Learning Hierarchical Models from Demonstrations for Deliberate Planning and Acting

Philippe Hérail

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *12/06/2024* **par :**

Philippe HÉRAIL

Learning Hierarchical Models from Demonstrations for Deliberate Planning and Acting

---

JURY

Dana S. NAU	Professor Emeritus	Rapporteur
Damien PELLIER	Maître de Conférences	Rapporteur
Thierry SIMÉON	Directeur de Recherche	Président du Jury
Jane Jean KIAM	Juniorprofessorin	Membre du Jury
Simon LACROIX	Directeur de Recherche	Directeur de Thèse
Arthur BIT-MONNOT	Maître de Conférences	Directeur de Thèse

École doctorale et spécialité :
EDSYS : Informatique et Robotique
Unité de Recherche :
LAAS-CNRS
Directeur(s) de Thèse :
Simon LACROIX et *Arthur BIT-MONNOT*
Rapporteurs :
Dana S. NAU et *Damien PELLIER*

Abstract

The development of autonomous agents, especially embodied agents such as robots, requires complex architectures operating at different levels of abstraction. Given the complexity of real environments, hand-crafting all the models used at the different levels quickly becomes impractical. In recent years, there has been a growing body of work focusing on learning such models at the sensorimotor level, i.e. for perception and basic motor capabilities. However, the same cannot be said for high-level models enabling deliberative functions.

Among such high-level models, we will focus our attention on Hierarchical Task Networks (HTNs), which are a common planning formalism used in many practical applications, from video-games to robotic agents. Presently, designing HTN models remains a mostly manual task, which requires expertise both of the application domain and of the systems used for hierarchical planning. While some approaches do exist for learning HTNs, they suffer from some limitations, mainly in the structure of the domains that can be learned or in the required data annotation.

In this thesis, we will propose a technique for learning HTNs with multiple hierarchy levels with minimal annotation work required. To this end, we will propose two main contributions: a procedure for learning HTN *structures* from demonstrations and one for learning their *parameters* from these same demonstrations.

The structure learning approach will leverage frequent pattern mining to detect interesting behavioural patterns to abstract in the demonstrations, which we couple with an existing goal regression algorithm. The quality of a given HTN structure during the search will be evaluated through a novel metric based on the Minimum Description Length (MDL) principle to use as an efficient proxy for planning performance.

In addition, we propose a new method for identifying a sensible set of parameters for HTNs, relying on a MAX-SMT approach, which can be applied to most HTN models. Coupling our contributions for learning an HTN model structure and the identification of its parameters allows us to produces complete HTN models which we evaluate on standard benchmarks of the HTN planning community.

Remerciements

Si j'écris ces quelques lignes, c'est que je suis arrivé au terme de ma thèse. Celle-ci n'a pas été une aventure de tout repos, et je tiens à remercier toutes les personnes qui m'ont accompagnées. Je resterai bref, les principaux concernés n'ayant pas besoin de lire ces lignes pour se reconnaitre.

En premier lieu, je souhaite remercier ma direction de thèse. Simon, pour m'avoir fait confiance durant cette thèse et avoir accepté de prendre part à mon encadrement, malgré l'éloignement entre mon travail et ses thématiques de recherche. Arthur, pour son accompagnement tout au long de cette aventure, qui lui a parfois couté quelques nuits de sommeil. Merci de m'avoir poussé à toujours m'améliorer et de m'avoir permis de mettre en avant la valeur de mes travaux. Merci également d'avoir su voir cette valeur là où j'avais parfois du mal à la trouver.

Je tiens ensuite remercier ma compagne Mathilde, pour m'avoir à la fois soutenu et supporté, en particulier durant les moments les plus difficiles de ce projet. Promis, je pourrai ressortir de ma grotte maintenant !

Je voudrais remercier également tous mes collègues du LAAS. Ceux présents au démarrage et ayant depuis vogué vers de nouveaux horizons, notamment Guilhem, Amandine et Antoine, qui m'ont permis de m'intégrer à l'équipe. En raison du télétravail imposé par une fameuse épidémie, dont je ne citerai par le nom, ce n'était pas une mince affaire. Mes camarades de "génération", Jérémy, Anthony et Ilinka, qui aujourd'hui ont tous soutenu – avec brio – leurs travaux. Guillaume, arrivé dans l'équipe bien avant moi et toujours présent lors de mon départ. Merci pour le soutien, et pour les pauses café pour m'ayant arraché à mon écran quelques minutes, même dans les périodes les plus intenses. Simon, dont les cheveux ont décidé ne pas le suivre dans cette aventure, merci pour ton énergie (moins pour tes questions d'automatique). Merci également à William, Smail, Laure, Bastien, Lou, Stéphy, Rebecca, Yannick, Adrien, Gianluca, Dario, Phani, Marcel, Léo, Fadma, Marie, Émilie, Roland, Émile et tous les autres que j'ai rencontrés au cours de ces quelques années.

Merci aussi aux permanents de l'équipe RIS, en particulier Malik, pour avoir pris le temps de discuter de mes idées et m'avoir aidé à prendre du recul sur celles-ci, et à Félix, grand maitre de la machine à café sans qui certaines phases de rédaction auraient sans aucun doute trainé plus en longueur.

J'ai également une pensée pour les personnes qui m'ont permis de découvrir le monde de la recherche : à la fois mes enseignants de l'INSA, mais surtout Daniela et Gaël qui m'ont permis d'effectuer un premier stage au LAAS avant ma thèse. Sans cette expérience, je ne me serai probablement pas lancé dans l'aventure

Merci aussi à tous mes amis de l'INSA, d'avoir été présents pour moi, malgré mes apparitions parfois un peu rares durant ces quatre années.

Merci également à ma famille, pour m'avoir offert quelques nécessaires bulles d'air afin de me permettre de souffler un peu au cours de ce marathon.

Contents

Acronyms vii

List of Figures ix

List of Tables xiii

List of Algorithms xv

1 Introduction 1

2 Learning Task Models for Acting in a Robotics Context 5
- 2.1 Introduction . . . 5
- 2.2 Automated Planning: Definitions . . . 6
  - 2.2.1 Classical Planning . . . 6
  - 2.2.2 Hierarchical Planning . . . 9
- 2.3 A Learner for Parameterized Hierarchical Task Networks from Demonstrations . 17
  - 2.3.1 Learning Problem . . . 18
  - 2.3.2 The HTN Domains Considered in this Thesis . . . 20
- 2.4 Related Work . . . 22
  - 2.4.1 Learning Action Models . . . 22
  - 2.4.2 Learning Hierarchical Models . . . 25
  - 2.4.3 Other Hierarchical Models . . . 30
  - 2.4.4 Generalized Planning . . . 32
  - 2.4.5 Other Relevant Approaches . . . 34
- 2.5 A Multi-Stage Iterative Learning Procedure . . . 35

3 Learning Hierarchical Task Networks Structure from Demonstrations 37
- 3.1 Introduction . . . 37
- 3.2 Generating Neighbours of HTN Structures . . . 39
  - 3.2.1 Goal Regression without Explicit Goals . . . 40
  - 3.2.2 Frequent Pattern Mining for Neighbour Generation . . . 46
  - 3.2.3 Simplifying a Candidate Structure . . . 51
- 3.3 Evaluating Candidate Models . . . 53
  - 3.3.1 The MDL Principle for HTN Structures Evaluation . . . 53
  - 3.3.2 Obtaining Decomposition Trees from HTN Structures and Action Sequences. . . . 58
- 3.4 The Complete Structure Search Algorithm . . . 61
- 3.5 Conclusion . . . 62

4 Learning Hierarchical Task Networks Parameters from Demonstrations 65
- 4.1 Introduction . . . 65
- 4.2 Learning Symbolic Parameters: Existing Approaches . . . 67
- 4.3 Generating Correct and Usable Parameters for a Given HTN Structure . . . 68
  - 4.3.1 Generating a Finite Candidate Parameter Set . . . 69
  - 4.3.2 Simplifying the Generated Candidate Sets . . . 74

4.4 Handling Recursive Task Definitions: the "*Loop-Until*" Pattern 79
4.4.1 Ensuring Consistency with the HTN Structure 84
4.4.2 Ensuring Compatibility with the Demonstrations 84
4.4.3 Minimizing Method Parameters Through Unification 85
4.5 Conclusion 86

5 Experimental Evaluation 87
5.1 Introduction 87
5.2 Planning Domains Presentation 87
5.3 Environment and Datasets 88
5.3.1 Environment 88
5.3.2 Datasets 90
5.4 Planning Performance 91
5.4.1 Rovers 91
5.4.2 Logistics 96
5.4.3 Childsnack 99
5.4.4 Satellite 102
5.4.5 Woodworking 105
5.5 Learning Times 108
5.6 Conclusion 113

6 Conclusion 115

A Résumé en Français 117

B Handling Recursive Task Definitions: Arbitrary Recursive Structures 121
B.1 A New Argument Propagation Procedure 121
B.1.1 Direct Recursions 121
B.1.2 Indirect and Independent Recursions 123
B.2 Parameter Minimization 126
B.2.1 Required Features in the MAX-SMT Solver 131
B.2.2 Defining Datatypes and Functions 131
B.2.3 Defining the Constraints 132
B.2.4 Defining the Optimization Objectives 137
B.3 Conclusion 138

C The Minimum Description Length Principle 141

D Notable Domains Used in Experiments 143

References 153

Acronyms

ASP Answer Set Programming. 25

BPMN Business Process Model and Notation. 35

BT Behaviour Tree. 1, 31, 32, 34, 35

CCG Combinatory Categorial Grammar. 31

CFG Context Free Grammar. 29, 31, 32, 36, 47, 141

CNF Chomsky Normal Form. 30

CP Constraint Programming. 25

DSL Domain Specific Language. 31, 34

EM Expectation Maximization. 31

FSM Finite State Machine. 1

GE Grammatical Evolution. 32

GP Genetic Programming. 32

HGN Hierarchical Goal Network. 26, 29, 30

HTN Hierarchical Task Network. i, ix–xiii, 5, 9–12, 14, 16–18, 20, 26–37, 39–56, 58–62, 65–69, 71–77, 79, 81, 86, 87, 89–91, 93, 96, 98, 101, 104, 107, 108, 113, 115, 119–121, 123, 126–129, 131, 141

ILP Inductive Logic Programming. 24

IPC International Planning Competition. xiii, 25, 29, 61, 87, 88, 90, 91, 96, 97, 102

LfD Learning from Demonstration. 22, 32

LTL Linear Temporal Logic. 32, 68

MAX-SAT maximum satisfiability. 28

MDL Minimum Description Length. i, 31, 36, 47, 49, 53, 62, 113, 115, 120, 141, 142

NLP Natural Language Processing. 29–31

PbD Programming by Demonstration. 22, 34

pCCG probabilistic Combinatory Categorial Grammar. 31

pCFG probabilistic Context Free Grammar. 31

pHTN probabilistic HTN. 28–30

RL Reinforcement Learning. 28, 34

RRL Relational Reinforcement Learning. 25

SCC Strongly Connected Component. 74, 124

TO totally-ordered. 11, 19, 20, 37, 39

List of Figures

1.1 A simplified view of an actor's architecture, adapted from [GNT14]. 2
1.2 A simple hierarchical domain for the travel example high-level task. 2

2.1 Location graph for the LOGISTICS planning problem example. 9
2.2 An intuitive view of an HTN hierarchy. Primitives are typeset as $\underline{\texttt{action}}(?x, \ldots, ?y)$. 10
2.3 A task network example. Here, all the t_i are task identifiers, and the directed edges represent the ordering of the task network elements. This ordering can be written as: $\{(t_1 \prec t_2), (t_1 \prec t_3), (t_1 \prec t_4), (t_2 \prec t_4), (t_4 \prec t_5)\}$. 11
2.5 A visual representation of the task networks of the methods in M. 13
2.6 Initial task network tn_I for the delivery planning problem. 14
2.8 A possible decomposition for the task $\texttt{deliver}(p_1, l_1)$, starting in state s_0. 16
2.9 A possible decomposition for the initial task network which is not a solution to the HTN planning problem. 17
2.10 A possible decomposition for the initial task network which is a solution to the HTN planning problem. 17
2.11 Example of initial task vocabulary and corresponding maximally abstract demonstration sets for a `deliver` task. All the demonstration sets correspond to the same sequence of primitive actions but are abstracted at different levels. Here, the considered optimality metric is the number of actions in the plan. Details of states omitted for clarity. 19
2.12 A simplified graphical representation for an HTN. Parameters, pre-conditions and effects omitted for clarity. 21
2.13 Illustration of the possible structures of the learned domain for a simple learning task with two demonstration of how to perform a task t. Note that for conciseness the parameters and pre-conditions of the task and methods are omitted. 21

3.1 Example of initial task vocabulary and corresponding maximally abstract demonstration sets for a `deliver` task and the corresponding simplified version without parameters. All the demonstration sets corresponds to the same sequence of primitive actions but are abstracted at different levels. Here, the considered optimality metric is the number of actions in the plan. Note how the removal of parameters also removes intermediate states, as they then do not contain any information. 38
3.2 HTN structure for the method group of $m_{2,2}$. 42
3.3 A possible decomposition of t into the subsequence $\langle a_2, a_3, a_4 \rangle$. 43
3.4 Demonstration set used as an example, recalled from Figure 2.11. Only actions are represented, as states are not used in this algorithm. 45
3.6 Possible HTNs generated using the modified HTN-MAKER algorithm and random sampling. 46
3.7 Grammar for the generated patterns, using BNF syntax. 48
3.8 A HTN without preconditions or parameters and an equivalent grammar. Note that this grammar is not regular, as can be proved using the pumping lemma for regular languages. 48

3.9 Hierarchical representation of the pattern $p = \langle a^+ b \rangle$. Here, len($p$) = 2, only counting the a and b symbols. 49
3.10 Pattern substitution example for a pattern $p = \langle a^+ b \rangle$. 49
3.11 Example of possible HTN structure generation using the HTN-Maker-based operator on a demonstration with the abstracted frequent patterns as shown in Figure 3.10. The learned methods are a trivial lookup of all the demonstrations, but are enough to present the sharing of behaviours afforded by the use of pattern mining. 50
3.12 HTN before simplification. 52
3.13 HTN after simplification. 52
3.14 An HTN structure and the equivalent set of grammar rules. 54
3.16 Model length calculation for the domain presented in Figure 3.17b. 56
3.17 Illustration of the possible structures of the learned domain for a simple learning task with two demonstration of how to perform a task t. 57
3.19 Base HTN and the associated one for plan verification. 59
3.21 Base HTN and the associated one for plan verification. 60

4.1 An example of HTN without learned parameters and the same HTN with ideal parameters. 66
4.2 Example HTN structure and corresponding subhierarchies. Here, t_{top} is a demonstrated abstract task and t_s is a learned abstract tasks, while the other tasks are primitive. This example presents the case where we have an incomplete HTN domain where only the primitive and demonstrated abstract tasks' arguments are known, with ? used to denote task and methods where the parameters are unknown. 69
4.3 Example of argument superset generation for t_s. 72
4.4 Extracted parameters for t_{top} 72
4.5 Example of argument propagation in a decomposition tree for a demonstration of $t_{top}(a_1, a_2)$ as the sequence $\langle t_{p_1}(d_1), t_{p_2}(e_1, e_2) \rangle$ 72
4.6 Propagation graph for an HTN. 73
4.7 A HTN with multiple recursions in different methods, and the corresponding propagation graph 74
4.8 An incomplete HTN and the corresponding propagation graph. Colours and line styles used in all subfigures are used to clarify the edge labelling in the propagation graph. 75
4.9 Example of decomposition tree with instantiation identifiers for a demonstration $\texttt{dlv} : \langle \texttt{mv}, \texttt{ld}, \texttt{mv}, \texttt{uld} \rangle$ and presented HTN structure. 76
4.10 Example of extended subhierarchy used for downward information propagation. . 79
4.11 Subhierarchy for a *goto* pattern. Preconditions omitted for clarity. 80
4.12 Possible example trace and corresponding decomposition tree example for the *goto* task, with the subhierarchy parameters extracted using the method presented in previous section. Colours are used to highlight identical constants. 80
4.13 Parameters modification for recursive tasks. 81
4.14 Generic decomposition tree for a recursive hierarchy. 82
4.15 Extracted subhierarchy for a *goto* task before recursion processing. 82
4.16 Possible example trace and corresponding decomposition tree example for the *goto* task. Colours are used to highlight identical constants. 83

5.1 Planning time cumulative distribution for the ROVERS domain, for the best parameterized and non-parameterized domains. . . . 92
5.2 Planning time cumulative distribution for the ROVERS domain, focusing on parameterized and non-parametrized HTNs. . . . 93
5.3 Planning time box plot, limited to the best and reference models and the instances solved by both of them. . . . 94
5.4 Plan length box plot, limited to the best and reference models and the instances solved by both of them. . . . 94
5.5 Impact of training demonstration set size on coverage. Note that the x-axis scale is not linear. . . . 95
5.6 Planning time cumulative distribution for the LOGISTICS domain, for the best parameterized and non-parameterized domains. . . . 97
5.7 Planning time cumulative distribution for the LOGISTICS domain, focusing on parameterized and non-parametrized HTNs. . . . 98
5.8 Distribution of the plan length over the set of commonly solved instances (100/100) in the LOGISTICS domain. . . . 99
5.9 Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear. . . . 99
5.10 Planning time cumulative distribution for the CHILDSNACK domain, for the best parameterized and non-parameterized domains. . . . 100
5.11 Planning time cumulative distribution for the CHILDSNACK domain, focusing on parameterized and non-parametrized HTNs. . . . 101
5.12 Planning time cumulative distribution for the SATELLITE domain, for the best parameterized and non-parameterized domains. . . . 103
5.13 Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear. . . . 103
5.14 Planning time cumulative distribution for the SATELLITE domain, focusing on parameterized and non-parametrized HTNs. . . . 104
5.15 Planning time cumulative distribution for the WOODWORKING domain, for the best parameterized and non-parameterized domains. . . . 106
5.16 Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear. . . . 106
5.17 Planning time cumulative distribution for the WOODWORKING domain, focusing on parameterized and non-parametrized HTNs. . . . 107
5.18 Structure learning time for all the tested domains, limited to the *most recursive* neighbour generation. . . . 109
5.19 Parameterization time for all the tested domains, limited to the *most recursive* neighbour generation. . . . 110
5.20 Structure learning time for all the tested domains, comparing different neighbour generation modes. . . . 111
5.21 Parameterization time for all the tested domains, comparing different neighbour generation modes. . . . 112

A.1 Une vue simplifiée de l'architecture d'un acteur, adapté de [GNT14]. . . . 118
A.2 Un domaine hiérarchique simplifié pour la tâche de haut niveau de notre exemple de voyage. . . . 119

