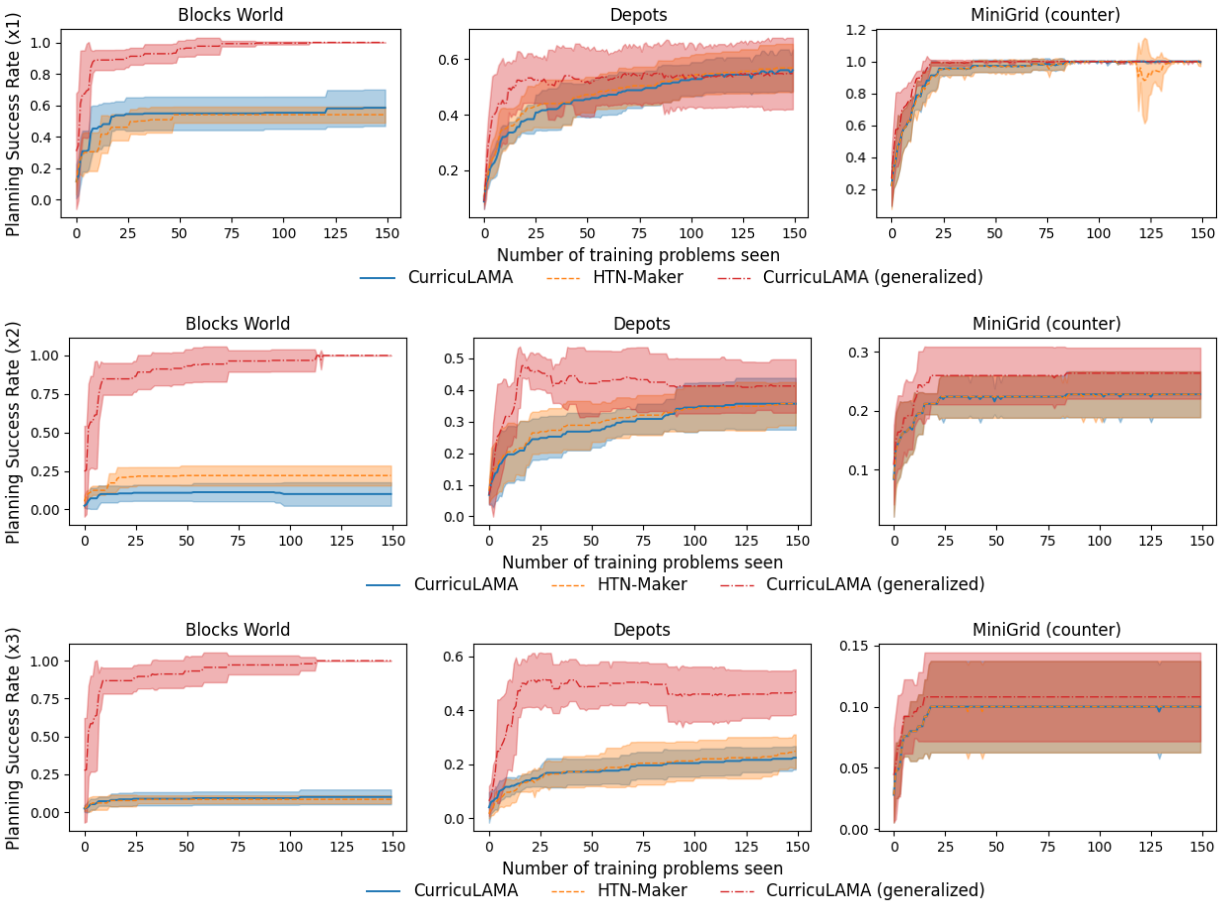


Figure 5.4: *Convergence analysis.* The figures illustrate convergence of planning success rates for three distinct problem sizes, denoted as $\times 1$, $\times 2$, and $\times 3$, positioned on the left side of each plot. The y-axis shows the fraction of problems that the planner could successfully solve using the methods that each learning algorithm learned, and the x-axis shows the number of training problems from which the methods were learned. The shaded regions show the variance in problems solved across five separate trials.

words, these problems inherently require longer and more involved solutions, contributing to the increased average plan length and planning time.

5.4 Summary

This chapter has introduced an innovative approach to learning generalized HTN methods that exhibit recursive characteristics, with the aim of generalizing problem-solving capabilities