B.1 New argument propagation scheme algorithm example on a simple goto task. In each step, orange arguments are the ones that were propagated upwards and blue ones the ones that were added by updating the subtasks. . . . 122
B.2 An HTN with an indirect recursion. Here, t is recursive through t_r and t_s is a non-recursive abstract task. . . . 123
B.3 Naive application of the propagation algorithm, focusing on the subhierarchy of task t. . . . 124
B.4 Naive application of the propagation algorithm, focusing on the subhierarchy of task t_r. . . . 124
B.5 Task dependency graph for the hierarchy in Figure B.2. . . . 125
B.6 Base subhierarchies for the recursion group $\{t, t_r\}$ after processing t_s and generating the subtask arguments. . . . 126
B.7 Application of Algorithm B.4 to the HTN structure in Figure B.2, step 1. . . . 126
B.8 Application of Algorithm B.4 to the HTN structure in Figure B.2, step 2. . . . 127
B.9 Application of Algorithm B.4 to the HTN structure in Figure B.2, step 3. . . . 127
B.10 Application of Algorithm B.4 to the HTN structure in Figure B.2, step 4. . . . 127
B.11 Application of Algorithm B.4 to the HTN structure in Figure B.2, step 5. . . . 128
B.12 HTN structure with two independent recursion groups, one of which is indirect, and associated dependency graphs. . . . 128
B.13 Base subhierarchies for the recursion group $\{t, t_r\}$ after processing *goto* and generating the subtask arguments . . . 128
B.14 The goto task, with its propagated parameters modified to be converted to MAX-SMT constants. . . . 129
B.15 Possible example trace and corresponding decomposition tree example for the goto task. Colours are used to highlight identical constants. . . . 130

C.1 C programs to generate the sequences presented earlier. . . . 142

D.1 Graphical representation of the IPC domain in Rovers. . . . 144
D.2 Graphical representation of the best learned domain in Rovers. . . . 145
D.3 Graphical representation of the reference domain in Logistics. . . . 146
D.4 Graphical representation of the adapted IPC domain in Logistics. . . . 147
D.5 Graphical representation of the best learned domain in Logistics. . . . 148
D.6 Graphical representation of the IPC domain in Childsnack. . . . 149
D.7 Graphical representation of the best learned domain in Childsnack. . . . 150
D.8 Graphical representation of the IPC domain in Satellite. . . . 151
D.9 Graphical representation of the best learned domain in Satellite. . . . 152

List of Tables

2.1 Summary of the action model learning approaches. Obs. and ND stand *Observability* and *Non-Determinism*, respectively. 23
2.2 Summary of the main HTN learning approaches and comparison with our proposed approach. 27

3.1 Description length for the examples presented in Figure 3.17. 56

4.1 Constraints used in the MAX-SMT problem. Conditions are fully expanded to remove quantifiers, and a constraint is added for each expansion of the corresponding condition. Quantifiers are also expanded in each constraint as the chosen solver does not support optimization with quantifiers. Upper constraints are hard while the lower one is soft. 78

5.1 The different parameters that govern the search procedure and their domains. . . 89
5.2 Number of International Planning Competition (IPC) planning instances and generated demonstration traces for each domain. 91
5.3 Learning evaluation parameters for the ROVERS domain. 91
5.4 Coverage for the ROVERS domain, restricted to the best four parameterized and non-parameterized domains. 92
5.5 Learning evaluation parameters for the LOGISTICS domain. 96
5.6 Coverage for the LOGISTICS domain, restricted to the best four parameterized and non-parameterized domains. 96
5.7 Learning evaluation parameters for the CHILDSNACK domain. 100
5.8 Coverage for the CHILDSNACK domain, restricted to the best four parameterized and non-parameterized domains. 100
5.9 Learning evaluation parameters for the SATELLITE domain. 102
5.10 Coverage for the SATELLITE domain, restricted to the best four parameterized and non-parameterized domains. 102
5.11 Learning evaluation parameters for the WOODWORKING domain. 105
5.12 Coverage for the WOODWORKING domain, restricted to the best four parameterized and non-parameterized domains. 105

B.1 Example of direct evidence constraints that can be generated 133
B.2 Example of direct evidence constraints that can be generated 134

List of Algorithms

2.1 HTN Search - High-Level Process . . . 36
3.1 Extract Methods(t, d) . . . 40
3.2 Generate Neighbours HTN-Maker(D, H) - Initial Version . . . 43
3.3 Generate Neighbours HTN-Maker(D, H) . . . 44
3.4 Find Best Patterns(D_i, k, l, H_B) . . . 51
3.5 Gen Best Neighbour(H, D) . . . 62
3.6 Find Best Structure(H, D, k, l) . . . 62
4.1 Parameter Superset Generation . . . 70
4.2 Propagate Args Upwards(h_t) . . . 71
4.3 Update Subtasks Args(h_t, Π_{new}) . . . 71
4.4 Parameter Removal . . . 79
A.1 Processus global d'apprentissage de HTN . . . 119
B.1 Argument Propagation For Recursion . . . 122
B.2 Propagate Arguments Upwards(h) . . . 123
B.3 Update Subtasks Basic(h, A) . . . 123
B.4 Argument Propagation For Recursion - Improved . . . 125
B.5 Update Subtasks(h, A) . . . 126

Chapter 1

Introduction

In order to act purposefully within their environment, autonomous agents typically need to achieve *high-level tasks*, relying on a set of elementary *skills* to perform actions in their environment.

Each skill represents an elementary operation, such as picking up an object or putting it down. These skills potentially abstract over the lowest levels of control primitives (such as motor controls in the case of a robot), and need not be a fixed sequence of such primitives. Indeed, they may have to adapt to their execution context: for example, we can consider using different movements to pick up an empty cup and a liquid-filled bowl. These skills then need to be combined to achieve the desired high-level behaviours.

To give a simple example of a high-level task that can be achieved with some combination of skills, let us consider an agent travelling to a conference in Paris from Toulouse, where going by train and by plane are the two available options. We can define the following set of skills for our agent: {`BuyTrainTckt`, `BuyPlaneTckt`, `WalkTo`, `TakeCab`, `TakeTrain`, `TakePlane`}. These skills can also be parameterized to specify their application context, such as the destination for the `WalkTo` skill. We can then imagine choosing the following sequence of skills to achieve our top-level task of travelling to the conference venue in Paris:

$$\langle \texttt{BuyTrainTckt}(Paris), \texttt{WalkTo}(Station), \texttt{TakeTrain}(Paris), \texttt{TakeCab}(Venue) \rangle$$

While the agent skills may simply be combined *reactively* to obtain such a sequence, using policies like simple Finite State Machines (FSMs), neural network-based policies or Behaviour Trees (BTs), this approach may fall short when long horizons have to be considered to achieve a given task. In our travel example, some of the steps require planning ahead: for example, using the `BuyTrainTckt` action requires knowing at least part of the future actions (here, knowing that we want to take the train and not fly to the conference city). We therefore situate ourselves within the *deliberative acting* paradigm, as presented by Ghallab, Nau and Traverso [GNT14]. In this framework, the authors define an action as something the agent does to change its state or its environment, while deliberating is a *reasoning* process that leads the agent to choose to perform one action or the other considering its long term goals, often through the use of *planning techniques*.

In essence, we can view an actor interacting with its environment as presented in Figure 1.1. Here, we can separate an *execution platform* and a *deliberation* component. The first one is tasked with transforming the skill commands coming from the deliberation component into actuations of the actor's that allow executing the commands in its environment. It is also tasked with converting the raw sensor data into representations usable by the deliberation functions. The deliberation component can be divided into two components, *planning* and *acting*. The planning component mainly receives the high level activities to be performed, and generates long-term strategies for them, often considering an abstracted environment, in order to guide the acting component. The acting component is in charge of carrying out the actual execution of the plan, using the guidance from the planner while monitoring the execution of the actions

to handle exogenous events and skill failures.

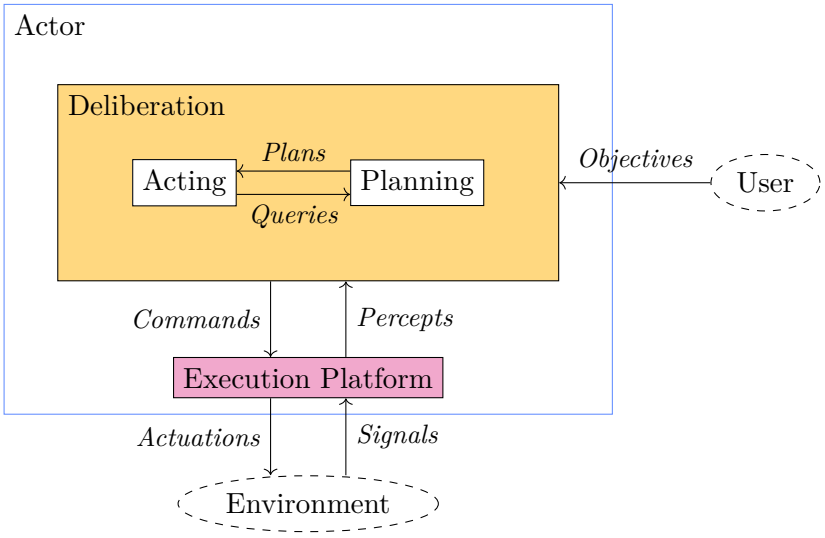


Figure 1.1: A simplified view of an actor's architecture, adapted from [GNT14].

It can be noted that real architectures for an actor are more complex, such as the robotic architecture presented by Lemaignan et al. [Lem+17], exhibiting multiple components in the deliberation layer. However, we can still distinguish both layers, as well as the planning and acting components.

Considering different robotic execution systems [DI00; Ing+96; MCA22; SdSP06; TB22], we observe that they rely on *models* of the tasks and actions to be performed, often hierarchical in nature to efficiently describe complex tasks. Reusing our travel example, models of the skills would allow specifying applicability conditions and effects. For example, let us consider our `WalkTo`(*Station*) skill. It could be specified to be only applicable if there is a pedestrian path from our current location to the train station, and that it is less than three kilometres away. The effects of this skill would be to have the agent be at the train station. If we wanted to have a model of our top level task, then we could have a hierarchical structure as presented in Figure 1.2, where we have two different options to achieve the task, travelling either by train or by plane.

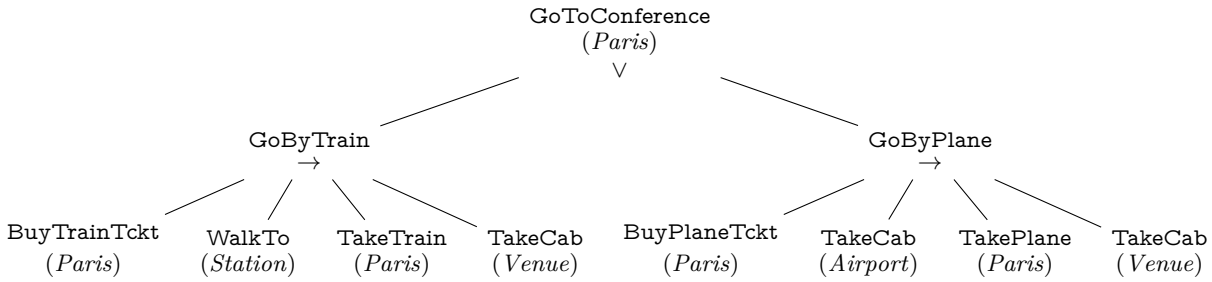


Figure 1.2: A simple hierarchical domain for the travel example high-level task.

Hierarchical formalisms have been developed which provide ways to decompose a top level task or goal down to the level of the agent's skills. These models are often complex to specify by hand, and while learning for autonomous agents has gained a lot of traction in the last decade, most approaches are focusing on the sensorimotor level [CSL21; Kle+20]. Comparatively,

learning complex high-level task models has been less studied. Learning these high-level task models will be the focus of this thesis, in the following context:

- *Skill availability*: we assume that the list of available elementary capabilities (skills) of the agent is known. These can be either learned from existing approaches or hand-programmed.
- *Demonstration based*: we assume that the learning agent is given a set of demonstrations of how to achieve the task to be learned. These demonstrations would take the form of sequences of skills, showcasing some combination that achieves the desired behaviour in a given state.

We will specifically focus on learning hierarchical models, due to them being easier to understand by human experts and for their ability to constrain the (potentially very large) search space of all the possible plans that an agent may choose to enact, while still reproducing the demonstrated behaviour. Furthermore, we situate ourselves in a context where demonstration data is costly to obtain, such as learning for robotic agents, directing our work towards a sample-efficient method.

Chapter 2

Learning Task Models for Acting in a Robotics Context

Contents

2.1 Introduction . . . 5
2.2 Automated Planning: Definitions . . . 6
2.2.1 Classical Planning . . . 6
2.2.2 Hierarchical Planning . . . 9
2.3 A Learner for Parameterized Hierarchical Task Networks from Demonstrations . 17
2.3.1 Learning Problem . . . 18
2.3.1.1 Inputs . . . 18
2.3.1.2 Goals . . . 19
2.3.2 The HTN Domains Considered in this Thesis . . . 20
2.4 Related Work . . . 22
2.4.1 Learning Action Models . . . 22
2.4.2 Learning Hierarchical Models . . . 25
2.4.2.1 Macro Operators . . . 25
2.4.2.2 Hierarchical Planning Domains . . . 26
2.4.3 Other Hierarchical Models . . . 30
2.4.3.1 Grammar Inference . . . 30
2.4.3.2 Behaviour Trees . . . 31
2.4.4 Generalized Planning . . . 32
2.4.5 Other Relevant Approaches . . . 34
2.4.5.1 Programming by Example . . . 34
2.4.5.2 Process Mining . . . 35
2.5 A Multi-Stage Iterative Learning Procedure . . . 35

2.1 Introduction

While we have introduced the idea of *hierarchical models* in the introduction of this document, let us now give a more formal definition of these models. Specifically, we want to learn Hierarchical Task Networks (HTNs), a specific kind of hierarchical *planning* models. Therefore, we will present the planning formalism that we use in this document in the next section.

We will then present our learning problem, as well as the scope of our learner, mainly restricting it to totally-ordered Hierarchical Task Networks (HTNs). Next, we will review the existing approaches related to our goal, before finally presenting a high-level overview of our learning procedure.

2.2 Automated Planning: Definitions

Let us present some formal definitions of *classical* and *hierarchical* task planning. Because our focus is on learning planning domains in the context of deliberate acting in robotics, we will mainly adapt the definitions given by Ghallab, Nau and Traverso [GNT14] for classical planning and the formalism of Alford, Bercher and Aha [ABA15] for hierarchical planning.

Example 2.1 *Running example: a* LOGISTICS*-like domain.*
To illustrate this section, we will use as running example a simple LOGISTICS-like planning domain. This domain can be described in natural language as follows: a *truck* is tasked with delivering a *package* from one *location* to another. The actions available in this domain are as follows:

1. A *truck* t can move from a *location* l_1 to a *location* l_2 provided they are connected together.
2. A *package* p can be loaded in a *truck* t if both are at the same *location* and t is empty.
3. A *package* p can be unloaded from a *truck* t at the *location* where it is.

This natural-language definition will be made more formal as we define the relevant concepts.

2.2.1 Classical Planning

Before outlining the existing algorithms and classical planning systems, we will provide some definitions of *planning domains* and *planning problems* to formalize the notations.

The formalism we use is based on a quantifier-free first-order predicate logic $\mathcal{L} = (P, T, V, C)$ where:

- T is a set of *type* symbols. These types are used to subdivide the sets of constants C and variables V, typeset as *type-name*.
- C is a set of typed *constants*. Constants are used to refer to objects in the world, and are typeset *const-name*. Each constant is associated with at least one type in T. Multiple types associated with a single constant $c \in C$ can be used to create type hierarchies: for example, c may be associated with types *car* and *vehicle*
- V is a set of typed *variables*. Variables refer to objects that are not yet specified, and are typeset ?*var-name*. Each variable is associated with a single type in T, which restricts its domain (allowed values) to constants of the same type.
- P is a set of *predicate* symbols[1]. A predicate has an associated arity n which defines the number of its parameters variables. Predicates are typeset pred-name. Each parameter of a predicate is associated with a type.

From this, we can define atoms, literals, states and actions:

[1] Some formalisms allow state functions instead of predicates. Because only the latter are required to describe our approach, we restrict our formalism for simplicity.

Definition 2.1 (Atom)**.** An *atom* is an instantiation of a predicate with its parameters $\texttt{pred-name}(?v_1, \ldots, ?v_n)$. It is *ground* if all its parameters are constant and *lifted* if they contain variables. This definition of ground and lifted can be similarly extended to other construct using parameters

The set of all ground atoms represents all possible state features

Definition 2.2 (Literal)**.** A *literal* is an atom or its negation.

Definition 2.3 (State)**.** A *state* s is a set of ground atoms. It indicates which features of the state are true. Atoms absent from s are assumed to be false (closed world assumption).

Definition 2.4 (Action Template)**.** An *action template* is a tuple $a = (\text{head}(a), \text{pre}(a), \text{eff}(a))$.

- $\text{head}(a)$ is the name of the action as well as its typed parameters, such as $\texttt{act-name}(?x_1 : \texttt{type}_1, \ldots, ?x_n : \texttt{type}_n)$. We can separate $\text{head}(a) = (\text{name}(a), \text{args}(a))$, the name of a and its arguments, respectively. Action names uniquely identify an action.
- $\text{pre}(a)$ is a set of atoms called the *pre-conditions* of a.
- $\text{eff}(a)$ is a set of literals called the *effects* of a. We write $\text{eff}(a)^+$ and $\text{eff}(a)^-$ the subsets of atoms corresponding respectively to the positive and negative literals.

All the variables used in $\text{pre}(a)$ and $\text{eff}(a)$ must be in the parameters of a.

Definition 2.5 (Action Instance)**.** An *action instance* is an action template that has been grounded, i.e. all its parameters have been replaced with constants. An action instance a is said to be *applicable* in a given state s if and only if $\text{pre}(a)$ holds in s.

We can now present the definition of a classical planning domain:

Definition 2.6 (Classical Planning Domain)**.** A *classical planning domain* is a tuple $\Sigma = (\mathcal{L}, A)$, with A a finite set of *action templates*. The set of states S is implicitly defined over all subsets of all ground atoms.

For clarity in the remainder of this document, let us define A_g the set of all *ground* of actions. We can then define a partial function $\gamma : S \times A_g \to S$. This function is defined only for each pair $(s, a) \in S \times A_g$ where a is applicable (i.e. $\text{pre}(a)$ hold in s). $s' = \gamma(s, a)$ is the state in which the agent will end up after applying a, with $s' = (s \setminus \text{eff}(a)^-) \cup \text{eff}(a^+)$.

Example 2.2 *The planning domain for our* LOGISTICS*-like domain.*
Using the definitions presented up to this point, we can more formally present a *planning domain* associated with our LOGISTICS domain. We can define the set of types T:

$$T = \left\{ \begin{array}{l} \texttt{movable} \\ \texttt{truck} \\ \texttt{package} \\ \texttt{location} \end{array} \right\}$$

From here, assuming that $\{?o, ?t, ?p, ?l, ?l_1, ?l_2\} \subset V$ we can define the sets of predicates

P and actions A:

$$P = \left\{ \begin{array}{l} \mathsf{at}(?o : \textit{movable}, ?l : \textit{location}) \\ \mathsf{in}(?p : \textit{package}, ?t : \textit{truck}) \\ \mathsf{empty}(?t : \textit{truck}) \\ \mathsf{connected}(?l_1 : \textit{location}, ?l_2 : \textit{location}) \end{array} \right\}$$

$$A = \left\{ \begin{array}{rl} \mathsf{move} = & \left(\begin{array}{l} \text{head} : \mathsf{move}(?t : \textit{truck}, ?l_1 : \textit{location}, ?l_2 : \textit{location}) \\ \text{pre} : \{\mathsf{at}(?t, ?l_2), \mathsf{connected}(?l_1, ?l_2)\} \\ \text{eff} : \{\mathsf{at}(?t, ?l_2), \neg\mathsf{at}(?t, ?l_1)\} \end{array} \right) \\ \mathsf{load} = & \left(\begin{array}{l} \text{head} : \mathsf{load}(?p : \textit{package}, ?t : \textit{truck}, ?l : \textit{location}) \\ \text{pre} : \{\mathsf{at}(?t, ?l), \mathsf{at}(?p, ?l), \mathsf{empty}(?t)\} \\ \text{eff} : \{\mathsf{in}(?p, ?t), \neg\mathsf{at}(?p, ?l)\neg\mathsf{empty}(?t)\} \end{array} \right) \\ \mathsf{unload} = & \left(\begin{array}{l} \text{head} : \mathsf{unload}(?p : \textit{package}, ?t : \textit{truck}, ?l : \textit{location}) \\ \text{pre} : \{\mathsf{at}(?t, ?l), \mathsf{in}(?p, ?t)\} \\ \text{eff} : \{\mathsf{at}(?p, ?l), \mathsf{empty}(?t), \neg\mathsf{in}(?p, ?t)\} \end{array} \right) \end{array} \right\}$$

Definition 2.7 (Classical Planning Problem)**.** A *classical planning problem* is a tuple $\mathcal{P} = (\Sigma, s_0, g)$. Σ is a classical planning domain, s_0 is the *initial state* and g is a set of ground atoms called the *goal*.

Definition 2.8 (Plan)**.** A *plan* is a finite sequence of ground actions: $\pi = \langle a_0, \ldots, a_n \rangle$. The *length* of a plan π is the number of actions in the sequence.

Let us recursively define the successor state of an action a_i as $s_{i+1} = \gamma(s_i, a_i)$.

A plan is said to be *applicable* (or *executable*) in a state s_0 if, $\forall i \in [0..n], a_i$ is applicable in s_i. As a shorthand, we can write $\gamma(s_0, \pi) = s_{n+1}$ for denoting the application of plan π in the state s_0.

A plan $\pi = \langle a_0, \ldots, a_n \rangle$ is a *solution* to a planning problem $\mathcal{P}$ if π is applicable and $g \subset \gamma(s_0, \pi)$. For an arbitrary cost function that associates a cost to a given plan, a solution π is *optimal* if there exist no other solution plan π' such that $\text{cost}(\pi') < \text{cost}(\pi)$. A commonly used cost metric is the length of the plan.

For clarity in later parts of this document, we similarly define a solution trace as follows:

Definition 2.9 (Solution Trace)**.** A *solution trace* is a finite sequence of actions and states $\sigma = \langle s_0, a_0, s_1, \ldots, s_n, a_n, s_{n+1} \rangle$, where s_0 is the initial state of the corresponding problem and $\forall i \in [0, n], s_{i+1} = \gamma(s_i, a_i)$.

For a given action a_i, s_i is called the *pre-state* of a_i and s_{i+1} is called the *post-state*.

Example 2.3 *A planning problem for our* LOGISTICS*-like domain.*
We can now present a planning problem associated with the LOGISTICS planning domain presented earlier. Let us start by defining the set C of constants with their types (from T) for clarity:

$$C = \left\{ \begin{array}{r} t : \textit{truck}, \textit{movable} \\ p_1, p_2 : \textit{package}, \textit{movable} \\ l_0, l_1, l_2, l_3, l_4 : \textit{location} \end{array} \right\}$$

Using the previously defined sets P and A, assuming V contains all the necessary variables, we can then write $\mathcal{L} = (P, T, V, C)$. We have our planning domain as $\Sigma = (\mathcal{L}, A, \gamma)$ Defining a starting state s_0 and a goal state g as below, we have $\mathcal{P} = (\Sigma, s_0, g)$.

$$s_0 = \left\{\begin{array}{ll} \texttt{connected}(l_0, l_1) & \texttt{connected}(l_1, l_0) \\ \texttt{connected}(l_0, l_3) & \texttt{connected}(l_3, l_0) \\ \texttt{connected}(l_1, l_4) & \texttt{connected}(l_4, l_1) \\ \texttt{connected}(l_2, l_4) & \texttt{connected}(l_4, l_2) \\ \texttt{connected}(l_3, l_4) & \texttt{connected}(l_4, l_3) \\ & \\ \texttt{at}(t, l_0) & \texttt{empty}(t) \\ \texttt{at}(p_1, l_3) & \texttt{at}(p_2, l_4) \end{array}\right\}$$

$$g = \texttt{at}(p_1, l_1) \wedge \texttt{at}(p_2, l_2)$$

Note the symmetry of the `connected` predicates, which are used because we consider two-way roads. This starting state s_0 can be viewed as the graph presented in Figure 2.1, where edges in the graph show the `connected` predicates, and t, p_1 and p_2 are shown at their locations.

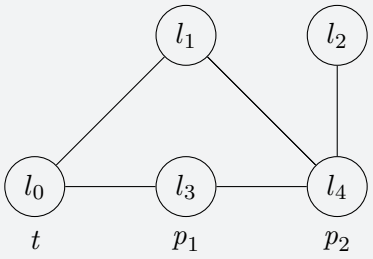


Figure 2.1: Location graph for the LOGISTICS planning problem example.

A possible solution to this problem is the following plan:

$$\pi = \langle \texttt{move}(t, l_0, l_3), \texttt{load}(p_1, t, l_3), \texttt{move}(t, l_3, l_0), \texttt{move}(t, l_0, l_1), \texttt{unload}(p_1, t, l_1), \\ \hookrightarrow \texttt{move}(t, l_1, l_4), \texttt{load}(p_2, t, l_4), \texttt{move}(t, l_4, l_2), \texttt{unload}(p_2, t, l_2) \rangle$$

Note that this plan is not unique, as it is easy to imagine a plan where we repeatedly move back and forth between two locations and arbitrary number of times. Such a plan would obviously achieve the same goal state with a different sequence of actions.

In practice, most classical planning domains and problems are expressed using the PDDL language [Gha+98].

2.2.2 Hierarchical Planning

Hierarchical planning reuses state and actions representation from classical planning for representing the world, and mainly differs on the way planning for a solution is done. The main formalism is Hierarchical Task Network (HTN) [EHN94] planning. Here, the idea is not to achieve some set of goals, but rather to perform a set of *tasks*, which need to be *refined* into

primitive actions through the use of *methods*.

We focus on lifted HTNs, and base our formalism on the one described by Alford, Bercher and Aha [ABA15].

We reuse the same logic $\mathcal{L}$ as presented in the previous section on classical planning. However, in a Hierarchical *Task* Network, we can distinguish two kinds of tasks: *primitive tasks* (also called actions, identical to the ones used in the classical planning setting) and *abstract tasks* (also called non-primitive tasks or compound tasks).

Example 2.4 *An intuitive hierarchical version of the* Logistics *domain example.*
Before detailing the formalism, let us give an intuitive idea of what an HTN domain could look like, with a hierarchical view presented in Figure 2.2, reusing the actions defined in the classical planning example as primitive tasks.

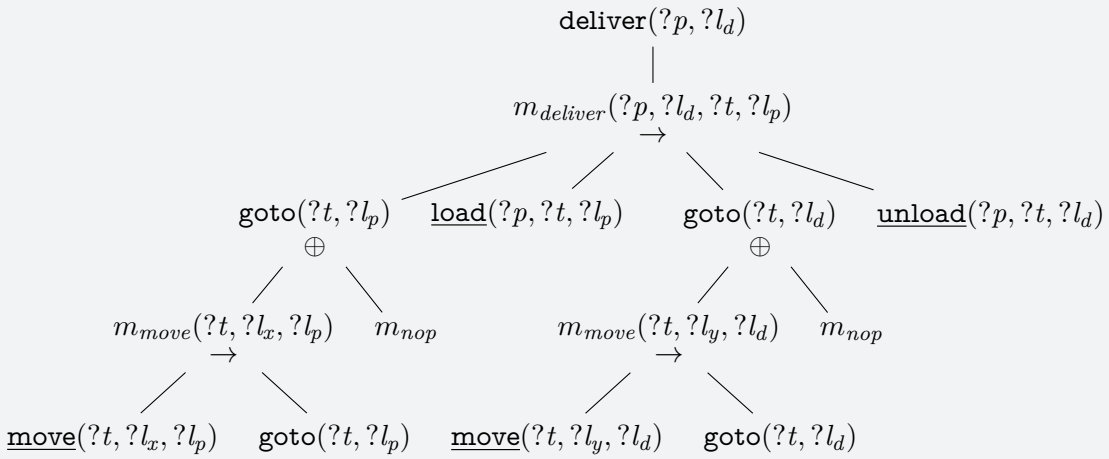


Figure 2.2: An intuitive view of an HTN hierarchy. Primitives are typeset as action$(?x, \ldots, ?y)$.

At the top level of this hierarchy is an abstract deliver task, representing the activity that we want to achieve. This task is decomposed into lower level tasks with the single method $m_{deliver}$, where all its subtasks are executed in left-to-right order.
However, we note that this decomposition relies on two instantiations of the goto task goto is not an action, but rather another abstract task, representing the activity of going from the truck's current location to another location, possibly with intermediate steps. Because it is abstract, it needs to be refined further. This is achieved with two methods among which the planner will have to choose: one to actually move and the other to end the decomposition of the goto task.

Let us start with the definition of the primitive and abstract tasks:

Definition 2.10 (Primitive Task)**.** A *primitive task* is an action, as defined in classical planning.

Definition 2.11 (Abstract Task)**.** An *abstract task* t is similar to an action, but is simply defined with head(t). Contrary to actions, it does not induce state transitions but rather is used to reference a mapping to one or more task networks which can be used to refine t. This mapping is given by a given set M of methods.

Similar to the actions, all the variables in pre(t) and post(t) must be parameters of t.

We can now define the structure of an HTN, called a *task network*, which represents partially ordered multi-sets of tasks.

Definition 2.12 (Task Network)**.** A task network tn over a set of parameterized task names X is a tuple $(I, \prec, \alpha)$ such that:

- I is a (possibly empty) set of *task identifiers*.
- $\prec$ is a strict partial order over I.
- $\alpha : I \to X$ maps identifiers to task heads.

It is possible to restrict $\prec$ to be a *total* order. To distinguish both versions of the formalism, we will use the names *totally ordered* (TO) HTNs and *partially ordered* HTNs.

Task identifiers are arbitrary symbols that serve as placeholders for the actual task they represent. Identifiers are necessary because a task can occur multiple times within the same network.

A task network is primitive if all the task identifiers map to primitive tasks. It is *ground* if all its parameters are bound to constants in C, and *lifted* otherwise. If a task network tn is ground and primitive, tn is executable in a state s_0 if there is a linearization of its tasks that is executable s. This is similar to saying that tn can be converted to an executable plan in the classical planning sense.

Figure 2.3 illustrates how a task network could be represented graphically.

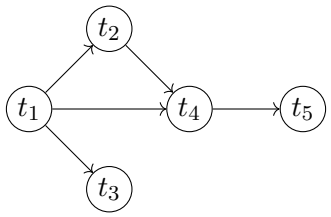


Figure 2.3: A task network example. Here, all the t_i are task identifiers, and the directed edges represent the ordering of the task network elements. This ordering can be written as: $\{(t_1 \prec t_2), (t_1 \prec t_3), (t_1 \prec t_4), (t_2 \prec t_4), (t_4 \prec t_5)\}$.

Definition 2.13 (Method)**.** A method $m \in M$ is a tuple $(c, tn, \text{pre}(m))$, where c, called the *method head*, is an abstract task's head and tn is a task network called the method's subtasks.

- The parameters of m is a set of parameter symbols (variables or constants) associated with parameters of c or of the tasks in tn. A single parameter of m may be associated with several parameters of c or its subtasks in order to enforce their unification. Additionally, a method is ground if and only if all its parameters are all constants.
- pre(m) is the (possibly empty) set of pre-conditions of the method. It is defined similarly to the pre-conditions of an action: they determine when a decomposition method can be applied[1], and all the variables in $pre(m)$ must be parameters of the method.

[1]Checking the validity of a method's preconditions is not a trivial task [Höl+20], especially in the context of partially ordered domains. In this work, we make the same assumption as in the HDDL language specification [Höl+20] and assume they are compiled as a primitive task ordered before the other method subtasks.

For example, an always applicable method to decompose a task $\mathtt{t}(?x, ?y)$ into a task network tn would be written $m = (\mathtt{t}(?x, ?y), tn, \emptyset)$.

Analogously to a classical planning domain, we can now define a hierarchical planing domain:

Definition 2.14 (Hierarchical Planning Domain)**.** A hierarchical planning domain $\mathcal{H}$ is a tuple $(\mathcal{L}, T_C, T_P, M)$ defined as follows:

- $\mathcal{L}$ is the underlying predicate logic.
- T_P and T_C are the sets of primitive and compound tasks.
- M is a set of decomposition methods with heads from T_C and task networks over the parameterized names $T_C \cup T_P$.

Definition 2.15 (Hierarchical Planning Problem)**.** A hierarchical planning problem $\mathcal{P}_h$ is a tuple $(\mathcal{H}, s_0, tn_I, g)$ where:

- $\mathcal{H}$ is a hierarchical planning domain.
- $s_0 \in S$ is the initial state.
- tn_I is the initial task network. It represents the top-level activities to be achieved.
- g is a possibly empty set of goal atoms that must hold in the final state.

While HTN planning is not about achieving a state based goal, adding it allows specifying problems closer to the PDDL specification. It is used to ensure that all the desired state features hold together, i.e. that none was undone by another execution. This is especially useful in cases where the hierarchy is used to model *advice* and not dynamics of the environment [Beh+19; McD00]. Furthermore, it can be left empty if unused.

Example 2.5 *A hierarchical planning domain and problem for our* LOGISTICS*-like domain.*

With these definitions, we can define the same LOGISTICS-like domain and problem that we described in the previous section in a hierarchical way. Please note that in this example, task networks will be presented as graphs using task heads instead of identifiers to give a more human-readable description. Nodes in the graph will allow distinguishing task instantiations.

To define our hierarchical domain $\mathcal{H} = (\mathcal{L}, T_A, T_P, M)$, we use the same logic $\mathcal{L}$ as in the classical planning case, and we introduce two abstract tasks: `deliver` and `goto`. `deliver` represents our high-level delivery activity, while `goto` represents the act of getting from one location to another while possibly requiring intermediate steps. We can therefore define T_C as:

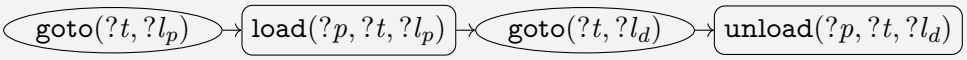
$$T_C = \left\{ \begin{array}{l} \mathtt{deliver}(?p : \mathit{package}, ?l : \mathit{location}) \\ \mathtt{goto}(?t : \mathit{truck}, ?l : \mathit{location}) \end{array} \right\}$$

Because we use the same primitive tasks as in the classical planning case, the set T_P is the same as the set of actions in this previous case:

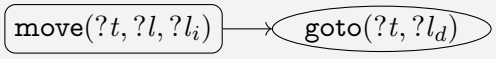
$$T_P = \left\{ \begin{array}{rl} \texttt{move} = & \left(\begin{array}{l} \text{head} : \texttt{move}(?t : \textit{truck}, ?l_1 : \textit{location}, ?l_2 : \textit{location}) \\ \text{pre} : \{\texttt{at}(?t, ?l_2), \texttt{connected}(?l_1, ?l_2)\} \\ \text{eff} : \{\texttt{at}(?t, ?l_2), \neg\texttt{at}(?t, ?l_1)\} \end{array} \right) \\ \texttt{load} = & \left(\begin{array}{l} \text{head} : \texttt{load}(?p : \textit{package}, ?t : \textit{truck}, ?l : \textit{location}) \\ \text{pre} : \{\texttt{at}(?t, ?l), \texttt{at}(?p, ?l), \texttt{empty}(?t)\} \\ \text{eff} : \{\texttt{in}(?p, ?t), \neg\texttt{at}(?p, ?l)\neg\texttt{empty}(?t)\} \end{array} \right) \\ \texttt{unload} = & \left(\begin{array}{l} \text{head} : \texttt{unload}(?p : \textit{package}, ?t : \textit{truck}, ?l : \textit{location}) \\ \text{pre} : \{\texttt{at}(?t, ?l), \texttt{in}(?p, ?t)\} \\ \text{eff} : \{\texttt{at}(?p, ?l), \texttt{empty}(?t), \neg\texttt{in}(?p, ?t)\} \end{array} \right) \end{array} \right\}$$

We define the set of methods M as in the equation below. The task networks in the methods in M can also be represented graphically as in Figure 2.5.

$$M = \left\{ \begin{array}{rl} m_{\texttt{deliver}} = & \left(\begin{array}{l} c_{\texttt{deliver}} = \texttt{deliver}(?p : \textit{package}, ?l_d : \textit{location}), \\ tn_{\texttt{deliver}} = \left\{ \begin{array}{l} (\texttt{goto}(?t, ?l_p)_1 \prec \texttt{load}(?p, ?t, ?l_p)_2) \\ (\texttt{load}(?p, ?t, ?l_p)_2 \prec \texttt{goto}(?t, ?l_d)_3) \\ (\texttt{goto}(?t, ?l_d)_3 \prec \texttt{unload}(?p, ?t, ?l_d)_4) \end{array} \right\} \end{array} \right) \\ m^{move}_{\texttt{goto}} = & \left(\begin{array}{l} c^{move}_{\texttt{goto}} = \texttt{goto}(?t : \textit{truck}, ?l_d : \textit{location}), \\ tn^{move}_{\texttt{goto}} = \{(\texttt{move}(?t, ?l, ?l_i)_5 \prec \texttt{goto}(?t, ?l_d)_6)\} \end{array} \right) \\ m^{nop}_{\texttt{goto}} = & \left(\begin{array}{l} c^{nop}_{\texttt{goto}} = \texttt{goto}(?t : \textit{truck}, \texttt{l}_d : \textit{location}), \\ tn^{nop}_{\texttt{goto}} = \emptyset \end{array} \right) \end{array} \right\}$$



(a) Task network for the method $m_{deliver}$.