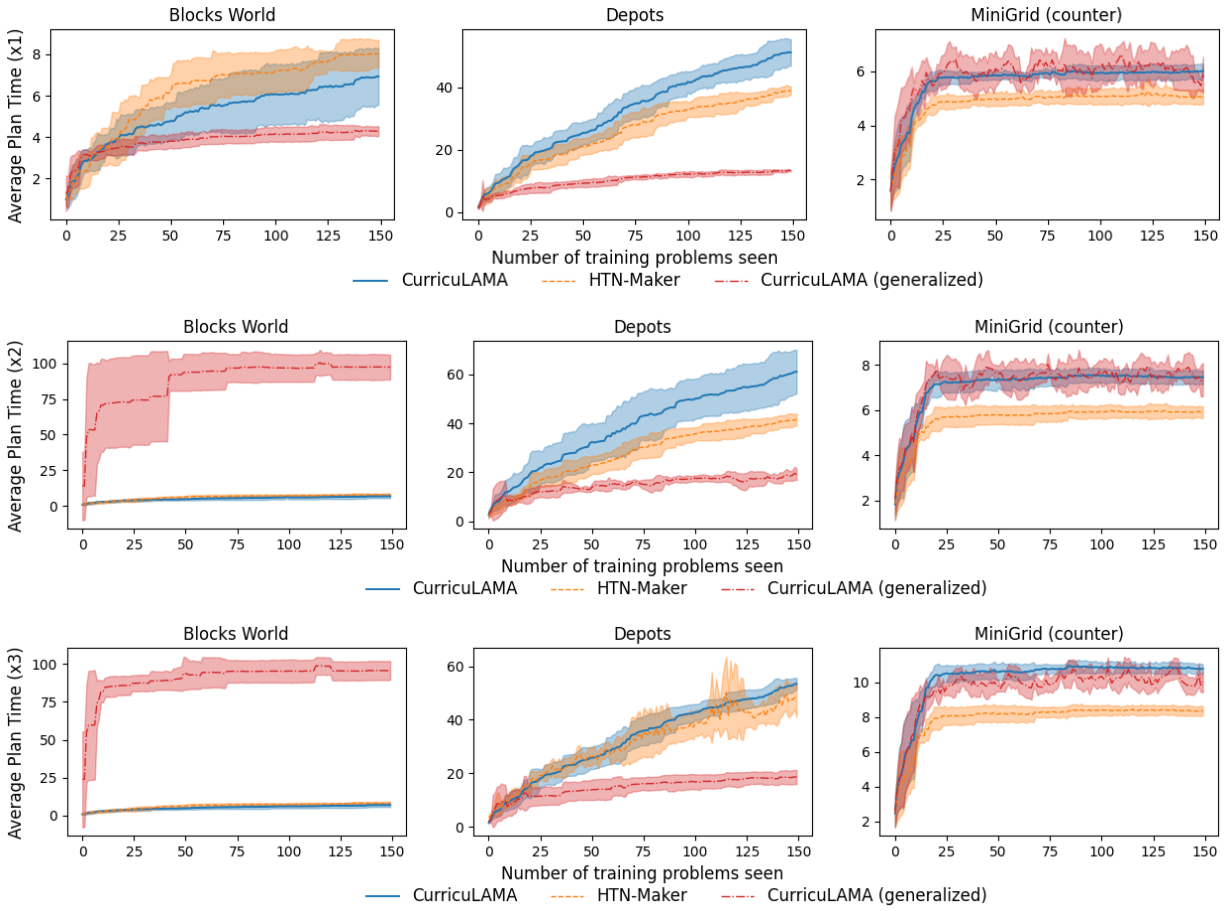


Figure 5.5: *Time to solve planning problems using the learned methods.* The figures illustrate the average planning time for three distinct problem sizes, denoted as $\times 1$, $\times 2$, and $\times 3$, positioned on the left side of each plot. The x-axis gives the number of training problems from which each learning algorithm learned its methods, and the y-axis gives the average planning time over the 50 test problems. The shaded regions denote the variance in planning time across five separate trials.

across different scales and complexities, contributing to more scalable AI planning systems.

The development of a generalization algorithm marks a significant advancement, effectively merging analogous methods to produce generalized, recursive methods. This not only reduces the number of methods needed but also enhances their applicability across varying problem sizes. The algorithm proves to be both efficient in execution and impactful in results, leading to a reduced method count and improved success rates in problem-solving.

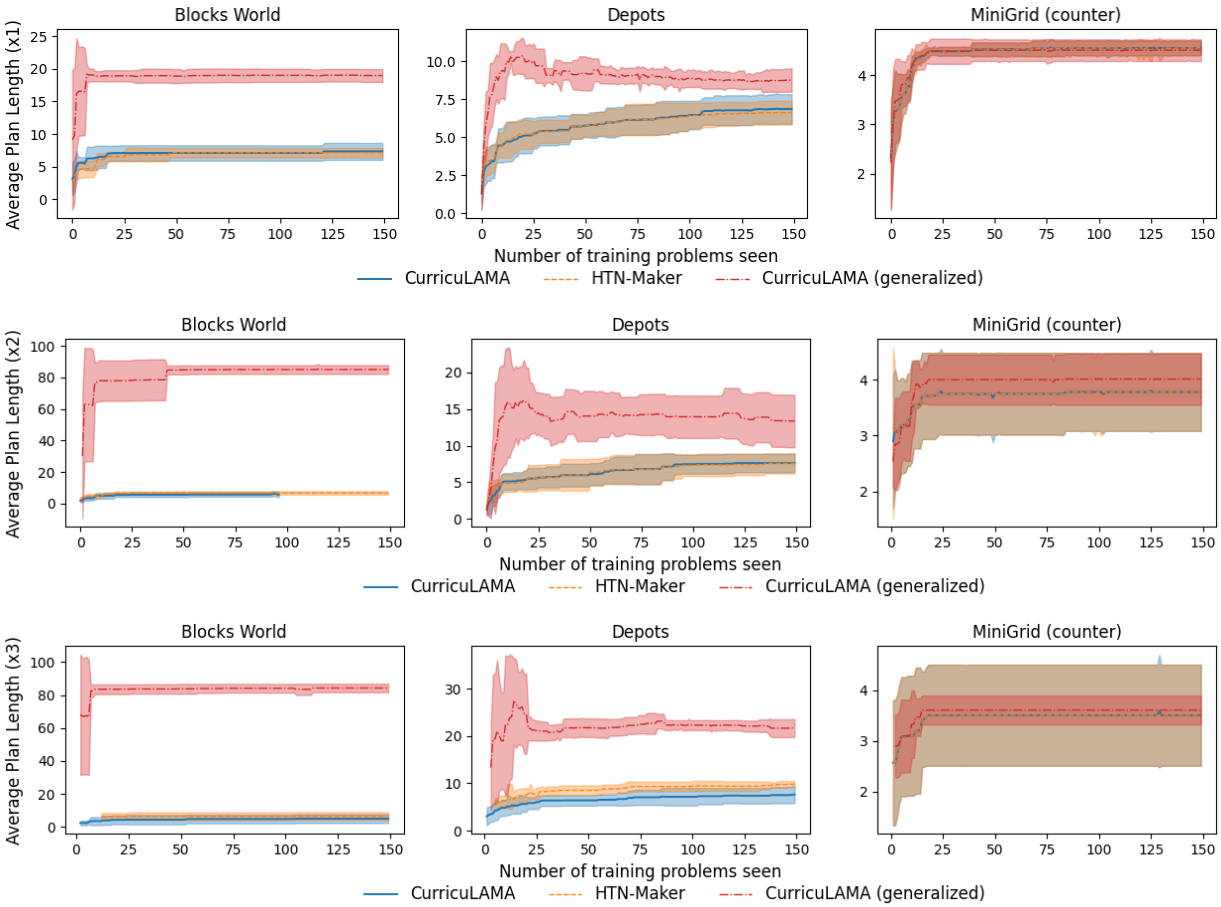


Figure 5.6: *Average plan lengths.* The figures illustrate the average plan lengths for three distinct problem sizes, denoted as $\times 1$, $\times 2$, and $\times 3$, positioned on the left side of each plot. The y-axis shows the average length of the plans that the planner produced using the learned methods, and the x-axis shows the number of training problems from which the methods were learned, ranging from zero to 150 training problems. The shaded regions show the variance in plan length across five separate trials.