(b) Task network for the method $m^{move}_{\texttt{goto}}$.

Figure 2.5: A visual representation of the task networks of the methods in M.

The planning problem $\mathcal{P}_h = (\mathcal{H}, s_0, tn_I, g)$ uses the same s_0 and g as in the classical planning example presented earlier, in order to solve an equivalent problem:

$$s_0 = \left\{\begin{array}{ll} \texttt{connected}(l_0, l_1) & \texttt{connected}(l_1, l_0) \\ \texttt{connected}(l_0, l_3) & \texttt{connected}(l_3, l_0) \\ \texttt{connected}(l_1, l_4) & \texttt{connected}(l_4, l_1) \\ \texttt{connected}(l_2, l_4) & \texttt{connected}(l_4, l_2) \\ \texttt{connected}(l_3, l_4) & \texttt{connected}(l_4, l_3) \\ & \\ \texttt{at}(t, l_0) & \texttt{empty}(t) \\ \texttt{at}(p_1, l_3) & \texttt{at}(p_2, l_4) \end{array}\right\}$$

$$g = \texttt{at}(p_1, l_1) \wedge \texttt{at}(p_2, l_2)$$

The initial task network tn_I only contains two instantiations of the `deliver` task, and is presented as a graph in Figure 2.6.

Figure 2.6: Initial task network tn_I for the delivery planning problem.

Solutions to an HTN are typically obtained by iteratively substituting abstract tasks through method decomposition until we obtain an executable primitive task network. Such solutions can be obtained from the problem's initial task network, through grounding, application of methods and addition of ordering constraints.

More formally, methods are applied through *decomposition*, which transforms one task network into another: a task decomposed through a method is removed from the network and replaced with its subtasks, inheriting the ordering relations that held for the abstract task.

Definition 2.16 (Decomposition)**.** Let $m = (c, (I_m, \prec_m, \alpha_m))$ be a ground method, and $tn_1 = (I_1, \prec_1, \alpha_1)$ a task network such that $I_m \cap I_1 = \emptyset$ (which can be achieved through renaming). Then, m decomposes a task identifier $i \in I_1$ into a task network $tn_2 = (I_2, \prec_2, \alpha_2)$ if and only if $\alpha_1(i) = c$ and:

$$I_2 = (I_1 \setminus \{i\}) \cup I_m$$

$$\prec_2 = \left(\begin{array}{l} \prec_1 \cup \prec_m \\ \cup \{(i_1, i_m) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \\ \cup \{(i_m, i_1) \in I_m \times I_1 \mid (i, i_1) \in \prec_1\} \end{array}\right) \setminus \left\{(i', i'') \in I_1^2 \mid i' = i \vee i'' = i\right\}$$

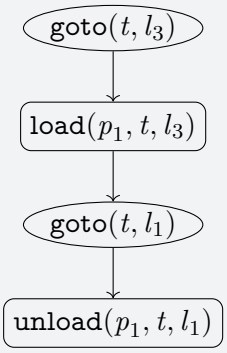
$$\alpha_2 = (\alpha_1 \cup \alpha_m) \setminus \{(i, c)\}$$

Definition 2.17 (Solution)**.** Let $\mathcal{P} = (\mathcal{H}, s_0, tn_I, g)$ be a planning problem with $\mathcal{H} = (\mathcal{L}, T_P, T_C, M)$ and $tn_S = (I_S, \prec_S, \alpha_S)$. tn_S is a solution to an HTN planning problem if and only if the following conditions hold:

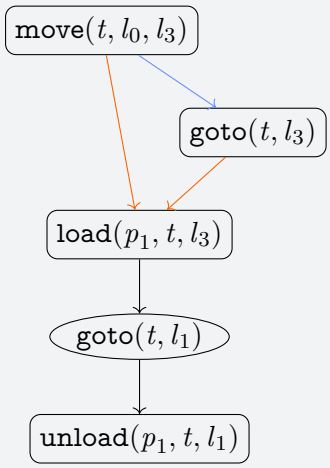
- There exists a sequence of decompositions from tn_I resulting in a task network $tn =$ (I, $\prec$, α) such that $I = I_S$, $\prec \subseteq \prec_S$, and $\alpha = \alpha_S$.
- tn_S is primitive, ground and $\prec_S$ is a strict total order[1].
- The plan π_S corresponding to tn_S is applicable and $g \subset \gamma(s_0, \pi_S)$

[1]Some formalisms [ABA15] only require that such ordering exist for tn_S to be a solution.

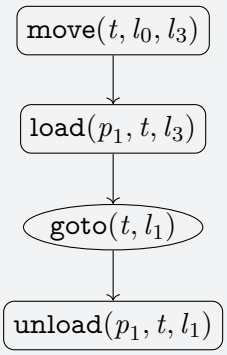
Example 2.6 *Applying decomposition rules to solve our example.*
Let us now apply decomposition rules to obtain a solution to the planning problem presented in the previous example. In this example, arrows represent the precedence constraints, as in Figure 2.3 shown earlier. While both decompositions cannot be completely independent of one another, as will be shown later, let us first focus on the decomposition of $\mathsf{deliver}(p_1, l_1)$ to have a clearer illustration.



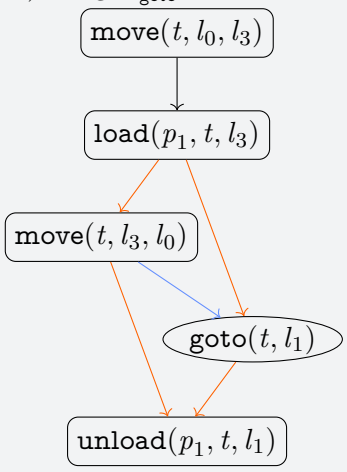
(a) Task network after decomposing $\mathsf{deliver}(p_1, l_1)$ using m_{deliver}.



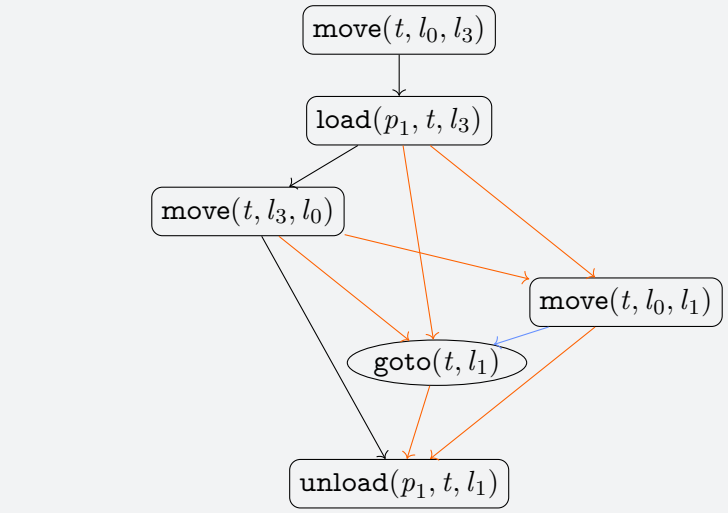
(b) Task network after decomposing $\mathsf{goto}(t, l_3)$ using m^{move}_{goto}.



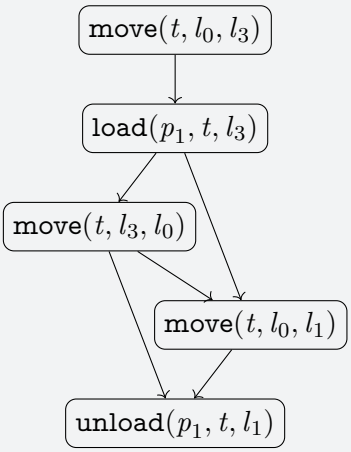
(c) Task network after decomposing $\mathsf{goto}(t, l_3)$ using m^{nop}_{goto}.



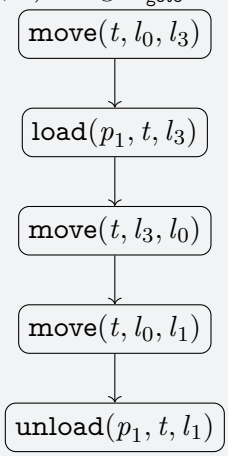
(d) Task network after decomposing $\mathsf{goto}(t, l_1)$ using m^{move}_{goto}.



(e) Task network after decomposing $\texttt{goto}(t, l_1)$ using $m^{move}_{\texttt{goto}}$.



(f) Task network after decomposing $\texttt{goto}(t, l_1)$ using $m^{nop}_{\texttt{goto}}$.



(g) Task network after removing redundant edges.

Figure 2.8: A possible decomposition for the task $\texttt{deliver}(p_1, l_1)$, starting in state s_0. Here, for each decomposition of a task with identifier i:

- Blue edges are the ones that are part of the method task network ($\prec_m$ in Def. 2.16).
- Orange edges are the ones newly created to satisfy the ordering of i with regard to the rest of the network.

If one were to now try and decompose the second task $\texttt{deliver}(p_2, l_2)$ starting from s_0, one possible decomposition of the global initial task network could be the one presented in Figure 2.9. However, this decomposition would not lead to a solution to HTN planning problem: while it is primitive, it cannot be totally ordered such that the resulting linearization is executable and achieves the goal.

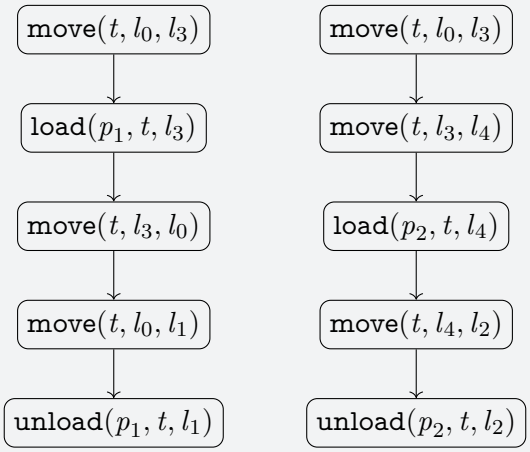


Figure 2.9: A possible decomposition for the initial task network which is not a solution to the HTN planning problem.

However, the decomposition presented in Figure 2.10 (solid lines only), can be turned into a solution, as it can trivially be totally ordered by adding the precedence constraint represented by the dashed line, resulting in the solution plan presented in the earlier classical planning example.

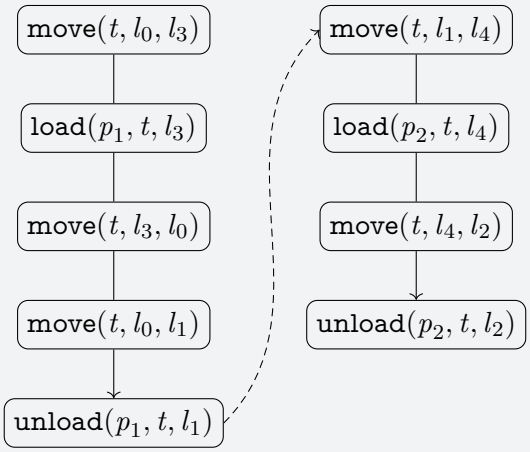


Figure 2.10: A possible decomposition for the initial task network which is a solution to the HTN planning problem.

2.3 A Learner for Parameterized Hierarchical Task Networks from Demonstrations

Let us now present an overview of the problem tackled in this thesis: the learning of HTN domains from non-hierarchical demonstration traces given by a tutor. While the next chapters will focus on specific parts of this system, this overview will help give the reader an understanding of the interaction of the different parts of our system.

2.3.1 Learning Problem

At a high level, we wish to learn HTN domains from a set of demonstrations such that the learned domains can be efficiently used by an off-the-shelf HTN planner. We consider that the learned hierarchical structure models advice and not dynamics of the environment. In a nutshell, we consider that the primitive tasks pre-conditions and effects will accurately reflect dynamics in the planning world, while the hierarchical structure should provide guidance to the planner. This specifically means that every planning problem associated with a learned hierarchical planning domain $\mathcal{H}_L$ must specify a goal state.

2.3.1.1 Inputs

We consider as learning problem a tuple $L = (T_P, T_I, M_I, D, \text{post})$.

- T_P is a set of primitive tasks, as defined for hierarchical task networks, representing the primitives of the agent. Each action in T_P is completely defined. Its head (name and parameters) will be used by our learner, while the preconditions and effects are required to correctly consider the dynamics of our planning environment.
- T_I is a set of abstract tasks, representing the initial vocabulary of high level tasks which we want to learn to decompose.
- M_I is a potentially empty initial set of methods. It is used to consider pre-existing knowledge as a starting point for the learner.
- post is a function that associates with every task $t \in T_I$ a (possibly empty) goal g. For a goal $g = \text{post}(t)$, the parameters of the predicates in g are included in the parameters of t. We call this goal the *post-conditions* of t.
- D is a set of demonstrations. A demonstration is an alternating sequence of states and tasks, similar to a solution trace, with the difference that the tasks can be either primitive or abstract. Each demonstration is associated with an instantiation of a task $t \in T_I$ and ends in a state s_f where t has been successfully achieved (i.e. $\text{post}(t) \subseteq s_f$).

The demonstrations are considered optimal and maximally abstract with regard to the initial task vocabulary and a chosen metric. Therefore, there exists no other demonstration which has a strictly lower cost. In a case where actions are uniform in cost, one may naturally consider the total number of primitive actions required to achieve the demonstrated task as the optimality metric.

Definition 2.18 (Maximally Abstract Demonstration)**.** A *maximally abstract demonstration* d with regard to a set of abstract tasks T_I is such that no task in $t \in T_I$ can be used to abstract a subsequence of d.

Figure 2.11 shows an example of two different maximally abstract demonstration sets on our simple LOGISTICS-like domain for demonstrating the `deliver` task. In both cases, the set of primitive actions T_P is the one presented in Figure 2.11a, and the delivery task is always decomposed in the same manner: the package is recovered from its original location and then dropped off at the target location.

In the first case (Figure 2.11b), the behaviour is demonstrated using only the primitive actions, as it is the maximal level of abstraction that can be achieved using T_P and T_I^1. In the second case, the demonstrations make use of the known abstract tasks `get` and `dropoff` to

provide a maximally abstract demonstration for the deliver task. Information on how to achieve these intermediate tasks is provided as two other separate demonstrations.

$$T_P = \left\{ \begin{array}{l} \texttt{move}(?t : \textit{truck}, ?l_1 : \textit{location}, ?l_2 : \textit{location}) \\ \texttt{load}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{unload}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \end{array} \right\}$$

(a) Initial set of primitive actions.

$$T_I^1 = \left\{ \texttt{deliver}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \right\}$$

$$D_1 = \left\{ \begin{array}{l} \texttt{deliver}(t_1, l_3, p_1) : \langle s_0 \rightarrow \texttt{move}(t_1, l_1, l_2) \rightarrow s_1 \rightarrow \texttt{load}(t_1, l_2, p_1) \rightarrow s_2 \\ \qquad\qquad \hookrightarrow \texttt{move}(t_1, l_2, l_3) \rightarrow s_3 \rightarrow \texttt{unload}(t_1, l_3, p_1) \rightarrow s_4 \rangle \end{array} \right\}$$

(b) A first set of initial vocabulary with an associated set of demonstrations.

$$T_I^2 = \left\{ \begin{array}{l} \texttt{get}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{dropoff}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{deliver}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{l} \texttt{deliver}(t_1, l_3, p_1) : \langle s_0 \rightarrow \texttt{get}(t_1, l_2, p_1) \rightarrow s_2 \rightarrow \texttt{dropoff}(t_1, l_3, p_1) \rightarrow s_4 \rangle \\ \quad \texttt{get}(t_1, l_2, p_1) : \langle s_0 \rightarrow \texttt{move}(t_1, l_1, l_2) \rightarrow s_1 \rightarrow \texttt{load}(t_1, l_2, p_1) \rightarrow s_2 \rangle \\ \texttt{dropoff}(t_1, l_2, p_1) : \langle s_2 \rightarrow \texttt{move}(t_1, l_2, l_3) \rightarrow s_3 \rightarrow \texttt{unload}(t_1, l_3, p_1) \rightarrow s_4 \rangle \end{array} \right\}$$

(c) A second set of initial vocabulary with an associated set of demonstrations. Note how the demonstrations make use of the additional abstraction levels to remain maximally abstract.

Figure 2.11: Example of initial task vocabulary and corresponding maximally abstract demonstration sets for a deliver task. All the demonstration sets correspond to the same sequence of primitive actions but are abstracted at different levels. Here, the considered optimality metric is the number of actions in the plan. Details of states omitted for clarity.

2.3.1.2 Goals

The primary objective of our learner is to be able to produce a hierarchical planning domain $\mathcal{H}$ that is both complete and sound. To formally define these properties, let us first define $\text{Decs}(\mathcal{H}, tn)$ the set of all the ground, primitive and totally-ordered (TO) task networks that tn can be decomposed into with $\mathcal{H}$. We write $\text{Decs}_s(\mathcal{H}, tn)$ to denote the same set restricted to a decomposition starting in state s.

Because a ground primitive TO task network can trivially be converted to a classical plan, for a given $n \in \text{Decs}(\mathcal{H}, tn)$, with π_n the corresponding plan, we will write $\pi_n \in \text{Decs}(\mathcal{H}, tn)$ to denote the fact that π_n is a plan that can be obtained by decomposition. For a task t, we also write $\text{Decs}(\mathcal{H}, t)$ as a shorthand for $\text{Decs}(\mathcal{H}, tn)$ when tn only contains t.

Furthermore, for an instantiation of a task $t \in T_I$, we write $\mathcal{P}_t = (\mathcal{H}, s_0, t, \text{post}(t))$ the planning problem corresponding to decomposing t using $\mathcal{H}$ in state s_0.

Definition 2.19 (Completeness)**.** We say that a hierarchical planning domain $\mathcal{H}$ is *complete* if every task can be decomposed in any state:

$$\forall t \in T_I, \quad \text{Decs}(\mathcal{H}, t) \neq \emptyset$$

Definition 2.20 (Soundness)**.** We say that a hierarchical planning domain $\mathcal{H}$ is *sound* if every decomposition of a task results in a valid plan that achieves its postconditions:

$$\forall t \in T_I, \forall s_0 \in S, \mathcal{P}_t = (\mathcal{H}, s_0, t, \text{post}(t))$$
$$\forall \pi \in \text{Decs}_{s_0}(\mathcal{H}, t), \pi \text{ is a valid solution to } \mathcal{P}_t$$

To learn such a model, considering a learning problem $L = (T_P, T_I, M_I, D)$ as defined previously, we want to produce a set of compound tasks T_L and a set of methods M_L such that $\mathcal{H} = (\mathcal{L}, T_P, T_I \cup T_L, M_I \cup M_L)$, such that $\mathcal{H}$ satisfies these properties.

Note however that the soundness of an HTN must be considered in combination with an automated planner that exploits it: the hierarchical structure itself may allow generating unsound plans, but the planner should be able to filter them out using the actions pre-conditions. Similarly, a complete model may be so permissive that it does not provide guidance to the planner, and therefore not be of interest. We can therefore say that the quality of a planning domain $\mathcal{H}$ is tightly coupled with the ability of an automated planner to exploit it to quickly derive solution plans.