While powerful, this approach has limitations. It relies on the presence of a recursive element in the problem structure, which may not always exist. In other words, this approach is less suitable for domains where each problem presents unique, non-repetitive challenges that require bespoke solutions. On the other hand, the abstraction process may also trim more preconditions than necessary, thereby causing the methods to become problematic. For instance, overly generalized methods could fail to terminate, resulting in infinite loops, or leading to incomplete or incorrect solutions.

Future research could focus on exploring the application of this methodology across a broader range of domains especially the ones characterized by:

1. Problems that involve repetitive actions or structures, such as stacking or nesting operations, can benefit immensely from recursive solutions.

2. Domains where problems can scale in complexity without altering the fundamental nature of the tasks (*e.g.*, sorting larger lists, navigating increasingly complex mazes).

3. Problems that naturally decompose into smaller, similar problems, such as parsing nested grammatical structures or decomposing arithmetic expressions.

This could further validate METHODGENERALIZER's adaptability and robustness.

Chapter 6: Related Work

In HGN planning, when domain knowledge is incomplete, the Goal-Decomposition Planner (GDP) [28] is unable to plan for some goals that lack decomposition methods. The Goal Decomposition with Landmarks (GoDeL) algorithm [29] overcomes this issue by leveraging landmarks to break down otherwise unsolvable goals. Our work is inspired by this technique. However, we not only use landmarks to break down goals into subgoals but also learn task decomposition methods (annotated with goals) from a curriculum generated from those subgoals. This allows us to automatically complete or learn from scratch the set of methods in the library.

Several researchers have explored methods for learning HTN methods. Lotinac and Jonsson [38] employed invariant analysis to construct HTNs from the PDDL descriptions of planning domains, focusing on a single representative instance. While their approach emphasizes domain-specific invariants, it does not involve learning from sequences of actions as our work does.

Learn-HTN [39] learns HTN method preconditions and action models from task decomposition trees, using a weighted MAX-SAT solver to address constraints. Similarly, HTNLearn [40] learns HTN methods and action models from partially observed plan traces annotated with potentially incomplete decomposition trees that capture task-subtask relationships. These approaches handle partial observability, whereas our approach assumes full observability.

DInCaD [41] utilizes domain-independent task decomposition techniques in scenarios where

cases are the primary source of domain knowledge. This method applies to domains where hierarchical cases are available but lacks a formal planning domain theory or case adaptation knowledge. Unlike our approach, this work is not tailored to HTNs and does not use curricula for learning.

HTN learning systems often require a substantial number of plan traces to converge. To address slow convergence, Ilghami et al. [42] proposed generating solution plans before the planning domain is fully learned by acquiring approximate method preconditions. This technique could complement our approach by reducing the need for extensive training.

Hérail and Bit-Monnot [43] iteratively learns HTNs using bottom-up pattern recognition and compression techniques on recurring subtask sequences in traces, replacing them with task symbols. Unlike their approach, our research focuses on curriculum learning guided by landmarks and emphasizes learning generalized recursive methods.

Segura-Muros et al. [44] applies process and data mining to learn HTNs by converting execution traces into event logs and extracting task preconditions and effects using a fuzzy rule-based learning algorithm. In contrast, our work employs curriculum learning through landmarks and concentrates on learning generalized and recursive HTN methods, rather than relying on data mining techniques.

The algorithm by Li et al. [45] uses techniques similar to probabilistic context-free grammar learning and learns probabilistic HTNs (pHTN). Although this work is able to recognize recursive structures, however, it does not generalized the pHTN decomposition strategy to make it applicable to a unlimited number of situations.

Word2HTN [46] combines semantic text analysis techniques and subgoal learning to create HTNs. Plan traces are viewed as sentences where a plan trace consists of actions with their

grounded preconditions and effects. Each word in the sentence is an atom or an action in the plan trace. This work uses Word2Vec to convert each word into a vector representation and applies agglomerative hierarchical partitioning on the learned vectors to learn methods with binary-subtask decompositions. As an extension to their approach, Fine-Morris et al. [47] approximates landmarks using solution traces and learns methods with symbolic and numeric preconditions that initially decompose problems using two or more landmarks, then finish the decomposition using (primitive task, complex task) right-recursion. This work requires a set of goals and plan traces as input and extracts landmarks by analyzing the probabilistic occurrence of actions in the plan traces. In contrast, our work does not require a predefined set of goals or plan traces; besides, we explicitly discover landmarks using dedicated landmark algorithms.

CC-HTN [48] and Circuit-HTN [49] translate execution traces into multi-trace graphical representations where primitive tasks comprise vertices and edges indicate sequential tasks. They apply bottom-up consolidation techniques to group simple tasks into progressively larger ones. However, these approaches learn HTNs in the form of task graphs instead of decomposition methods, and they do not use any curricula to progressively guide the learning.

HPNL [50] is a system that learns new methods for Hierarchical Problem Networks (HPN) [51] by analyzing sample hierarchical plans, using violated constraints to identify state conditions, and ordering conflicts to determine goal conditions. However, this work is for learning HPNs instead of HTNs, and does not incorporate curriculum learning or recursive pattern recognition.

The hierarchical plan recognition algorithm by Geib [52] uses Combinatory Categorial Grammars (CCGs) as part of the ELEXIR framework. It requires a hand-authored CCG representing structure of plans done by agents. Lex_{Learn} [53] learns CCGs by enumerating CCG

categories for a set of plan traces from templates. Lex_{Greedy} [54] employs a greedy approach to improve the scalability of CCG learning. Lex^{T}_{Greedy} [54] learns CCGs in domains with type trees as an extension to Lex_{Greedy}. While these approaches advance the learning of CCGs, they do not use curricula and do not learn recursive decomposition strategies.

Teleoreactive Logic Programs (TLPs) are a framework for encoding knowledge using ideas from logic programming, reactive control, and HTNs. The work that learns TLPs includes ways to learn recursive TLPs from problem solution traces [55], a learning method that acquires recursive forms of TLP structures from traces of successful problem solving [56], and an incremental learning algorithm for TLPs based on explanation-based learning [57]. The learning of TLPs does not involve curriculum. Even though recursive TLP structures that could be applied to unlimited number of situations are learned, they took advantage of high level concepts that recognizes classes of situations in each domain. While our work uses curriculum learning and does not rely on those high-level concepts.