In particular, for any given planner we are aiming at maximizing the efficiency of the planner for solving a planning problem given $\mathcal{H}$, which is typically measured as the runtime of the planner. This leads to defining the *coverage* of domain as the ratio of solved problems by a given planner under computational limits. Furthermore, we would like the plans generated with our learned domains to approach the underlying optimality metric in the demonstrations, which leads to the definition of the *quality gap* as the ratio between the cost of the plan generated using our domain and one generated using an optimal oracle.

2.3.2 The HTN Domains Considered in this Thesis

First, let us note that we restrict ourselves to learning *totally-ordered* (TO) HTNs decomposition methods. This restriction will allow us to learn the structure and the parameters of an HTN domain in two separate steps. While these two steps will be presented in more details later in this chapter, let us simply say that the structure-learning component ignores all parameters. This would then make it difficult to separate interleaved primitive tasks in our demonstrations sequences that are actually part of different (intermediate) high-level behaviours. This issue is not as prevalent in the TO case, as we can assume that such interleaving does not often happen.

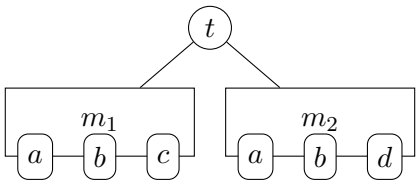
Note that whenever the context is clear enough, in the remainder of this document, we will call a hierarchical planning domain simply an HTN. Because subtasks are always sequentially ordered in this case, we can provide a simpler graphical representation for an HTN, as presented in Figure 2.12, in which the order of a method's subtasks is implicitly represented through their left to right ordering.

Let us now present an initial intuition about the shape of the task networks that could be learned and the implications for the learning process. Let us assume that we have a planning domain with a set of four primitive actions $T_P = \{a, b, c, d\}$ and single demonstrated compound task t. Figure 2.13 presents several possible networks (figures 2.13b-2.13e) that could be generated based on two example sequences (figure 2.13a). These four domains are just a handful of examples among the many possible ones that could be generated.

The first domain (2.13b) allows the choice of any of the four primitive actions $\{a, b, c, d\}$, each placed in a specific method. This domain relies on a recursive call to t to repropose the same choice until the task's post-conditions are achieved. While this domain allows building any sequence of actions (it is therefore intrinsically *complete*) it does not help the agent towards

$$\mathcal{H} = \left(\begin{array}{l} T_P = \{a, b, c, d\} \\ T_C = \{t\} \\ M = \left\{ \begin{array}{l} m_1 = \left(\begin{array}{l} c_1 = t \\ tn_1 = \{a \prec b \prec c\} \end{array} \right) \\ m_2 = \left(\begin{array}{l} c_2 = t \\ tn_2 = \{a \prec b \prec d\} \end{array} \right) \end{array} \right\} \end{array} \right)$$

(a) HTN definition.

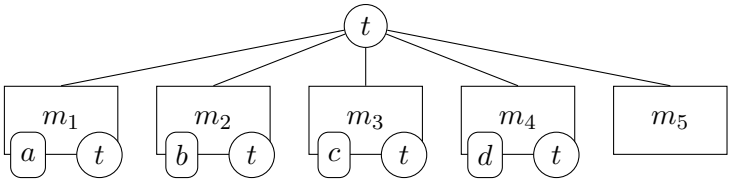


(b) HTN graphical representation.

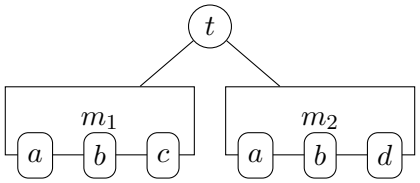
Figure 2.12: A simplified graphical representation for an HTN. Parameters, pre-conditions and effects omitted for clarity.

$$t \to \langle a, b, c \rangle \qquad t \to \langle a, b, d \rangle$$

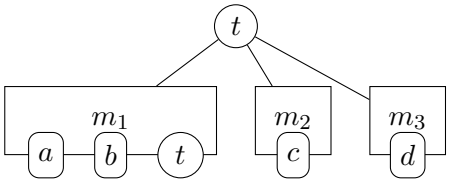
(a) Available demonstrations, showing that t was once achieved with the $\langle a, b, c \rangle$ action sequence and once with the $\langle a, b, d \rangle$ action sequence. Intermediate states omitted for brevity.



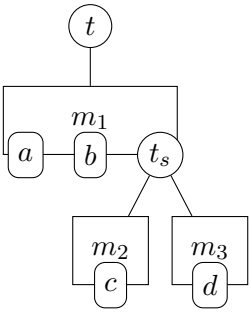
(b) Generic domain where the planner might pick any of the primitive actions and rely on the recursive call to t to continue if needed.



(c) Domain where each demonstration is fully encoded into a dedicated method.



(d) Intermediate domain the common $\langle a, b \rangle$ sequence is grouped. It relies on the recursive call to t in m_1 to produce a full sequence.



(e) Domain where the $\langle a, b \rangle$ sequence is shared, requiring a new abstract task t_s

Figure 2.13: Illustration of the possible structures of the learned domain for a simple learning task with two demonstration of how to perform a task t. Note that for conciseness the parameters and pre-conditions of the task and methods are omitted.

a meaningful sequence based on demonstrations.

The second domain (2.13c) takes the opposite approach and records each known trace into a method. This domain is obviously strongly tied to the demonstration set: it covers only planning problems where the plan structure matches exactly that of the demonstrations, and therefore does not exhibit any generalization capabilities.

In between these two extremes, we have the domains (2.13d) and (2.13e) that present different options to abstract common subsequences. The former encodes the repeated $\langle a, b \rangle$ sequence in a single method and relies on the recursive call to complete the sequence. The latter delays the choice between c and d to after the execution of a and b, using a new abstract task t_s.

Denoting as Θ the set of possible domains, the objective of a learning system is to find, or at least approach, the optimal domain $\mathcal{H}^* \in \Theta$

$$\mathcal{H}^* = \arg\min_{\mathcal{H} \in \Theta} cost(\mathcal{H}) \tag{2.1}$$

where $cost(\mathcal{H})$ is a function that measures the cost of a particular domain and should typically account for the size of the domain as well as its capacity to solve both demonstrated and unseen problems.

2.4 Related Work

Let us now review the works relating to our proposed approach. Before focusing on learning hierarchical models of tasks, we will start with methods for learning skill *models* in the form of classical planning operators, only then gradually moving towards hierarchical models, in order to give the reader an overview of the landscape of learning for automated planning. While we will focus on hierarchical *planning* domains, we will extend our review to encompass other approaches with a hierarchical structure, such as grammars and hierarchical policies.

We will also present work done on Programming by Demonstration (PbD), due to its inclusion in the larger field of Learning from Demonstration (LfD), and work on process mining, given our focus on learning higher level behaviour from sequences of basic primitives. Finally, we will also present the work done on generalized planning, highlighting the intersection between this research domain and that of learning policies, hierarchical planning knowledge and programming by demonstration.

2.4.1 Learning Action Models

In a robotics system, skills can be learned by the agent. One of the common approaches to combine these skills to achieve a goal is to use them with a planning system, abstracting them as planning actions. However, this requires that *models* of these actions are available, so that the planner can reason on their pre-conditions and effects. Let us therefore focus on the systems that learn such models, first focusing on ones that conform closely to the definition of actions in classical planning (deterministic effects and fully observable environment), before extending it to broader action types. We present a summary of these model-learning approaches in Table 2.1.

EXPO [Gil94] is one of the first action model learning system, trying to complete the domain knowledge available to a planner, adding missing pre-conditions or effects to existing planning operators. It does so by monitoring the real world during plan execution, ensuring that expected outcomes match the state of the real world.

The OBSERVER system [Wan95], on the other hand, leverages expert-generated solution traces and *practice problems* available on a simulator to learn its operators. The global idea is

Approach	Input	Output	Obs.	Noise	ND
EXPO [Gil94]	Incomplete action models Action execution results	Pre-conditions and effects	Full	✗	✗
OBSERVER [Wan95]	Expert solution traces Simulator for executing actions	Pre-conditions and conditional effects	Full	✗	✗
TRAIL [Ben95]	Simulator for executing actions Oracle for action choice requests	Pre-conditions and probabilistic effects	Full[1]	✓	✓
ARMS [YWJ07]	Plans with initial and goal states	Pre-conditions and effects	Partial	✗	✗
FAMA [AJO19]	Partial plans with initial and goal states	Pre-conditions and effects	Partial	✗	✗
[Mou+12]	Partial and noisy action execution results	Pre-conditions and effects	Partial	✓	✗
[OC96][2]	Simulator for executing actions	Pre-conditions and probabilistic effects	Full	✗	✓
[PZK07]	Noisy action execution results	Pre-conditions and probabilistic effects	Full	✓	✓
[MAT15]	Simulator for executing actions Oracle for action choice requests	Pre-conditions and probabilistic effects	Full	✓	✓
SAM [JS22]	Solution traces	Conservative pre-conditions and effects	Full	✗	✓
[GJ20][3]	Partial solution traces	Pre-conditions, effects and temporal annotations	Partial	✗	✗
[BG20]	Transition graphs between black box states, with edges labelled with action names	Pre-conditions and effects	Partial[4]	✓[4]	✗

Table 2.1: Summary of the action model learning approaches. Obs. and ND stand *Observability* and *Non-Determinism*, respectively.

[1] Partial observation of the state features may be considered as noise in this approach.
[2] Limited to 0-ary actions and predicates.
[3] Temporal planning framework.
[4] In extension [Rod+21]

to find the difference between pre- and post-states in the expert trajectories, giving an initial set of pre-conditions and effects, which are then refined through the automated generation of new (non-expert) trajectories using the simulator.

The learner integrated in the TRAIL system [Ben95] attempts to learn pre-conditions for slightly different action types, called *teleo-operators*, and focuses on *reactive* models, rather than planning ones. However, these teleo-operators are similar enough to planning operators, for model-learning purposes, to be included here. This learner takes an Inductive Logic Programming (ILP)-based approach in order to handle noise in the demonstration data, which also allows the learner to consider negative examples (i.e. examples of failed actions) to learn more efficiently. The example used as inputs are either sequences of states and actions similar to solution traces, or simple tuples of the form (s_i, a_i, s_{i+1}), leveraging both random exploration and tutor guidance. These examples are then converted into a set of background facts, representing what holds in a given state, and a set of foreground facts, which represents positive and negative examples. It learns clauses that can be used to determine whether a conjunction of literals is a pre-condition or an effect for an operator.

The ARMS system [YWJ07] relaxes the full observability assumption of the previous approaches, allowing incomplete pre- and post-states in the given solution traces. It uses a MAX-SAT-based approach, adding weighted constraints to encode that a literal precedes or follows a given action. It leverages the fact that literals relevant to the pre-conditions (respectively effects) of an action are likely to appear in a pre-state (respectively a post-state) more often than irrelevant ones. A larger weight will be attributed to frequently encountered constraints, and the MAX-SAT solver will therefore favour them when solving the constraint system. The set of satisfied constraints at the end of the procedure can then be used to extract the pre-conditions and effects: if a precedence (resp. succession) constraint is satisfied, then the literal is a pre-condition (resp. effect).

These observability constraints are further relaxed in the FAMA [AJO19] system, which allows partial observability not only in the states of the solution traces, but also allows missing actions. However, no noise is tolerated in the traces. This approach uses planning to build the planning domain, compiling the learning task into a planning one. This new planning task replaces the original actions set with three new ones:

- A set to *program* actions, adding preconditions or effects to a given original action.
- One to apply a programmed action in the world, considering the programmed pre-conditions and effects.
- A set of validating actions, to ensure that the programmed actions are able to produce the example traces.

This planning problem is then solved, and the action model is extracted from the solution, using the programmed actions.

The work by Mourao et al. [Mou+12] handles both noise and partial observability through a two-part approach: first, an implicit action model is learnt, and then explicit rules are extracted from it. The implicit models are learned using classifiers which learn whether an action a changes a literal when applied in a given context, and are used to learn independent rules for each effect of a. These independent rules are then combined to end up with a single rule for each action.

Extending the scope from deterministic planning domains to *non-deterministic* ones, several other approaches can be cited, such as the work by Oates and Cohen [OC96] who propose an algorithm to learn probabilistic action pre-conditions and effects models. It relies on random

exploration of the state space, detecting dependencies between pre- and post-states literals. However, the presented domain does not have any variables (only 0-ary predicates and actions), which, coupled with the random exploration procedure, questions whether this approach can be extended to more complex domains.

The approach by Pasula, Zettlemoyer and Kaelbling [PZK07] learns much richer probabilistic models, allowing the use of parameters in the operators' descriptions. This system is robust to noise but requires a fully observable environment. The algorithm uses a greedy local search algorithm to find the action model that best fits the provided examples, using a penalty on model complexity to avoid overfitting. It is able to learn new predicates, leveraging the use of *deictic references* to limit the space of possible models. Deictic references are variables that are uniquely defined in terms of predicates parameterized with other variables.

To improve the sample efficiency of model learning algorithms, some approaches leverage active tutor requests to reduce the search space. One such approach is the one by Martínez, Alenyà and Torras [MAT15]. It uses the same model representation as the previously presented work [PZK07], but integrates it within the Relational Reinforcement Learning (RRL) approach REX [LTK12]. The tutor's burden is limited through the use of autonomous exploration and the use of an *excuses* [Goe+10] framework to determine which part of the model is most likely in need of correction.

The SAM algorithm [JLS21] focuses on the *safety* properties of the learned models in a deterministic case. It does so by constructing the action model through an iterative restriction of the pre-conditions and an iterative construction of the effects, which ensures that the learned action model is more restrictive than the real one. This algorithm was later extended to deal with non-deterministic domains [JS22], bounding the probability of different effects for any action.

Model learning is not limited to classical planning, as the approach by Garrido and Jimenez [GJ20] uses Constraint Programming (CP) to learn temporal action models, allowing to use both (partial) observations and additional expert knowledge easily.

While all the approaches presented up to this point focus on learning action models from *symbolic* data, research has been done on trying to learn these models without requiring to first extract this symbolic representation from the states. The approach by Bonet and Geffner [BG20] relies on graphs representing the structure of the state space (i.e. the transitions from one state to another using an action), without any information on the content of the states, using a SAT approach. This approach is able to learn models in several simple International Planning Competition (IPC) domains, and was later improved [Rod+21] using Answer Set Programming (ASP) to obtain better models at a lower computational cost.

2.4.2 Learning Hierarchical Models

Moving away from classical planning action models, we now focus on hierarchical models. This section will not focus solely on planning domains, but will encompass other classes of hierarchical models as well.

2.4.2.1 Macro Operators

While not hierarchical models per se, macro operators can be seen as sequences of planning operators abstracted as single higher level operator in order to provide shortcuts through the search space to speed up planning. Therefore, they also leverage the concept of *abstraction* that

we are looking for in our compound tasks. For this reason, we will present a brief overview of the work done on learning such macro operators.

We can find early mentions of this idea in the work of Fikes, Hart and Nilsson [FHN72], motivated by the idea of reusing plans generated using STRIPS in the context of controlling a mobile robot. As this system may learn a very large number of operators, which may actually hamper the search of the planning system, the MORRIS [Min85] learner limits the maximal number of macro operators that can be memorized at any given time. Furthermore, this work was further extended [Moo88] to support partial ordering of actions in a macro operator.

While the previously presented methods learn macro operators leveraging domain and/or planner knowledge, a more general approach by Newton et al. [New+07] uses genetic algorithms to learn such operators from solution traces. MUM [CVM14] employs a different technique to learn similar macro operators with much lower learning times, guiding its search using *outer entanglements*, that is causal relations between planning operators and initial/goal state predicates. Pushing the reduction of learning times further, the OMA learner [CVM15] does away with the online training phase and quickly extracts macro operators from domain and problem couples before using a planner. Its performance remains competitive with offline approaches.

2.4.2.2 Hierarchical Planning Domains

In order to give an overview of the different approaches for learning HTN domains, and to situate our proposed approach in this context, we present a summary of the characteristics of the main approaches in Table 2.2.

Early Works We can find the idea of learning hierarchical domain knowledge as far back as the work on the X-Learn system [RT97]. This system learns from exercises, i.e. problems of increasing difficulty, generalizing its solutions to extract *d-rules* relating goals, subgoals and conditions. In a d-rule, a top level goal and a set of conditions that determine its applicability in a given state are associated with a (possibly recursive) totally ordered decomposition into subgoals. This is similar to the structure of HTNs, but the use of goals instead of tasks pushes it closer to Hierarchical Goal Networks (HGNs)[1][Shi+12].

The work by Nejati, Langley and Konik [NLK06] introduces the idea of leveraging *goal regression*[2], to learn hierarchical domain knowledge in the form of *teleoreactive logic programs* from demonstration traces. This approach uses a different terminology than our HTN formalism: in this framework, *primitive skills* represent sequences of actions, with pre-conditions and effects, abstracting over lower level actions. They can be seen as primitive actions in our formalism. *High-level skills* are similar to methods in HTNs, but the head of a method is the *goal* achieved by it rather than the *task* achieved, which also pushes this approach towards HGNs.

Using goal regression, the learner tries to chain skills together to explain and generalize the demonstrated traces, iteratively achieving (sub)goals and outstanding skill pre-conditions. This approach heavily relies on the decomposition structure bias given by the set of concepts which are given as input to the learning problem, thus requiring non-negligible domain expertise, similar to giving annotated tasks as input in other approaches.

[1] For an overview of HGNs, see Background A on page 30.

[2] In a nutshell, goal regression is the act of finding an action a that achieves part of a goal g, then finding another action a' that achieves the remaining parts of g as well as the preconditions of a, and repeating this process until nothing is left to achieve.

Approach	Input	Output HTN: New Tasks	Output HTN: Lifted	Output HTN: Method Pre-Conditions	Noise	Observability	Non-Determinism
[NLK06]	Solution traces Goals	✗	✓	✓	✗	Full	✗
HTN-Maker [HMK08]	Solution traces Annotated abstract tasks	✗	✓	✓	✗	Full	✓[1]
HTNLearn [ZMY14]	Partial decomposition trees Annotated abstract tasks	✗	✓	✓	✗	Partial	✗
HierAMLSI [GPF22]	Random Walks	✗	✓	✓	✓	Partial	✗
Word2HTN [GMK18]	Solution traces	~[2]	✓	✓	✗	Full	✗
[LJ16]	PDDL domain & instance	~[2]	✓	✓	✗	–	✗
[SPF17]	Solution traces	~[3]	✓	✓	✓	Partial	✗
pHTN [Li+14]	Solution traces	✓	✗	✗	✗	Full	✗
CircuitHTN [Che+21]	Solution traces	✓	✗	✗	✗	Full	✗
CC-HTN [HS16]	Solution traces	✓	✗	✗	✗	Full	✗
Ours	Solution traces Top level task with effects	✓	✓	✗	✗	Full	✗

Table 2.2: Summary of the main HTN learning approaches and comparison with our proposed approach.

[1] In extended version [HKM09].
[2] Limited to single predicate goals.
[3] Handling of recursion and parameterization scaling unclear.

Learning Parameterized HTNs Let us now present approaches that learn HTN structures closer to our presented formalism, starting with approaches that learn HTNs where tasks, methods and actions are parameterized using variables.

The HTN-Maker [HMK08] algorithm learns HTNs from solution traces and tasks annotated with pre- and post-conditions, leveraging again goal regression. These annotated tasks constrain the possible HTN structures that can be learned, similarly to concepts in the previously presented approach, as no new abstract tasks will be generated by the algorithm. The HTN-Maker learner consider subsequences in the set of input plans, and traverse each of these subsequences from the end, detecting whenever a task t's post-conditions are achieved in a state s_f that comes after a state s_i where t's pre-conditions are holding. This subsequence can then be considered as a method that achieves t in the context of the state s_i. One of the innovations proposed by this algorithm is to not only use goal regression through the actions in a plan trace, but also through the set of previously learned methods. This regression through previously learned methods is made possible by keeping track of the encountered *method instantiations*, and reusing them through a recursive call to the parent task of such an instantiation.

One downside is that the resulting HTN structures end up highly recursive, which is hard to use for automated planners in practice, as shown for example by Fine-Morris et al. [Fin+22]. The HTN recursivity is even more pronounced in the extension of HTN-Maker to nondeterministic domains [HKM09], as a right recursive structure is enforced to handle the potential nondeterministic effects of the actions. A further extension of this algorithm has seen it coupled it with Reinforcement Learning (RL) [HKM10] to determine quality values for each method in a given state, with the intent of finding a way to generate not only valid plans, but also *good* plans. This notion of plan quality can be related to encoding user preferences in HTNs, as done in other works dedicated to learning probabilistic HTNs (pHTNs) [Che+21; Li+14].

With the HTNLearn algorithm, Zhuo, Muñoz-Avila and Yang [ZMY14] propose an approach that is able to learn both the HTN (its structure and method pre-conditions) as well as the primitive action models, using a weighted MAX-SAT approach similar to the one presented in ARMS [YWJ07]. It extends the author's own work on the HTN-Learner [Zhu+09] algorithm which was limited to learning method preconditions and action models.

This approach takes as input solution traces and annotated tasks, as does HTN-Maker, but also partial decomposition trees, which requires knowing at least part of the hierarchical structure to generate the input data, requiring more work than flat solution traces. While still focusing on deterministic domains, this approach is able to handle partially observable states in the demonstration traces. The learning procedure is mapped to a maximum satisfiability (MAX-SAT) problem, where the partial decomposition trees impose hard constraints on the structure of the resulting HTN, and the demonstration traces provide soft constraints. These soft constraints encode facts such as "a predicate that precedes a given action may be a precondition of this action". If this precedence is observed often, this soft constraint will be given a larger weight, making it more important to satisfy than other, less frequently observed, similar constraints, which allows the algorithm to handle partial observability.

The HierAMLSI approach [GPF22] also requires knowing all the abstract tasks in the hierarchy, but does not require their pre- or post-conditions. However, they require an oracle able to output a plan corresponding to a decomposition of the task if any exists, randomly sampling tasks to decompose. This oracle, however, may output partial and noisy sequences. This approach then learns an automaton modelling state transitions (induced by both primitive and non-primitive tasks) from which methods can be extracted. The set of methods is built from an initial set of methods with primitive subtasks only, iteratively trying to integrate the hierarch-

ical information in the automaton to efficiently minimize the final set of methods. The system then learns the action models as well as the method preconditions using the AMLSI approach [GPF20].

Doing away with the requirement for annotated tasks as input, the WORD2HTN system [GMK18] uses *word embeddings*, from the field of Natural Language Processing (NLP), and more precisely the WORD2VEC [Mik+13] system, generating vectors that allow situating predicates and actions in a latent space. Clustering these atoms according to their distance in this latent space allows finding *bridge atoms*, which allows determining subgoals that must be achieved to achieve a task. Doing this clustering hierarchically generates decompositions of goals into subgoals, putting the learned models more towards the HGN formalism [Shi+12]. The structures elicited by this approach are however highly constrained by the size of the clusters given as an input parameter, leading to methods which always decompose into two subtasks in the author's paper, which may limit the quality of learned HTNs domains. A very similar approach has been applied to a simplified Minecraft-based domain [Ngu+17], moving the learning of HTNs from domains specific to the planning community to ones inspired from video games, which are a common real world application of HTNs [KBK07].

To further reduce the need for human annotated tasks given as input, the learner by Segura-Muros, Pérez and Fernández-Olivares [SPF17] leverages *process mining*[1] techniques, specifically the approach by Leemans, Fahland and van der Aalst [LFvdA13a]. This algorithm uses a process miner to generate a process tree that is then converted to an HTN structure. Parameters are then propagated upwards from the primitive task up the hierarchy. This approach does not however exploit knowledge of the top level task achieved by a given trace, and it is unclear how the argument propagation works in case of recursive tasks and how well it scales to larger HTN domains.

Focusing on learning domain with numerical state variable, the T2N [Fin+22] algorithm pushes further the work done on WORD2HTN [GMK18]. Instead of learning binary methods, landmark goals are extracted using a similar technique, and then methods are learned to decompose each landmark, either with a structure similar to HTN-MAKER or as flat sequences of primitives. Multiple method structures are compared, and the results show that purely right-recursive method perform poorly, showing the limits of HTN-MAKER-like methods.

The approach presented by Lotinac and Jonsson [LJ16] learns HTNs from PDDL domains and a single associated planning problem, without the need for solution examples nor additional knowledge about the structure of the domain. It uses *invariant graphs*, which are, broadly, lifted transition graphs between mutually exclusive fluents, to generate tasks and methods to achieve some of these fluents. While this method produces solid results on some domains, enough for the resulting HTNs to be included as benchmark instances in the 2020 IPC [BHB21], the quality of the resulting hierarchies is highly dependent on the structure of the invariant graphs of a given domain. Furthermore, the resulting domains do not feel natural compared to one written by a human expert and are hard to interpret.

Learning Grounded HTNs While the previously presented approaches learn parameterized HTNs, some approaches only learn grounded HTNs, that is HTNs with no variable parameters.

Li et al. [Li+14] propose learning probabilistic HTNs (pHTNs), which are used to consider user preferences, as a probabilistic Context Free Grammar (CFG). The algorithm works first by finding the structure of the grammar with a crude version of *pattern mining*, and then applies

[1] Process mining [Van12] is the extraction of data from real world processes, often through the analysis of event logs.

expectation maximization to learn the probability associated with each method. This approach, however suffers from a limitation inherent to grounded HTNs: it requires manually selecting relevant parameters in the ground actions to avoid an exponential explosion of the number of actions. Furthermore, it limits the HTN structures to be similar to Chomsky Normal Form (CNF) grammars in order to efficiently extract the most probable parse of a sentence as part of the learning procedure. These structural limitations will generate hard to understand models and may render the learned HTN inefficient when used with an automated planner.

The CircuitHTN [Che+21] system learns similar grounded pHTN. However, the approach used is based around the fact that demonstrations can be used to build *action graphs* which represent the possible paths through all the demonstration set. These action graphs are then considered as an electrical circuit, with each action represented by a resistance, and the problem of learning the HTN is cast as reducing all the resistances in the network to a single equivalent resistance. According to the authors' evaluation, on the considered domains, their method learns smaller and more correct domains than the approach by Li et al. [Li+14].

The CC-HTN approach [HS16] also relies on the structure of action transitions to learn grounded HTNs, finding chains and cliques in the graph, allowing a partial order of tasks. While this algorithm does not rely on specific symbolic representations for states and actions, subgoals reached in any given state must be known in the task graph, and obtaining the grounded actions require work similar to the other approaches to select relevant parameters.

Background A: *Hierarchical Goal Networks*
While HTNs is the most used framework for planning, Hierarchical Goal Networks (HGNs) [Shi+12] is another possibility. Contrary to HTNs, methods are defined as with *subgoals* instead of subtasks. A method m is then defined with:

- A head $\text{head}(m)$, similar to that of an action.
- A sequence of subgoals $\text{sub}(m) = \langle g_1, \ldots, g_k \rangle$, with each g a set of literals.
- Pre-conditions, as in the HTN case.
- *Post-conditions*, defined as $\text{post}(m) = g_k$.

Method or action instances are said relevant for a (sub-)goal g if their postconditions or effects entail at least one literal in g. Instead of classical planning method chaining or HTN decomposition rules, methods or actions can only be applied if they are relevant to the currently solved goal.

2.4.3 Other Hierarchical Models

Besides HTN domains (and HTN-like models), other approaches share some characteristics, in their structure, their goal or both.

2.4.3.1 Grammar Inference

Due to their similarities with HTNs [EHN94], it is not surprising that grammar learning approaches are of interest. While most of this work is done in the context of NLP, where the amount of data available is vastly superior to what is available in the context of robotics, grammars are also used in other contexts, such as goal recognition [GG11] and plan validation

[BMC19]. These are contexts where planning-based approaches are also used [Höl+18; SRU16]. The use of planning techniques in these contexts where grammars are used, as well as the use of grammar learning for learning HTNs [Li+14] and the similarity between HTNs and formal grammars highlights the relevance of the field of grammar inference for HTN learning.

As it has been shown that even relatively simple grammars (in the Chomsky hierarchy) cannot be learned from positive examples alone [AK95; Gol67], using negative examples for grammar inference seems natural. However, it is more difficult to obtain negative examples rather than positive examples, in the context of NLP, as there exists large bodies of correct text that are readily available, but no such amount of data is available for incorrect text. This difficulty to leverage negative example is visible in the survey by D'Ulizia, Ferri and Grifoni [DFG11], where eleven out of the fourteen listed approaches rely on positive examples only. Therefore, some approaches have leveraged a simplicity bias to learn grammars from positive examples only, leveraging the idea that between multiple possible grammars, the simplest one that is able to generate the target language should be preferred. Among such approaches, we can cite the work of Grünwald [Grü96] and the GRIDS algorithm [LS00], which use a metric based on the Minimum Description Length (MDL) principle coupled with a local search mechanism. The work by Sapkota, Bryant and Sprague [SBS12] applies the e-GRIDS algorithm [Pet+04] to a real world Domain Specific Language (DSL) to generate the corresponding grammar

While the previously cited works learn Context Free Grammars (CFGs), more recent approaches focus on Combinatory Categorial Grammar (CCG), a more expressive class of grammars, which have been used in the context of plan and goal recognition [GG11]. In this research avenue, we can cite the GENLEX algorithm [ZC05] which learns probabilistic CCGs (pCCGs) in the context of matching sentences and logical meaning. Learning is done by alternating structure hypothesis, to generate grammatical categories, and parameter estimation to assign probabilities to these categories. Bisk and Hockenmaier [BH12] propose another approach for pCCG induction, but they leverage the availability of well-defined atomic categories in the context NLP and tagged sentences to learn complex categories, following this grammar learning with Expectation Maximization (EM) parameter estimation. It should be noted that this structure hypothesis followed with EM parameter estimation is used similarly for learning HTNs as probabilistic CFGs (pCFGs) [Li+14].

More recently, the LEX_{learn} family of approaches [GK18; KOG19] propose to learn pCCGs specifically in the context of plan recognition. The original approach [GK18] used a structurally restricted exhaustive generation of complex categories from plan examples, in order to avoid overfitting grammars, estimating the parameters using gradient ascent. A more recent version, LEX_{greedy} [KOG19], uses very simple pattern mining for finding frequent sequences of actions in order to extract the grammar's categories. This is similar to the work by Li et al. [Li+14]. This pattern mining relaxes the necessary structural constraints put on the learned grammar and allows the approach to scale to longer plan traces.

2.4.3.2 Behaviour Trees

Let us now turn our attention to Behaviour Trees (BTs), a kind of transparent hierarchical policy. Contrary to HTNs, they are providing a reactive way to use skills, and would therefore be part of the *acting* component in the architecture of our deliberate actor, lacking the deliberation capabilities intrinsic to planning models. While BTs are modular and human-understandable, it is still a difficult task to design a good BT for an agent, hence why several approaches have been developed for learning or synthesizing them automatically.

Because BTs share a tree-like structure with HTNs, we include techniques for their acquisi-

tions in this survey section. These similarities between HTNs and BTs have been noted before in the literature [RGK17; RW15], with work going as far as combining both approaches [NMB18], using BTs as the primitive actions of an HTN.

Genetic Programming (GP) techniques are commonly used for learning these structures, starting with the work of Perez et al. [Per+11] using Grammatical Evolution (GE), which uses a hand-designed CFG grammar to constrain the evolution of the individuals during the search, to learn BTs controlling a video game agent. Their results show particularly good reactive behaviour encoding, however the authors note that long-term decision capabilities are better handled by planning systems. Other approaches also use GP algorithms, such as the work by Colledanchise and Ögren [CÖ18] and Zhang et al. [Zha+18]. The first approach [CÖ18] uses a combination of greedy modifications of the BTs and GP-based evolution to learn more efficiently, and provides some safety guarantees due to the way BTs are composed together during learning. The second approach [Zha+18] focuses on the constraints used to learn BT, proposing to add dynamic constraints on the structure of the learned BTs, contrasting with the static constraints used in the previously highlighted approaches. These dynamic constraints are generated by identifying and preserving frequent subtrees in the population, following the rationale that these subtrees must encode behaviour that has a beneficial effect on an individual's fitness, and thus should help the search converge to a good solution more efficiently.

Approaches based on the Learning from Demonstration (LfD) paradigm have also been used for the learning of BTs, such as the BT-Espresso algorithm [Fre+19]. This algorithm first converts the demonstrations into decision trees using the C4.5 algorithm [Qui93] to learn decision trees from teacher demonstrations, before converting them into BTs.

Aside from learning to automatically acquire BTs, synthesis approaches have also been proposed. One of these approach uses Linear Temporal Logic (LTL) [CMÖ17] to specify the desired behaviour that the policy has to achieve, before synthesizing a BT conforming to this specification. Another approach uses automated planning [CAÖ16; RGK17] to build the tree: starting with an empty tree, each time we cannot achieve a goal, the tree structure is updated from the plans in order to handle this situation directly in the BT next time.

2.4.4 Generalized Planning

Following the definitions given in recent works [HD11; JSJ19; Lot17; SIZ11], we consider *generalized planning* as the problem of finding plans that work for several instances. This notion is generic and does not rely on a specific planning framework, and we propose to present it using an adapted version of Hu and De Giacomo [HD11]'s formalism. Our goal is here to give the reader an intuition of the generalized planning problem, with a focus on making a link with the planning formalisms that we use, rather than developing a complete generalized planning framework.

We first need to consider an *agent* with a set of actions and observation $\mathcal{A} = (Act, Obs)$ where Act is the set of actions our agent can perform and Obs is the set of observations it can make. An agent's *plan* is defined as $p : Obs \times Act \cup \{\texttt{stop}\}$, a function that maps observations to actions, where `stop` stands for plan termination.

We also consider a deterministic *environment* $\mathcal{E} = (Events, S, s_0, \delta)$ in which the agent's actions are executed such that:

- *Events* is the set of all events in the environment.
- S is the set of possible spaces in the environment.

- $s_0 \in S$ is the initial state of the environment.
- $\delta = S \times Events \to S$ is the (partial) transition function.

To execute the agent's plan in the environment, we define two functions to determine how the agent's actions and observations are related to the environment's events and state respectively:

- $obs : S \to Obs$ determines how much of the environment the agent can observe.
- $exec : Acts \to Events$ maps the agent's actions to the events they cause in the environment.

These functions allow handling partial observability through the *obs* function, but also non-determinism. Indeed, even if the environment itself is deterministic, the *obs* mapping may hide some context, which may in turn make the transition appear non-deterministic to the agent [Sri10].

A basic planning problem is then defined as a tuple $\mathcal{P}_b = (\mathcal{A}, \mathcal{E}, obs, exec, G)$, where G is a goal, i.e. a partial state specification[1]. The agent has to find a solution plan so that the last state after executing this plan fulfils G.

A *generalized planning problem* is then a set of basic planning problems $\mathcal{P}_g = \{\mathcal{P}_b^1, \mathcal{P}_b^2, \dots\}$ where the $\mathcal{P}_b^i$ share the same agent. The goal of this generalized planning problem is then to find a plan p_g that is a solution for every $\mathcal{P}_b^i \in \mathcal{P}_g$.

To give an intuition of what this means, we apply this definition to the case of classical planning problems and domains. Because of the absence of exogenous events and in the case of full observability, we have $Act = Events = A$, the set of planning actions, and $Obs = S$, our classical state space. We can therefore roughly consider that the agent is a given domain, and that the environment is a planning instance[2], with it its initial state and goal. A generalized plan p_g is therefore, in this case, one that can solve every classical planning instance associated with a given domain. In this sense, a classical planner *is* a generalized plan, albeit an inefficient one.

Given this definition of a generalized plan, we can easily notice that it encompasses many approaches, with different levels of constraint imposed on the next action to choose at any given time. On one end of the spectrum, we have deterministic policies and computer programs, fully specifying the next action, while on the other end, we have classical planners, which do not specify the actions to apply at all. In between these extremes, we find partially specified solutions, requiring a search mechanism to produce a solution, but constraining the search space with domain specific knowledge. It is easy to notice that HTN domains (with a hierarchical planner) therefore can be seen as generalized plans. We will therefore present an overview of the approaches that exist to obtain these generalized plans, especially in relation with our goal of automatically acquiring domain knowledge. We refer the reader to the survey by Jiménez, Segovia-Aguas and Jonsson [JSJ19] for a more complete overview.

In the fully-specified category, we can cite *generalized policies* [MG04], which are rule sets extracted from demonstrated instances. The authors note that their approach can learn complete plans given enough training instances, as it does an exhaustive search over the possible rule sets, but tends to not find optimal solution. Furthermore, the exhaustive search limit the scalability of the approach. Another fully-specified solution learns features and abstract actions before synthesizing a policy [BFG19].

[1] The original formalism [HD11] provides a more general definition of a goal, not required for our presentation purpose.

[2] To conform to the definition, the environment would need to be supplemented with the transition function. This would, however, make the equivalence less intuitive, defeating the purpose of this paragraph.

Moving away from learning and towards synthesis, we can note the *planning programs* approach [JJ15]. Here, classical planning domains and sets of instances are compiled to program-like constructs with control-flow operators and procedures. This approach is a good way to understand how Programming by Demonstration (PbD) can also find its place in the context of generalized planning.

We also note that RL-based policies also fit in this context, especially considering the recent advances using graph neural networks to apply deep learning techniques to symbolic domains [KS20; SBG22]. The automatic acquisition of BTs, presented earlier, can also be considered as solving a generalized planning problem, as do planning techniques dealing with multiple contingencies, such as conformant and contingent planning.