Chapter 5 highlights the importance of identifying analogous subtasks, a key step in recognizing recursive patterns. The research area of computational analogy [58] [59] offers valuable insights into this process, particularly emphasizing the role of structural similarity in analogical reasoning. Structural consistency is crucial for meaningful analogical inference and is central to leveraging recursive patterns in problem-solving tasks. However, while this body of work provides a strong foundation, it does not specifically address the HTN domain. In this context, our research introduces a technique to compare similarities and identify common recursive structures within HTNs.

Chapter 7: Conclusion

In this dissertation, we presented new algorithms for automatically learning structural domain knowledge for automated planning systems. Specifically, the work included the application of curriculum learning to facilitate this process, the use of landmarks to generate effective curricula, and the development of a generalization algorithm to learn recursive HTN methods.

This dissertation provided the following contributions:

- **HTN Learning with Curricula:** We developed a new algorithm called CURRICULEARN that could efficiently learn HTN methods from curricula by focusing on specific subtraces of solution plans. Our empirical evaluation results demonstrated that CURRICULEARN, compared to HTN-MAKER, significantly reduced computational effort, decreased the number of learned methods, and enhanced the quality of learned methods. We also theoretically proved that given the curricula designed by a domain expert, CURRICULEARN can use the learned methods to solve the hierarchical planning problem that is equivalent to the classical planning problem used to learn those methods.

- **Automatic Curriculum Generation:** We developed an innovative algorithm called CURRICULAMA that uses landmarks to automatically generate curricula and guides the learning of HTN methods from the curricula. CURRICULAMA completely eliminated the manual effort required in traditional methods, while our experimental results showed that it

maintains comparable learning performance compared to HTN-MAKER. Our theoretical analysis showed that given the automatically generated curricula, CURRICULEARN can still use the learned methods to solve the hierarchical planning problem that is equivalent to the classical planning problem used to learn those methods.

- **Generalized (Recursive) Method Learning:** We developed an algorithm called METHODGENERALIZER that learns generalized methods that can be recursively applied to solve more complex problems. This technique analyzes patterns in method structures to generate effective generalized methods. Our empirical analysis showed that the generalized methods significantly improved planning success rates in testing problems that are more complex than the training problems.

However, while the accomplishments of this work represent significant advancements, there are also some limitations:

- **Number of Methods Learned:** While CURRICULEARN generally learns fewer methods than HTN-MAKER, there are instances where it may still produce more methods than necessary, depending on the domain and problem complexity. This can lead to inefficiencies, which should be addressed in future work.

- **Domain Restrictions:** The scope of this work is currently limited to classical planning domains. The developed methods and algorithms may not perform optimally in more complex or dynamic environments that require advanced reasoning, such as those involving uncertainty or real-time decision-making.

- **Limitations of the Generalizer Algorithm:** Although METHODGENERALIZER is effec-

tive, it has limitations in its ability to generalize across different types of problems, particularly in domains where recursive patterns are less prevalent or where the problem structure exceeds the patterns recognized by the algorithm.

7.1 Implications and Future Work

The contributions of this dissertation represent meaningful progress in the field of automated planning. By introducing innovative ways to generate and utilize curricula and learn generalized HTN methods, this work fully automates HTN learning from classical planning problems and enhances the capabilities of the learned methods.

Given these accomplishments, future research should explore several promising directions:

- **Expanding Beyond Classical Planning:** Future work could investigate extending the developed methods to handle more complex domains, such as those involving temporal planning, uncertainty, or multi-agent systems. This could involve integrating techniques from probabilistic planning or reinforcement learning.

- **Improving the Generalizer:** The recursive pattern recognition in METHODGENERALIZER could be enhanced to capture more complex patterns or to generalize across a wider variety of domains.

- **Addressing the Utility Problem:** Further research could focus on optimizing the selection of methods or developing techniques to prune unnecessary methods during the learning process, ensuring that only the most useful methods are retained, thus mitigating the utility problem.

By tackling these areas, this work opens up the possibility of ”planning to learn,” where the learner acquires structural knowledge from planning problems specifically designed to address areas it has not yet mastered. This approach could significantly advance the capabilities of automated planning systems and broaden their applicability.

Appendix A: An Example HTN Domain and Problem

Listing A.1: The Blocks World HTN Domain

```
( define ( domain Blocks4 )
  ( :requirements :strips :typing :equality :htn )
  ( :types block )

  ( :predicates
    ( on-table ?b - block )
    ( on ?b1 - block ?b2 - block )
    ( clear ?b - block )
    ( hand-empty )
    ( holding ?b - block )
  )

  ( :action !Pickup
    :parameters
    (
      ?b - block
    )
    :precondition
    ( and
      ( on-table ?b )
      ( clear ?b )
      ( hand-empty )
    )
    :effect
    ( and
      ( not ( on-table ?b ) )
      ( not ( clear ?b ) )
      ( not ( hand-empty ) )
      ( holding ?b )
    )
  )
```

Listing A.1 (Cont.): The Blocks World HTN Domain

```
  ( :action !Putdown
    :parameters
    (
      ?b - block
    )
    :precondition
    ( and
      ( holding ?b )
    )
    :effect
    ( and
      ( not ( holding ?b ) )
      ( hand-empty )
      ( on-table ?b )
      ( clear ?b )
    )
  )

  ( :action !Unstack
    :parameters
    (
      ?b1 - block
      ?b2 - block
    )
    :precondition
    ( and
      ( on ?b1 ?b2 )
      ( clear ?b1 )
      ( hand-empty )
    )
    :effect
    ( and
      ( not ( on ?b1 ?b2 ) )
      ( not ( clear ?b1 ) )
      ( not ( hand-empty ) )
      ( clear ?b2 )
      ( holding ?b1 )
    )
  )

  ( :action !Stack
```

Listing A.1 (Cont.): The Blocks World HTN Domain

```
    :parameters
    (
      ?b1 - block
      ?b2 - block
    )
    :precondition
    ( and
      ( holding ?b1 )
      ( clear ?b2 )
    )
    :effect
    ( and
      ( not ( holding ?b1 ) )
      ( not ( clear ?b2 ) )
      ( hand-empty )
      ( on ?b1 ?b2 )
      ( clear ?b1 )
    )
  )

  ( :method
    :head Make-On-Table ( ?goal - block )
    :precondition
    ( and
      ( on-table ?goal )
    )
    :subtasks
    (
    )
  )

  ( :method
    :head Make-On ( ?top - block ?bottom - block )
    :precondition
    ( and
      ( on ?top ?bottom )
    )
    :subtasks
    (
    )
  )
```

Listing A.1 (Cont.): The Blocks World HTN Domain

```
  ( :method
    :head Make-Clear ( ?target - block )
    :precondition
    ( and
      ( clear ?target )
    )
    :subtasks
    (
    )
  )
)
```

Listing A.2: A Blocks World HTN Problem

```
( define ( htn-problem p1 )
  ( :domain blocks4 )
  ( :requirements :strips :htn :typing :equality )
  ( :objects
    b1 - block
    b2 - block
    b3 - block
  )
  ( :init
    ( hand-empty )
    ( on-table A )
    ( on B A )
    ( on C B )
    ( clear b3 )
  )
  ( :tasks
    ( Make-Clear b1 )
  )
)
```

Appendix B: CURRICULEARN Theory

In this chapter, we prove that the methods we learn using CURRICULEARN are the ones that we want to learn. Given that the curricula provided to CURRICULEARN contain domain-specific patterns, our proofs must also be tailored to each domain. For brevity, we will present the proof for only one domain: the Blocks World domain.