Finally, let us note that HTN learning has already been considered in the framework of generalized learning: in his thesis, Lotinac [Lot17] argues that HTNs are used to reduce the complexity of the search space, and thus permits to reusing the same knowledge to solve problems that would be too difficult to solve using only classical planning. The learning approach used by the author is the invariant graphs-based one [LJ16] presented earlier.

2.4.5 Other Relevant Approaches

2.4.5.1 Programming by Example

Another approach that relates to our goal is that of Programming by Demonstration (PbD), where the goal is to obtain a computer program from a set of user-given examples. Real world examples can be found in tools such as Microsoft Excel's FlashFill, which provides automated text completion after typing a few examples of the desired output.

As mentioned in the section on generalized planning, programs are similar to policies, and are therefore more situated in the *acting* module of our actor rather the *planning* one. Nonetheless, we mention these approaches here due to the fact that they potentially have to handle different abstraction levels and to compose basic primitives.

Most PbD approaches use techniques to reduce the search-space to something manageable: in most cases, it uses some form of templating or Domain Specific Language (DSL), which can be seen as similar to giving annotated abstract tasks in approaches such as HTN-Maker [HMK08] and HTNLearn [ZMY14] or selecting action parameters for groundings [Che+21; Li+14]. All these techniques therefore require non-negligible effort from a domain expert before programs can be learned.

A well-known approach in programming by demonstration is the SKETCH [Sol09] algorithm. It uses sketches, a form of program with boolean and integer holes, written in a C-like language as templates to constrain the space of possible programs. From a sketch and set of input/output examples, the algorithm generates an equivalent set of constraints, such that solving the set of constraints is equivalent to filling the holes in the sketch. It then calls a validating oracle to check whether the synthesized program is valid for every possible input, which must returns a counterexample if this is not the case. While the approach works well, it has only been tested on low level tasks and obviously requires both work from a non-expert tutor, providing demonstrations and a domain expert to provide the sketches.

The OGIS [Jha+10] system is similar: it also requires input/output examples (in the form of queries to an oracle) and a validation oracle to determine if a program is correct or not. However, instead of sketches, the program uses a library of base components, that can be assembled into a program by solving an SMT problem. This system has similar requirements as the previous one: it requires demonstrations from a non-expert user as well as input from a domain expert,

choosing which components can be considered by the algorithm.

2.4.5.2 Process Mining

Process mining is a field of research mainly focused on discovering, monitoring and improving processes from event logs [Van12], where events may be at different abstraction levels. This area of research is mainly focused on real *business* processes, with the intention of extracting actionable insights for human analysts.

In this research field, process discovery focuses on extracting new processes from event logs, often in a format such as Petri nets for earlier works [Van12] or Business Process Model and Notation (BPMN) [CT12] for more recent works [vZel+21]. Another representation for process models is that of *process trees* [LFvdA13b], which are very similar to BTs. Improving processes is very similar, but starts with a model instead of building one from scratch. Monitoring of processes is used to assess the conformance to actual processes to a model, trying to determine the deviation from this model. This part is less relevant with regard to our goal of learning HTNs.

While processes are more rigid models than HTNs, being closer to policies, extracting them from event logs aligns with our goal of learning HTNs from demonstrations. A recent survey [Aug+19] highlights that most process mining algorithm have a focus on handling concurrent subtasks, which is less relevant with regard to our learning goal, given that we restrict ourselves to totally-ordered HTN formalism. Furthermore, abstraction extraction is still relatively recent and mostly relies on domain knowledge given by an expert [vZel+21]. Interestingly, we can observe that among the approaches that use process trees as a representation, evolutionary algorithms are highly represented [Aug+19].

Finally, it should be noted that most of these algorithms have a different focus than the work we present in this thesis:

- Recent approaches focus on handling very large event logs, which is antithetic to our goal of learning from sparse demonstrations.
- In most cases, these models are created to be used as a tool for human analysis, to obtain insights on some given process. Therefore, models do not have to be complete, Leemans, Fahland and Aalst [LFA14] citing models that cover 80% of the data while being only 20% of the true model. While this is good for process analysis, providing high level insights, this is not ideal for models used for planning.
- They do not deal with symbolic task and action parameters as is done in the HTN framework.

2.5 A Multi-Stage Iterative Learning Procedure

Let us now present the main components and ideas behind our proposed learning procedure to give the reader a better understanding of the interplay between the different components.

Algorithm 2.1 presents the base idea for the search. We first start from a base domain, either given as input, or a basic one automatically generated, which could be as simple as a useless domain without any method, unable to match any demonstration. We then generate the neighbouring domain structures, and evaluate them to find the best one. This process is repeated while the quality of the domain improves. Finally, we learn the parameters for the best HTN structure.

Algorithm 2.1 HTN Search - High-Level Process

1: $\mathcal{H}^* \leftarrow$ Generate Base Domain
2: **while** Quality($\mathcal{H}^*$) improves **do**
3: $\quad \Theta_c \leftarrow$ Gen Candidate Domain Structures($\mathcal{H}^*, D$)
4: $\quad \mathcal{H}^* \leftarrow$ Find Best Domain($\Theta_c \cup \{\mathcal{H}^*\}$)
5: $\mathcal{H}^* \leftarrow$ Extract Domain Parameters($\mathcal{H}^*, D$)

While most HTN-learning approaches learn structure and parameters simultaneously, one can view approaches that learn grounded HTNs [Che+21; HS16; Li+14] as learning only the structure of the HTNs, albeit with a larger number of actions. Furthermore, as cited earlier, one approach leveraging process mining techniques [SPF17], learns the structure of the HTNs and then adds parameters. Similarly to this work, this separation allows our work to draw on insights from other research domains, namely that of grammar learning and process mining. One key difference, however, is that our approach propose a much more comprehensive method for adding parameters to the learned HTN structure.

We now turn our attention to each of the steps of this process, summarizing them before going into more details in the next chapters.

HTN Structure Learning Because of the size of the space of structurally valid HTNs, the candidate generation should skew toward relevant structures while still being able to escape local optima.

To this end, we leverage techniques from *pattern mining* [ABH14] to abstract frequently co-occurring behaviours that can be used as subtasks to bootstrap the candidate generation, especially using an adapted version of the HTN-Maker algorithm [HMK08].

HTN Parameters Learning In order to parameterize the HTN structure resulting from the previous stage, a straightforward intuition is that parameters can be propagated through the hierarchy both from the primitive tasks and the demonstrated ones. However, it is easy to see that most of these naively propagated parameters will actually not propagate much information, neither horizontally across the methods' subtasks nor vertically across the tasks. Therefore, we propose a MAX-SMT-based approach to try and unify parameters to improve the situation.

Evaluating Candidate HTN Domains While the most natural way to evaluate the quality of a given HTN would be to evaluate its planning capabilities, and is indeed one of the metric that will be used to evaluate the global approach, it is prohibitively expensive to run during the structure phase search. We will therefore use a surrogate metric based on the Minimum Description Length (MDL) principle, which is easy to express given the similarities between HTNs and CFGs. Therefore, we will be able to search towards efficient domains able to reproduce the demonstrated behaviours for a relatively low computational cost.

Chapter 3

Learning Hierarchical Task Networks Structure from Demonstrations

Contents

3.1 Introduction . . . 37
3.2 Generating Neighbours of HTN Structures . . . 39
3.2.1 Goal Regression without Explicit Goals . . . 40
3.2.1.1 The HTN-Maker Algorithm . . . 40
3.2.1.2 HTN-Maker Without Arguments: Simplifying the Original Algorithm . . . 43
3.2.2 Frequent Pattern Mining for Neighbour Generation . . . 46
3.2.2.1 The Considered Patterns . . . 47
3.2.2.2 Substituting Patterns in Demonstrations and Extracting Neighbours . . . 48
3.2.2.3 Building a Set of Compressing Patterns . . . 49
3.2.2.4 Searching for the Best Abstraction Set . . . 50
3.2.3 Simplifying a Candidate Structure . . . 51
3.3 Evaluating Candidate Models . . . 53
3.3.1 The MDL Principle for HTN Structures Evaluation . . . 53
3.3.2 Obtaining Decomposition Trees from HTN Structures and Action Sequences. . . . 58
3.4 The Complete Structure Search Algorithm . . . 61
3.5 Conclusion . . . 62

3.1 Introduction

As was presented earlier in Chapter 2, Section 2.3, our proposed approach splits the learning of the HTNs domains into two steps, first learning the structure and then the parameters. This chapter will focus on the first stage.

Because we focus on learning the *structure* of HTNs domains, we will only consider *structural* information from our learning problem. In this context, the learning dataset can be simplified by removing all parameters. Let us reuse the one presented in the previous chapter (Figure 2.11), and highlight the differences this simplification makes in Figure 3.1, mainly allowing to also remove the intermediate states. While it is not shown on the figure for clarity, every precondition and effect is also removed, even the ones that rely on 0-ary predicates.

Because we are working with TO HTNs, we do not deal with interleaved tasks belonging to the same high-level tasks. We therefore assume that there exists an underlying TO hierarchical structure behind the generation process of the demonstrations. This is trivially the case if they

$$T_P = \begin{Bmatrix} \texttt{move}(?t : \textit{truck}, ?l_1 : \textit{location}, ?l_2 : \textit{location}) \\ \texttt{load}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{unload}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \end{Bmatrix} \mapsto \begin{Bmatrix} \texttt{move} \\ \texttt{load} \\ \texttt{unload} \end{Bmatrix}$$

(a) Initial set of primitive actions and simplified version with removed parameters.

$$T_I^1 = \{\texttt{deliver}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package})\} \mapsto \{\texttt{deliver}\}$$

$$D_1 = \begin{Bmatrix} \texttt{deliver}(t_1, l_3, p_1) : \langle s_0 \rightarrow \texttt{move}(t_1, l_1, l_2) \rightarrow s_1 \rightarrow \texttt{load}(t_1, l_2, p_1) \rightarrow s_2 \\ \qquad\qquad \hookrightarrow \texttt{move}(t_1, l_2, l_3) \rightarrow s_3 \rightarrow \texttt{unload}(t_1, l_3, p_1) \rightarrow s_4 \rangle \end{Bmatrix}$$
$$\mapsto \{\texttt{deliver} : \langle \texttt{move} \rightarrow \texttt{load} \rightarrow \texttt{move} \rightarrow \texttt{unload} \rangle\}$$

(b) A first set of initial vocabulary with an associated set of demonstrations and the corresponding simplified version with removed parameters.

$$T_I^2 = \begin{Bmatrix} \texttt{get}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{dropoff}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \\ \texttt{deliver}(?t : \textit{truck}, ?l : \textit{location}, ?p : \textit{package}) \end{Bmatrix} \mapsto \begin{Bmatrix} \texttt{get} \\ \texttt{dropoff} \\ \texttt{deliver} \end{Bmatrix}$$

$$D_2 = \begin{Bmatrix} \texttt{deliver}(t_1, l_3, p_1) : \langle s_0 \rightarrow \texttt{get}(t_1, l_2, p_1) \rightarrow s_2 \rightarrow \texttt{dropoff}(t_1, l_3, p_1) \rightarrow s_4 \rangle \\ \texttt{get}(t_1, l_2, p_1) : \langle s_0 \rightarrow \texttt{move}(t_1, l_1, l_2) \rightarrow s_1 \rightarrow \texttt{load}(t_1, l_2, p_1) \rightarrow s_2 \rangle \\ \texttt{dropoff}(t_1, l_2, p_1) : \langle s_2 \rightarrow \texttt{move}(t_1, l_2, l_3) \rightarrow s_3 \rightarrow \texttt{unload}(t_1, l_3, p_1) \rightarrow s_4 \rangle \end{Bmatrix}$$
$$\mapsto \begin{Bmatrix} \texttt{deliver} : \langle \texttt{get} \rightarrow \texttt{dropoff} \rangle \\ \texttt{get} : \langle \texttt{move} \rightarrow \texttt{load} \rangle \\ \texttt{dropoff} : \langle \texttt{move} \rightarrow \texttt{unload} \rangle \end{Bmatrix}$$

(c) A second set of initial vocabulary with an associated set of demonstrations and the corresponding simplified version with removed parameters. Note how the demonstrations make use of the additional abstraction levels to remain maximally abstract.

Figure 3.1: Example of initial task vocabulary and corresponding maximally abstract demonstration sets for a `deliver` task and the corresponding simplified version without parameters. All the demonstration sets corresponds to the same sequence of primitive actions but are abstracted at different levels. Here, the considered optimality metric is the number of actions in the plan. Note how the removal of parameters also removes intermediate states, as they then do not contain any information.

are generated using an oracle TO HTN domain, and can be considered as true if they are given by a human tutor to explicitly *teach* the agent. If a classical planner is used as an oracle, note that this assumption does not always hold, which may lead to suboptimal structures being generated. It is especially true if the goal associated with a given task can be decomposed into multiple independent subgoals: the planner may interleave tasks to solve these subgoals.

We define an *HTN structure* as:

Definition 3.1 (HTN Structure)**.** An *HTN structure* is a hierarchical planning domain without any arguments. This means that the head of a task is restricted to its name, and that preconditions and effects are empty.

Additionally, we say that for a domain $\mathcal{H}$, the corresponding HTN structure H *matches* a plan π (as defined in the classical planning sense) if and only if, ignoring the parameters of the actions in π, there is a decomposition of H into a solution such that its linearization is equal to π.

Unless otherwise specified we are considering HTN *structures* in this chapter. Therefore, tasks and methods do not have parameters and will be represented only through their *symbols*.

Because the space of possible HTNs is extremely large, we want to be able to focus our search on promising regions of this space. To this end, we will use a *local search* algorithm, with the goal of iteratively improving the coverage of the demonstration set, while still keeping generalization capabilities and not degenerating into a lookup of the demonstration set.

Because we are working with a local search algorithm, we need to define the neighbourhood of a given HTN structure and the operators to generate it, which will be the focus of the next section. The following section will then focus on a method to efficiently evaluate the quality of a given structure with regard to the previously stated goal. Finally, we will present the global structure search algorithm and the constraints that led to its current version.

3.2 Generating Neighbours of HTN Structures

As stated earlier, we assume that we start with a vocabulary of T_I of initial tasks, a set D of demonstrations of the tasks in T_I, and an initial set of methods M_I. This initial set of methods may have been given by a human expert or be the result of a previous learning attempt. In the most trivial case, M_I may be empty.

Our goal here is to generate a set of abstract tasks T_L and a set of methods M_L such that the resulting HTN structure is able to efficiently match the demonstrations in D. More details on this matching process will be given in Section 3.3.2 of this document. Learning new abstract tasks, and not only methods for the tasks in T_I, is a way to share behaviours across the hierarchy.

It should also be noted that we do not aim to find a structure that is able to *only* decompose into the sequences present in the demonstration set. Because we are learning with HTN domains that are intended to be used with planners, not simple reactive policies, we only wish to restrict the search space of these planners to limit the effort required to obtain a plan, but can rely on their ability to efficiently explore the space of possible decompositions. In this sense, our hierarchies aim to model *advice* and not dynamics of the environment [McD00], the latter being handled by the (known) preconditions and effects of the primitive actions.

We can divide the possibly generated neighbourhoods into two categories, which will each be detailed in the following subsections:

- Learn new methods from a single trace, through an adapted version of the HTN-Maker [HMK08] algorithm.
- Learn new abstract tasks and associated methods for frequent patterns, possibly abstracting behaviours across multiple demonstrations.

3.2.1 Goal Regression without Explicit Goals

In order to quickly generate neighbours that improve the coverage of a given candidate, we decided to adapt the goal regression algorithm found in HTN-Maker [HMK08].

We will start by presenting this algorithm, applied in the context of our search of HTN *structures*, before highlighting how this context impacts it, as well as the possible modifications it allows.

3.2.1.1 The HTN-Maker Algorithm

At a high level, the HTN-Maker algorithm uses abstract tasks annotated with pre- and post-conditions to determine subsequences in a given demonstration that achieves these tasks. These subsequences are then used to generate new methods, which are added to a given HTN to increase its coverage of the demonstrations tasks. For a single demonstration d which achieves task t, let us first present how we can extract methods from this demonstration using the same technique as in HTN-Maker. Here, for a demonstration d of a task t, we want to learn a set M_d of sets of new methods for t that covers d.

Algorithm 3.1 shows this procedure, which represents the application of the HTN-Maker algorithm, as described in the original paper [HMK08], in our simplified context without parameters and states. Thanks to this simplified context, the Learn subprocedure presented in the original paper becomes simple enough to be directly integrated in this algorithm, in the conditional statement.

Algorithm 3.1 Extract Methods(t, d)

Input: t is the currently demonstrated task
$d = \langle a_0, a_1, \ldots, a_n \rangle$

```
1:  M_I ← ∅                          ▷ Set of method instantiations, indexed by starting index
2:  M_d ← ∅                          ▷ Set of learned method sets for demonstration d
3:  for i ∈ n down to 0 do
4:      for j ∈ n down to i do
5:          if j = n then
6:              m ← NewMethod(a_i, a_{i+1}, ... a_j)
7:              R ← ∅
8:          else
9:              m ← NewMethod(a_i, a_{i+1}, ... a_j, t)
10:             R ← {M_I[x] | x ∈ [j + 1, n]}      ▷ Store required associated methods
11:         M_I[i] ← M_I[i] ∪ {m}
12:         M_d ← M_d ∪ {{m} ∪ R}
13: return M_d
```

In our application of the algorithm, we consider increasingly large subsequences starting from the end of the sequence (line 3). Note that the original algorithm also considers additional subsequences that can be detected through the pre- and post-conditions of t.

For each possible subsequence, we generate the methods m that correspond to parts of this subsequence (line 4): here, we either construct a trivial method containing the whole currently considered subsequence as subtasks (line 5) or we build methods that contain a shortened subsequence and a recursive call to the demonstrated task t (line 8). This generation of recursive methods is one of the main parts of the original HTN-Maker algorithm, using previously encountered method instantiations to insert calls to their parent tasks in order to regress goals through the hierarchy.

When learning this recursive structure, we also store a set R of associated methods to keep track of which methods are required in order to finish the recursive call to the demonstrated task, leveraging the set M_I of method instantiations, reasoning using indices to find the required methods. We need to store this set R with each learned method, because the candidate HTN is not built iteratively after each extraction, contrary to the original algorithm, so that we can generate multiple candidates from a single method search.

We then store the set $\{m\} \cup R$ in the set M_d of learned method groups, and update M_I with m. Example 3.1 presents the methods that can be generated using this procedure.