The Blocks World domain includes a number of blocks sitting on a table (possibly on top of each other), and a robotic hand that can grasp one block at a time. The objective is to learn methods to move a stack of *n* blocks using the robotic hand, keeping the top-to-bottom order of the blocks the same as in the original stack. We show that we can learn desired methods in a class of problems in the Blocks World domain.s

Definition 13. *We define a class of problems in the Blocks World domain* $P = (\Sigma, s_0, g)$*, where* s_0 = $\langle$*on(*b_0*,table), on(*b_1*,*b_0*), on(*b_2*,*b_1*),..., on(*b_n*,*b_{n-1}*), clear(*b_n*)*$\rangle$*, and* g_n = $\langle$*on(*b_0*,table), clear(*b_0*), on(*b_1*,table), on(*b_2*,*b_1*),..., on(*b_n*,*b_{n-1}*), clear(*b_n*)*$\rangle$.

Definition 14. π = $\langle$*unstack(*b_n*,*b_{n-1}*), putdown(*b_n*), unstack(*b_{n-1}*,*b_{n-2}*), stack(*b_{n-1}*,*b_n*), ..., unstack(*b_1*,*b_0*), stack(*b_1*,*b_2*), unstack(*b_1*,*b_2*), putdown(*b_1*), unstack(*b_2*,*b_3*), stack(*b_2*,*b_1*), ..., pickup(*b_n*), stack(*b_n*,*b_{n-1}*)*$\rangle$*, is a solution to* P*, where the first 2*n actions move a stack of n blocks from a block x onto the table while inverting the order, and the second 2*n actions move the stack of n blocks from the table onto the table while inverting the order again.*

Definition 15. `Make-nPile` *is an annotated task that has the effect of making a pile of* n *blocks on the table. Examples are as following:*

```
(:task Make-1Pile
 :parameters
   (?a)
 :precondition
   (and)
 :goals
   (and
     (on-table ?a)
     (clear ?a)))
```

(a) `Make-1Pile` Task

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

(b) `Make-2Pile` Task

Figure B.1: Example annotated tasks in the Blocks World.

Definition 16. *To represent a learning step, we use the task name following by the method type (a-f) and subtrace's beginning and ending indices. e.g.,* `Make-2Pile`*[b][3,4] is a learning step that learns a method of b type for task* `Make-2Pile` *from the 3rd to the 4th action in the plan trace.*

Definition 17. *The curriculum to learn methods from* P *and* π *as* C = ⟨*(1,2,*`Make-1Pile`*), (3,4,*`Make-2Pile`*), (1,4,*`Make-2Pile`*), (5,6,*`Make-3Pile`*), (1,6,*`Make-3Pile`*), ..., (1, 2n,* `Make-nPile`*), (2n+1,2n+2,*`Make-1Pile`*), (2n+3,2n+4,*`Make-2Pile`*), (2n+1,2n+4,*`Make-2Pile`*), (2n+5,2n+6,*`Make-3Pile`*), (2n+1,2n+6,*`Make-3Pile`*), ..., (2n+1, 4n,* `Make-nPile`*), (1,4n,* `Make-nPile`*)*⟩*. There are 4n-1 steps in* C*.*

Example Let τ be the task of moving a stack of two blocks (A and B) from block C onto the table, while maintaining their order (i.e. A on B). Let π be the following plan for that task:

Action 1: unstack(A,B)	Action 5: unstack(B,A)
Action 2: putdown(A)	Action 6: putdown(B)
Action 3: unstack(B,C)	Action 7: pickup(A)
Action 4: stack(B,A)	Action 8: stack(A,B)

Figure 3.1 shows what the plan does.

To teach how to accomplish τ, we can use the curriculum in Table 3.1. It consists of seven subplans of π, starting with simpler ones and combining them into progressively harder ones. For each subplan, the curriculum includes the annotated task (see Figure 3.4 for examples) that the subplan accomplishes. The preconditions of the methods learned from this curriculum include cases where part of the stack has already been moved, but not cases where a block is held in the robot hand.

Definition 18. *We define 6 kind of methods as following, where n ($n < 2$ and $n \in \mathbb{R}$) is a variable that indicates the number of blocks in the pile:*

1. *method type a:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, x), on(b_{n-1}, b_n), ..., on(b1, b2), clear(b1), empty-hand

   sub: unstack(b_n, x), putdown(b_n, table)

2. *method type b:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, x), clear(b_n), on(b1, table), on(b2, b1), ..., on(b_{n-1}, b_{n-2}), clear(b_{n-1}), empty-hand

   sub: unstack(b_n, x), stack(b_n, b_{n-1})

3. *method type c:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, table), clear(b_n), on(b1, table), on(b2, b1), …, on(b_{n-1}, b_{n-2}), clear(b_{n-1}), empty-hand

   sub: pickup(b_n, x), stack(b_n, b_{n-1})

4. *method type d:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, x), on(b_{n-1}, b_n), …, on(b1, b2), clear(b1), empty-hand

   sub: `Make-n-1Pile`*(b1, b2, …, b_{n-1}),* `Make-nPile`*(b1, b2, …, b_n)*

5. *method type e:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, table), on(b_{n-1}, b_n), …, on(b1, b2), clear(b1), empty-hand

   sub: `Make-n-1Pile`*(b1, b2, … b_{n-1}),* `Make-nPile`*(b1, b2, … b_n)*

6. *method type f:*

   head: `Make-nPile`$(b_1, b_2, \ldots, b_n)$*:*

   prec: on(b_n, b_{n-1}), clear(b_n), on(b_{n-1}, b_{n-2}), …, on(b2, b1), on(b_1, x), empty-hand

   sub: `Make-nPile`*(b_n, b_{n-1}, …, b1),* `Make-nPile`*(b1, b2, … b_n)*

Definition 19. *To represent a learned method with type, first we determine the value of n (the expression betwee* `Make` *and* `-Pile`*), then we annotate the grounded method type with a letter. e.g.,* `Make-1Pile`*[a] is a method of type a that resolves a* `Make-1Pile` *task. Sometimes, we further annotate the grounded method with begging and ending indices. e.g.,* `Make-1Pile`*[a][1,2]*

is a method of type a that resolves a `Make-1Pile` *task learned from subtrace that begins at index 1 and ends at index 2.*

Definition 20. *The desired set of method instances to learn for C on P and π is M = {1:* `Make-1Pile`*[a], 2:* `Make-2Pile`*[b], 3:* `Make-2Pile`*[d], 4:* `Make-3Pile`*[b], 5:* `Make-3Pile`*[d], . . . , 2n-2:* `Make-nPile`*[b], 2n-1:* `Make-nPile`*[d], 2n:* `Make-1Pile`*[a], 2n+1:* `Make-2Pile`*[b], 2n+2:* `Make-2Pile`*[d], 2n+3:* `Make-3Pile`*[b], 2n+4:* `Make-3Pile`*[d], . . . , 4n-3:* `Make-nPile`*[c], 4n-2:* `Make-nPile`*[e], 4n-1:* `Make-nPile`*[f]*}.

Proposition 3. *In the Blocks World domain, for any number of $n(n \geq 2)$, given P, π, and C,* CURRICULEARN *learns M.*

Proof.

Base case: when n=2, S_{02} = "on(A,B), on(B,C), on(C, table), clear(A)", Plan trace P2 = ⟨unstack(A,B), putdown(A), unstack(B,X), stack(B,A), unstack(B,A), putdown(B), pickup(A), stack(A,B)⟩, Curriculum C2 = ⟨(1,2,`Make-1Pile`), (3,4,`Make-2Pile`), (1,4,`Make-2Pile`), (5,6,`Make-1Pile`), (7,8,`Make-2Pile`), (5,8,`Make-2Pile`), (1,8,`Make-2Pile`)⟩. The following methods are learned using the curriculum:

- `Make-1Pile`[a] is learned from curriculum step 1 because action putdown(A) regresses goal "on(A, table), clear(A)" to "holding(A)", and action unstack(A,B) regresses "holding(A)" to "clear(A), on(A,X), empty-hand", which is the precondition of `Make-1Pile`[b].
- `Make-2Pile`[b] is learned from curriculum step 2 because action stack(B,A) regresses goal "on(B,A), on(A, table), clear(B)" to "holding(B), clear(A)", and action unstack(B,C) regresses "holding(B), clear(A)" to "clear(A), on(B,X), clear(B), empty-hand", which is the precondition of `Make-2Pile`[b].

- `Make-2Pile`[d] is learned from curriculum step 3 because `Make-2Pile`[b][3,4] regresses goal "on(A, table), on(B, A), clear(B)" to "on(B, X), clear(B), clear(A), hand-empty, on(A, table)", and `Make-1Pile`[a] regresses that to "on(B, X), on(A,B), clear(A)", which is the precondition of `Make-2Pile`[d].

- `Make-1Pile`[a] is learned from curriculum step 4 the same way as `Make-1Pile`[a] is learned from curriculum step 1.

- `Make-2Pile`[c] from curriculum step 5 is learned because action stack(A,B) regresses goal "on(A,B), on(B, table), clear(A)" to "holding(A), clear(B), on(B, table)", and action pickup(A) regresses "holding(A), clear(B), on(B, table)" to "clear(B), on(B, table), clear(A), on(A, table) empty-hand", which is the precondition of `Make-2Pile`[b].

- `Make-2Pile`[e] is learned from curriculum step 6 because `Make-2Pile`[c][7,8] regresses goal "on(B, table), on(A, B), clear(A)" to "clear(A), clear(B), hand-empty, on(B, table)", and `Make-1Pile`[a][5,6] regresses that to "on(A, table), on(B, A), clear(B), hand-empty", which is the precondition of `Make-2Pile`[e].

- `Make-2Pile`[f] is learned from curriculum step 7 because `Make-2Pile`[e] regresses goal "on(B, table), on(A, B), clear(A)" to "on(A, table), on(B, A), clear(B), hand-empty", and `Make-2Pile`[d] regresses that to "on(B, X), on(A,B), clear(A)".

Therefore, Methods learned M2 = ⟨1: `Make-1Pile`[a], 2: `Make-2Pile`[b], 3: `Make-2Pile`[d], 4: `Make-1Pile`[a], 5: `Make-2Pile`[c], 6: `Make-2Pile`[e], 7: `Make-2Pile`[f]⟩, in hierarchical form: (2f (2d (1a)[1,2] (2b)[3,4])[1,4] (2e (1a)[5,6] (2c)[7,8])[5,8])[1,8].

Induction step:

Induction assumption: when n = i,

When we are moving a stack of i blocks, according to Definition 1: the length of plan trace P_i is 4i (the first 2i actions move a stack of i blocks from a block X onto the table while inverting the order, and the second 2i actions move the stack of i blocks from the table onto the table while inverting the order again), and there are (2i-1)2+1 curriculum steps. We assume our hypothesis is true for n = i, so we learn the following method instances: M_i\{1: `Make-1Pile`[a][1,2], 2: `Make-2Pile`[b][3,4], 3: `Make-2Pile`[d][1,4], 4: `Make-3Pile`[b][5,6], 5: `Make-3Pile`[d][1,6], . . . , 2i-2: `Make-iPile`[b][2i-1,2i], 2i-1: `Make-iPile`[d][1,2i], 2i: `Make-1Pile`[a][2i+1,2i+2], 2i+1: `Make-2Pile`[b][2i+3,2i+4], 2i+2: `Make-2Pile`[d][2i+1,2i+4], 2i+3: `Make-3Pile`[b][2i+5,2i+6], 2i+4: `Make-3Pile`[d][2i+1,2i+6], . . . , 4i-3: `Make-iPile`[c][4i-1,4i], 4i-2: `Make-iPile`[e][2i+1,4i], 4i-1: `Make-iPile`[f][1,4i]\}, in hierarchical form: (i (i . . .)[d][1,2i] (i (i-1 . . .)[d][2i+1,4i-2] (i)[c][4i-1,4i])[e][2i+1,4i])[f][1,4i]

We proof that our hypothesis is true when n = i+1:

According to Definition 1, the length of plan trace P_{i+1} is 4i+4 (where the first 2i+2 actions move a stack of i+1 blocks from a block X onto the table while inverting the order, and the second 2i+2 actions move the stack of i+1 blocks from the table onto the table while inverting the order again), and there are 2i2+1 curriculum steps in C_{i+1} .

Curriculum step 1 to 2i-1 of C_{i+1} is analogous to curriculum step 1 to 2i-1 of C_i as action 1 to 2i of P_{i+1} and action 1 to 2i of P_i move a stack of i blocks from a block onto the table while inverting the order. Therefore, we learn the following methods from step 1 to 2i-1 of C_{i+1} in M_{i+1}:\{1: `Make-1Pile`[a][1,2], 2: `Make-2Pile`[b][3,4], 3: `Make-2Pile`[d][1,4], 4: `Make-3Pile`[b][5,6], 5: `Make-3Pile`[d][1,6], . . . , 2i-2: `Make-iPile`[b][2i-1,2i], 2i-1: `Make-iPile`[d][1,2i]\}, in hierarchical form: (i . . .)[d][1,2i]

Two more methods are learned from curriculum step 2i and 2i+1 of C_{i+1}: {2i: `Make-i+1Pile`[b][2i+1,2i+2], 2i+1: `Make-i+1Pile`[d][1,2i+2]}, analogous to how `Make-2Pile`[b][3,4] and `Make-2Pile`[d][1,4] are learned.

Curriculum step 2i+2 to 4i-2 of C_{i+1} is analogous to curriculum step 2i to 4i-4 of C_i as action 2i+3 to 4i of P_{i+1} and action 2i+1 to 4i-2 of P_i both move a stack of i-1 blocks from a block onto the table while inverting the order. Therefore, we learn the following methods from step 2i+2 to 4i-2 of C_{i+1} in M_{i+1}: {2i+2: `Make-1Pile`[a][2i+3,2i+4], 2i+3: `Make-2Pile`[b][2i+5,2i+6], 2i+4: `Make-2Pile`[d][2i+3,2i+6], 2i+5: `Make-3Pile`[b][2i+7,2i+8], 2i+6: `Make-3Pile`[d][2i+3,2i+8], ..., 4i-1: `Make-iPile`[b][4i+1,4i+2], 4i: `Make-iPile`[d][2i+3,4i+2]}, in hierarchical form: (i ...)[d][2i+3,4i+2] , which is similar to (i ...)[d][1,2i] with shifted indices.

The following methods are learned from curriculum step 4i+1 to 4i+3 of C_{i+1}: {4i+1: `Make-i+1Pile`[c][4i+3,4i+4], 4i+2: `Make-i+1Pile`[e][2i+3,4i+4], 4i-3: `Make-i+1Pile`[f][1,4i+4]}, analogous to how `Make-2Pile`[c][7,8], `Make-2Pile`[e][5,8] and `Make-2Pile`[f][1,8] are learned.

Therefore, we learn: {1: `Make-1Pile`[a][1,2], 2: `Make-2Pile`[b][3,4], 3: `Make-2Pile`[d][1,4], 4: `Make-3Pile`[a][5,6], 5: `Make-3Pile`[c][1,6], ..., 2i-2: `Make-iPile`[b][2i-1,2i], 2i-1: `Make-iPile`[d][1,2i], 2i: `Make-i+1Pile`[b][2i+1,2i+2], 2i+1: `Make-i+1Pile`[d][1,2i+2], 2i+2: `Make-1Pile`[a][2i+3,2i+4], 2i+3: `Make-2Pile`[b][2i+5,2i+6], 2i+4: `Make-2Pile`[d][2i+3,2i+6], 2i+5: `Make-3Pile`[b][2i+7,2i+8], 2i+6: `Make-3Pile`[d][2i+3,2i+8], ..., 4i-3: Make-i-1Pile[b][4i-1,4i], 4i-2: Make-i-1Pile[d][2i+3,4i], 4i-1: `Make-iPile`[b][4i+1,4i+2], 4i: `Make-iPile`[d][2i+3,4i+2], 4i+1: `Make-i+1Pile`[c][4i+3,4i+4], 4i+2:

`Make-i+1Pile`[e][2i+3,4i+4], 4i-3: `Make-i+1Pile`[f][1,4i+4]}, , in hierarchical form: (i+1 (i+1 (i . . .)[d][1,2i] (i+1)[b][2i+1,2i+2])[d][1,2i+2] (i+1 (i . . .)[d][2i+3,4i+2] (i+1)[c][4i+1,4i+2])[e][2i+3,4i+4])[f][1,4i+4].

□

Bibliography

[1] Chad Hogg, Héctor Muñoz Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, page 950–956, 2008.

[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48, 2009.

[3] Allen Newell and George Ernst. The search for generality. In *Proc. IFIP Congress*, volume 65, pages 17–24, 1965.

[4] Drew McDermott. A temporal logic for reasoning about processes and plans. *Cognitive science*, 6(2):101–155, 1982.

[5] 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.

[6] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[7] J Stuart et al. Artificial intelligence a modern approach third edition, 2010.

[8] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[9] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

[10] Håkan LS Younes and Michael L Littman. Ppddl1. 0: The language for the probabilistic part of ipc-4. In *Proc. International Planning Competition*, 2004.

[11] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. Learning hierarchical task models from input traces. *Computational Intelligence*, 32(1):3–48, 2016.

[12] Malik Ghallab, Dana Nau, and Paolo Traverso. The actor's view of automated planning and acting: A position paper. *Artif. Intell.*, 208:1–17, mar 2014. ISSN 0004-3702. doi: 10.1016/j.artint.2013.11.002. URL `https://doi.org/10.1016/j.artint.2013.11.002`.

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

[14] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Int. Res.*, 14(1):253–302, may 2001. ISSN 1076-9757.

[15] J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, KR'92, page 103–114, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602623.

[16] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[17] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[18] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 968–973, 1999.

[19] Edmund H. Durfee. Distributed problem solving and planning. In Michael Luck, Vladimír Mařík, Olga Štěpánková, and Robert Trappl, editors, *European Agent Systems Summer School*, pages 118–149, 2001.

[20] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[21] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Munoz-Avila, and J.W. Murdock. Applications of shop and shop2. *IEEE Intelligent Systems*, 20(2):34–41, 2005. doi: 10.1109/MIS.2005.20.

[22] 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 Antelligence Research*, 20:379–404, 2003.

[23] Dana Nau. Game applications of htn planning with state variables, 2013. URL `http://bitbucket.org/dananau/pyhop`.

[24] Dana Nau, Yash Bansod, Sunandita Patra, Mark Roberts, and Ruoxi Li. Gtpyhop: A hierarchical goal+ task planner implemented in python. *HPlan 2021*, page 21, 2021.

[25] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. The panda framework for hierarchical planning. *KI-Künstliche Intelligenz*, 35(3):391–396, 2021.

[26] David E Smith, Jeremy Frank, and William Cushing. The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, volume 31, 2008.

[27] Filip Dvorak, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab. A flexible anml actor and planner in robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*, 2014.

[28] Vikas Shivashankar, Ugur Kuter, Dana Nau, and Ron Alford. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 981–988, 2012.

[29] Vikas Shivashankar, Ron Alford, Ugur Kuter, and Dana Nau. The godel planning system: A more perfect union of domain-independent and hierarchical planning. IJCAI '13, page 2380–2386. AAAI Press, 2013. ISBN 9781577356332.

[30] Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau. Integrating acting, planning, and learning in hierarchical operational models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 478–487, 2020.

[31] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[32] Ruoxi Li, Mark Roberts, Morgan Fine-Morris, and Dana Nau. Teaching an htn learner. In *Proceedings of the 5th ICAPS Workshop on Hierarchical Planning (HPlan 2022)*, pages 68–72, 2022.

[33] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artif. Intell.*, 42(2–3):363–391, mar 1990. ISSN 0004-3702. doi: 10.1016/0004-3702(90)90059-9. URL https://doi.org/10.1016/0004-3702(90)90059-9.

[34] Ruoxi Li, Dana Nau, Mark Roberts, and Morgan Fine-Morris. Automatically learning htn methods from landmarks. *FLAIRS-24*, 2024.

[35] Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In *ECAI*, page 335–340, 2010.

[36] Jendrik Seipp, Álvaro Torralba, and Jörg Hoffmann. PDDL generators. https://doi.org/10.5281/zenodo.6382173, 2022.

[37] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *IJCAI*, page 968–973, 1999.

[38] Damir Lotinac and Anders Jonsson. Constructing hierarchical task models using invariance analysis. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ECAI'16, page 1274–1282, NLD, 2016. IOS Press. ISBN 9781614996712. doi: 10.3233/978-1-61499-672-9-1274. URL https://doi.org/10.3233/978-1-61499-672-9-1274.

[39] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Munoz-Avila. Learning htn method preconditions and action models from partial observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, page 1804–1809, 2009.

[40] Hankz Hankui Zhuo, Héctor Munoz-Avila, and Qiang Yang. Learning hierarchical task network domains from partially observed plan traces. *Artificial intelligence*, 212:134–157, 2014.

[41] Ke Xu and Hector Muñoz-Avila. A domain-independent system for case-based task decomposition without domain theories. In *Proceedings of the 20th National Conference on AI*, pages 234–240, 2005.

[42] Okhtay Ilghami, Hector Munoz-Avila, Dana S Nau, and David W Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 337–344, 2005.

[43] Philippe Hérail and Arthur Bit-Monnot. Leveraging Demonstrations for Learning the Structure and Parameters of Hierarchical Task Networks. In *FLAIRS*, volume 36, 2023. doi: 10.32473/flairs.36.133327. URL `https://journals.flvc.org/FLAIRS/article/view/133327`.

[44] José A Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Learning HTN domains using process mining and data mining techniques. In *ICAPS Workshop on Generalized Planning*, 2017.

[45] Nan Li, Subbarao Kambhampati, and Sungwook Yoon. Learning probabilistic hierarchical task networks to capture user preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, page 1754–1759, 2009.

[46] Sriram Gopalakrishnan, Héctor Munoz-Avila, and Ugur Kuter. Word2HTN: Learning task hierarchies using statistical semantics and goal reasoning. In *Working Notes of 4th Workshop on Goal Reasoning at IJCAI-2016*, 2016.

[47] Morgan Fine-Morris, EDU Bryan Auslander, Michael W Floyd, Greg Pennisi, Héctor Muñoz-Avila, and EDU Kalyan Moy Gupta. Learning hierarchical task networks with landmarks and numeric fluents by combining symbolic and numeric regression. In *Proceedings of the 8th Annual Conference on Advances in Cognitive Systems*, 2020.

[48] Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *ICRA*, pages 5469–5476, 2016. doi: 10.1109/ICRA.2016.7487760.

[49] Kevin Chen, Nithin Shrivatsav Srikanth, David Kent, Harish Ravichandar, and Sonia Chernova. Learning Hierarchical Task Networks with Preferences from Unannotated Demonstrations. In *CoRL*, pages 1572–1581, 2021. URL `https://proceedings.mlr.press/v155/chen21d.html`.

[50] Pat Langley. Learning hierarchical problem networks for knowledge-based planning. In *Proceedings of the 31st International Conference on Inductive Logic Programming*, 2022.

[51] Howard E Shrobe. Hierarchical problem networks for knowledge-based planning. In *Proceedings of the 9th Annual Conference on Advances in Cognitive Systems*, 2021.

[52] Christopher W. Geib. Delaying commitment in plan recognition using combinatory categorial grammars. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, page 1702–1707, 2009.

[53] Christopher Geib and Pavan Kantharaju. Learning combinatory categorial grammars for plan recognition. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, page 3007–3014, 2018.

[54] Pavan Kantharaju. *Learning Decomposition Models for Hierarchical Planning and Plan Recognition*. PhD thesis, Drexel University, Philadelphia, Pennsylvania, USA, 2021.

[55] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, pages 51–68, 2005.

[56] Pat Langley, Dongkyu Choi, Roland Olsson, and Ute Schmid. Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, 7(3), 2006.

[57] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 665–672, 2006.

[58] Dedre Gentner and Arthur B Markman. Defining structural similarity. *The Journal of Cognitive Science*, 6(1):1–20, 2006.

[59] Dedre Gentner and Kenneth D Forbus. Computational models of analogy. *Wiley interdisciplinary reviews: cognitive science*, 2(3):266–276, 2011.