Example 3.1 *Method generation structure.*
Assuming we have a demonstration $d = \langle a_0, a_1, \ldots, a_4 \rangle$, the set of generated methods has the form:

$j \backslash i$	4	3	2	1	0
4	$\langle a_4 \rangle$				
3	$\langle a_3, a_4 \rangle$	$\langle a_3, t \rangle$			
2	$\langle a_2, a_3, a_4 \rangle$	$\langle a_2, a_3, t \rangle$	$\langle a_2, t \rangle$		
1	$\langle a_1, a_2, a_3, a_4 \rangle$	$\langle a_1, a_2, a_3, t \rangle$	$\langle a_1, a_2, t \rangle$	$\langle a_1, t \rangle$	
0	$\langle a_0, a_1, a_2, a_3, a_4 \rangle$	$\langle a_0, a_1, a_2, a_3, t \rangle$	$\langle a_0, a_1, a_2, t \rangle$	$\langle a_0, a_1, t \rangle$	$\langle a_0, t \rangle$

As can be seen, indexing methods by row and column (respectively j and i in the algorithm), $m_{2,2} = \langle a_2, t \rangle$ depends either on $m_{3,4} = \langle a_3, a_4 \rangle$ or on $m_{3,3} = \langle a_3, t \rangle$ which itself depends on $m_{4,4} = \langle a_4 \rangle$. In this example, the required methods set $R_{2,2}$ would be:

$$R_{2,2} = \{m_{3,3}, m_{3,4}, m_{4,4}\}$$

Slightly less trivially, $m_{0,1}$ depends on either one of the $m_{2,i}$ methods. These methods obviously in turn have their own dependencies, yielding $R_{0,1}$ as follows:

$$R_{0,1} = \{m_{2,2}, m_{2,3}, m_{2,4}, m_{3,3}, m_{3,4}, m_{4,4}\}$$

Below is a reproduction of the previous array, highlighting the first level of dependencies for $m_{0,1}$ to clarify the example.

$j \backslash i$	4	3	2	1	0
4	$\langle a_4 \rangle$				
3	$\langle a_3, a_4 \rangle$	$\langle a_3, t \rangle$			
2	$\langle a_2, a_3, a_4 \rangle$	$\langle a_2, a_3, t \rangle$	$\langle a_2, t \rangle$		
1	$\langle a_1, a_2, a_3, a_4 \rangle$	$\langle a_1, a_2, a_3, t \rangle$	$\langle a_1, a_2, t \rangle$	$\langle a_1, t \rangle$	
0	$\langle a_0, a_1, a_2, a_3, a_4 \rangle$	$\langle a_0, a_1, a_2, a_3, t \rangle$	$\langle a_0, a_1, a_2, t \rangle$	$\langle a_0, a_1, t \rangle$	$\langle a_0, t \rangle$

Comparing our algorithm to the original one, we conserve the increasingly large subsequences

and the use of recursive calls to reuse already found methods, but the main modification is that we are reasoning with indices rather than outstanding pre-conditions and achieved post-conditions. Indeed, because we are only interested in the structure of the HTNs, it would be unnatural to rely on pre- and post-conditions which require the use of parameters. Furthermore, because we consider that our demonstrations are optimal, this means that for a demonstration $d = \langle s_0, a_0, s_1, \ldots, a_n, s_{n+1} \rangle$ of a task t, any subsequence starting at index i and going up to n has the following properties :

- It achieves the task t starting in state s_i.
- There is no strictly better subsequence that starts in s_{i-1} to achieve the task t.

Because of our assumption that the demonstrations are maximally abstract, if another subsequence of d achieved the task t, then this subsequence would already have been replaced by an instantiation of t. Therefore, we do not need to determine if another subsequence could achieve t This is what allows us to simplify the original Learn procedure to reason only with indices.

While in the previous example, we have explained that for a given method, we have a *choice* of methods on which to depend, we have also shown that our algorithm considers the set of all the potential dependencies to construct R (Alg. 3.1, line 10). Note that R may include redundant methods. This choice was motivated by two reasons:

- As we will show later, there is an exponential number of minimal subsets with no redundancies.
- Redundant methods can trivially be removed in a post-processing step thanks to our demonstration matching process.

With this set of learned method groups M_d, for a given demonstration d of task t, and considering an initial HTN structure H with a set of learned methods M_L, for each group of methods $M_g \in M_d$, we can create a new candidate H' associated with the set of learned methods $M'_L = M_L \cup M_g$. Example 3.2 shows the generated HTN structures with the method groups presented in Example 3.1.

Example 3.2 *Generated HTN structures.*
Here, we present the corresponding HTN structure for the method group for $m_{2,2}$, assuming with started with an empty structure. Only a subset of the methods is actually required to decompose t to obtain the demonstrated subsequence $\langle a_2, a_3, a_4 \rangle$, as shown in Figure 3.3.

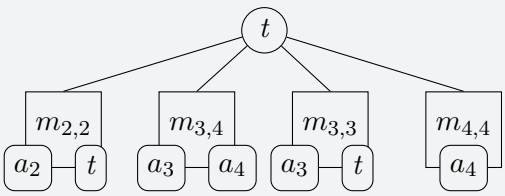


Figure 3.2: HTN structure for the method group of $m_{2,2}$.

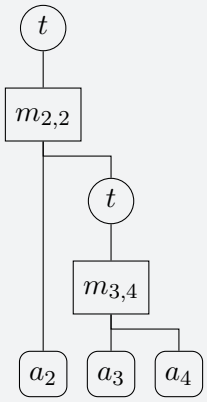


Figure 3.3: A possible decomposition of t into the subsequence $\langle a_2, a_3, a_4 \rangle$.

We can now present the global procedure used to generate neighbouring HTN structures, as presented in Algorithm 3.2, which simply extracts all possible groups of methods from the set of demonstrations D, and then create all the possible neighbours. Because we only reason on HTN structures, duplicate methods can be easily detected and not added.

Algorithm 3.2 Generate Neighbours HTN-Maker(D, H) - Initial Version

Input: D the set of demonstrations, with associated tasks
H a base domain structure from which to generate neighbours

```
H_N ← ∅                                          ▷ Set of neighbours
M ← ∅                                            ▷ Set of groups of methods
for all (t, d) ∈ D do
    M_d ← Extract Methods(t, d)
    M ← M ∪ M_d
for all M ∈ M do
    H' ← H + M                                   ▷ Add the methods from M to create the new neighbour
    H_N ← H_N ∪ {H'}
return H_N
```

Using this algorithm, the generated neighbours will add one set of methods for each task t, always providing at least one new way of decomposing the task to achieve the corresponding demonstrations if one exist. The added methods will always be either flat or right recursive, as in the original HTN-Maker algorithm.

By construction, each new candidate HTN structure H' will cover the demonstration d from which the methods were learned, while still covering any previously covered trace, as we only add new knowledge.

3.2.1.2 HTN-Maker Without Arguments: Simplifying the Original Algorithm

As can be observed in Example 3.1 (presented previously), the set of generated methods has a particular structure which exhibits a certain regularity. Therefore, for a demonstration $d =$

$\langle a_0, \ldots, a_n \rangle$, reordering the previous indices i and j, we can generalize this structure as follows:

$$\begin{array}{c} \\ 0 \\ 1 \\ \vdots \\ n-1 \\ n \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & \ldots & n-1 & n \end{array} \\ \begin{bmatrix} \langle a_0, t \rangle & \langle a_0, a_1, t \rangle & \langle a_0, \ldots, t \rangle & \langle a_0, \ldots, a_{n-1}, t \rangle & \langle a_0, \ldots, a_n \rangle \\ & \langle a_1, t \rangle & \langle a_1, \ldots, t \rangle & \langle a_1, \ldots, a_{n-1}, t \rangle & \langle a_1, \ldots, a_n \rangle \\ & & \langle \ldots, t \rangle & \langle \ldots, a_{n-1}, t \rangle & \langle \ldots, a_n \rangle \\ & & & \langle a_{n-1}, t \rangle & \langle a_{n-1}, a_n \rangle \\ & & & & \langle a_n \rangle \end{bmatrix} \end{array} \tag{3.1}$$

Therefore, writing r and c the indices for row and columns, with $r \in [0, n]$ and $c \in [r, n]$, we can write:

$$m_{r,c} = \begin{cases} \langle a_n \rangle & \text{if } r = n \wedge c = n \\ \langle a_r, \ldots, a_n \rangle & \text{if } r < n \wedge c = n \\ \langle a_r, \ldots, a_c, t \rangle & \text{if } r < n \wedge c < n \end{cases} \tag{3.2}$$

From, this we can easily define the method group $g_{r,c}$ associated with a given method as in Equation 3.3 below, which is a group containing the method $m_{r,c}$ and all the methods from previous levels that can be used to finish the end of the subsequence abstracted in $m_{r,c}$. As $m_{r,c}$ is structured as $\langle a_r, \ldots, a_c, t \rangle$, this means *all* the methods that can be used to reconstruct the subsequence $\langle a_{c+1}, \ldots, a_n \rangle$. Remember that we are not trying to generate a minimal set of methods that can achieve, but rather an overestimation of this set to limit the complexity.

$$g_{r,c} = \{m_{r,c}\} \cup \bigcup_{i=c+1}^{n} \bigcup_{j=i}^{n} \{m_{i,j}\} \tag{3.3}$$

Therefore, the neighbour generation algorithm presented earlier in Algorithm 3.2 can be rewritten as Algorithm 3.3. Because methods and tasks do not have parameters, and method ordering does not matter with regard to the semantics of the HTN, duplicate neighbours can be easily removed by introducing a canonical ordering of tasks and decomposition methods.

Algorithm 3.3 GENERATE NEIGHBOURS HTN-MAKER(D, H)

Input: D the set of demonstrations with associated tasks
H a base domain structure from which to generate neighbours
Output: H_N the set of generated neighbouring structures

```
H_N ← ∅
for all (t, d) ∈ D do
    n ← LENGTH(d)
    for r ∈ [0, n] do
        for c ∈ [r, n] do
            H' ← ADD METHODS(H, t, g_{r,c})        ▷ Add the methods from the group to t
            H_N ← H_N ∪ {H'}
return H_N
```

Using the direct formulation presented earlier and choosing any valid (r, c) tuple, it is also possible to generate an arbitrary valid HTN structure for decomposing a given task t associated with a given demonstration d without relying on the superset of possible methods. Equation 3.4 shows the formulation that allows to generate the set of all such groups of method $G'_{r,n}$, where

each group can be considered minimal in comparison to the previous one.

$$G'_{r,c} = \begin{cases} \{\{m_{r,c}\}\} & \text{if } c = n \\ \{\{m_{r,c}, m_{n,n}\}\} & \text{if } c = n-1 \\ \left\{ \{m_{r,c} \cup g\} \mid g \in \bigcup_{i=c+1}^{n} G'_{c+1,i} \right\} & \text{if } 0 \leq c \leq n-2 \end{cases} \tag{3.4}$$

However, the number of HTNs generated for a given demonstration d containing actions numbered from 0 to n is wildly different depending on the chosen method. Writing G_n the set of possible structures generated in the overestimation case and G'_n the one in the minimized case, their cardinality can be expressed as in Equations 3.5a and 3.5b. Here, the exponential size of $|G'_n|$ clearly shows the impracticality of generating an exhaustive set of neighbours relying on the minimal method groups. The minimized HTN generation can however be used when combined with random sampling.

$$|G_n| = \sum_{r=0}^{n} \sum_{c=r}^{n} 1 = O(n^2) \tag{3.5a}$$

$$|G'_n| = \sum_{r=0}^{n} \left(1 + \sum_{c=r}^{n-1} 2^{n-c-1} \right) = O(2^n) \tag{3.5b}$$

While we only considered the number of groups that can be generated in the previous equation, we must not forget that each group contains a certain number of methods which must also be generated. Fortunately, even considering the generation of all the possible methods in the case of G_n, the number of generated *methods* $|G'^m_n|$ remains polynomial, as presented in Equation 3.6. This allows an algorithm that generates the different associated candidates to remain tractable.

$$|G'^m_n| = \sum_{r=0}^{n} \sum_{c=0}^{r} \left(1 + \sum_{k=1}^{n-c} k \right) = O(n^4) \tag{3.6}$$

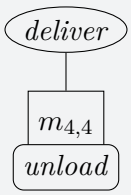
Example 3.3 presents HTNs structures that can be generated with a concrete demonstration, comparing the overestimated and the minimal method groups.

Example 3.3 *Methods and task hierarchies generated using the adapted HTN-Maker algorithm.*
Let us present a quick example of the structure of domains that can be generated using simply this algorithm for neighbour generation. In this example, will reuse the data presented in Figure 2.11, demonstrating the task `deliver`, and considering as demonstrations the set D, presented below. We will assume that we start with an empty HTN.

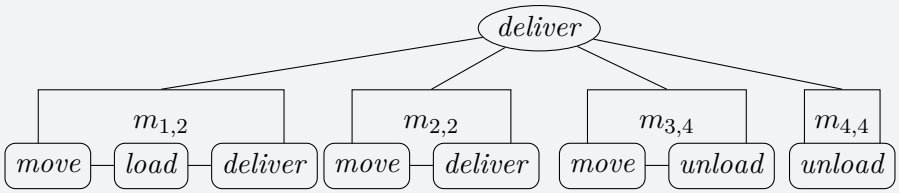
$$D = \{\texttt{deliver} : \langle \texttt{move} \rightarrow \texttt{move} \rightarrow \texttt{load} \rightarrow \texttt{move} \rightarrow \texttt{unload} \rangle\}$$

Figure 3.4: Demonstration set used as an example, recalled from Figure 2.11. Only actions are represented, as states are not used in this algorithm.

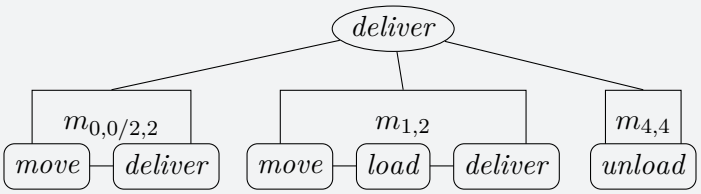
From this demonstration ($n = 4$), 15 HTNs could be generated using the overestimation method and 31 using the minimized version. Even though this number includes potential duplicates, we will only show a few of them to give an intuition about the shape of the domains that can be generated using the presented method.



(a) HTN for $r = 4, c = 4$.



(b) HTN for $r = 1, c = 2$ using the overestimation of required methods. Here, we can observe that $m_{3,4}$ is not strictly necessary when $m_{2,2}$ and $m_{4,4}$ are present.



(c) A possible HTN for $r = 0, c = 0$ using the minimized required method sets. Here we can observe that $m_{0,0}$ and $m_{2,2}$ would have been duplicates and only one version is required.

Figure 3.6: Possible HTNs generated using the modified HTN-Maker algorithm and random sampling.

3.2.2 Frequent Pattern Mining for Neighbour Generation

The HTN-Maker algorithm however suffers from several limitations:

- It cannot learn new intermediate tasks, preventing the system from abstracting common behavioural patterns.
- The learned models are highly recursive.
- In our application, it essentially learns different methods for each demonstration[1], with potentially little overlap.

This means that, assuming the only known abstract tasks that we have in our learning problem represent independent high-level behaviours for which we want to learn a hierarchy, HTN-Maker is only capable of generating hierarchies with a single level of methods, relying on recursive calls to achieve the task. However, this kind of hierarchy will quickly grow wide,

[1] In the original algorithm, this is somehow mitigated through method subsumption detection.

with many methods decomposing a given task, and will not encode behaviour in a manner that is both natural to understand and easy to use for an automated planner, as a large number of possible method instantiations will have to be considered for each recursion. If a multi-level hierarchy is desired with the HTN-Maker-based approach, all the intermediate tasks must be given as part of the set of annotated tasks, which requires effort from a human expert.

To generate more interesting hierarchies, we turn to *pattern mining* and propose an approach to find frequent patterns that is inspired by the GoKrimp algorithm [Lam+14], exploiting the MDL principle [Grü07]. This approach generates a set of *candidate patterns* and greedily selects the one that compresses the most the set of sequences. Subsequences covered by the chosen pattern are then replaced with a new symbol, and this process is repeated until the compression stops improving. It is easy to see how this approach can be transposed to our sets of demonstration sequences.

The reasoning behind the use of frequent patterns is that in many domains, an abstract task encodes some sub-activity that is carried out to achieve the higher level task. Obviously, encoding a given activity as a reusable task will be most useful to the domain if it is occurring frequently and in several high level activities. Frequent patterns are therefore a way to find common blocks of activity that occur in a set of demonstrations, which should therefore be good candidates for generating new intermediate abstract tasks.

From a set of demonstrations in which frequent patterns have been abstracted, we then will be able to apply the previously presented HTN-Maker to generate neighbours, which will naturally contain multiple levels of hierarchy.

Background B: *Frequent Pattern Mining*
Using the definition from Aggarwal [Agg14], frequent pattern mining is the problem of finding relationships between items in a dataset, and *sequential* pattern mining is that of finding relationships given an ordering of the items in the dataset. As we are focusing on learning from *sequences* of actions, we will focus on this second part of the definition.

Background C: *The Minimum Description Length Principle*
The Minimum Description Length (MDL) principle [Grü07], comes from information theory and states that learning can be viewed as a form of data compression, as both intend to find some regularity in some source material. Therefore, in this framework, the best model is the one that can compress the data the most. Furthermore, as the total size of the compressed data includes the size of the model itself, used to reconstruct this data, this principle naturally guards against overly specific models.
Example uses of this technique range from learning CFGs [SBS12] (of which HTNs are close) to finding common graph patterns [BCF20].
More details are given in Appendix C.

3.2.2.1 The Considered Patterns

Let us start by describing the patterns that we wish to generate. They are a subset of the valid regular expressions, and are defined using the grammar presented in Figure 3.7. This grammar recognizes two types of patterns:

- Sequences of task symbols with optional quantifiers ($\{?, *, +\}$), such as $\langle ab \rangle$, $\langle a^+ b \rangle$ or $\langle aa^? bc^* \rangle$. These are naturally called *sequence patterns* and are used to abstract repeated

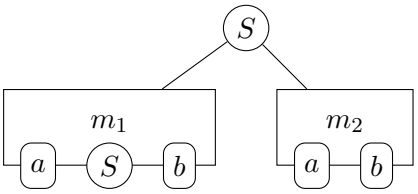
(∗ and + quantifiers) or optional (? quantifier) behaviours. A common example of such behaviour is the "repeat until" pattern.

- Alternations between sequences of task symbols, such as $\langle a|b\rangle$, $\langle(aa)|(bc)\rangle$ or $\langle a|b|(cc)\rangle$. These are called *choice patterns*, generating alternative methods for abstracting over some specific activity.

$$
\begin{aligned}
\langle\texttt{symbol}\rangle &::= \text{any task symbol}\\
\langle\texttt{sym-seq}\rangle &::= \langle\texttt{sym-seq}\rangle\langle\texttt{sym-seq}\rangle \mid \langle\texttt{symbol}\rangle\\
\langle\texttt{choice-base}\rangle &::= \langle\texttt{choice}\rangle \mid \langle\texttt{sym-seq}\rangle\\
\langle\texttt{choice}\rangle &::= \langle\texttt{choice-base}\rangle\texttt{"|"}\langle\texttt{choice-base}\rangle\\
\langle\texttt{star}\rangle &::= \langle\texttt{symbol}\rangle\texttt{"*"}\\
\langle\texttt{plus}\rangle &::= \langle\texttt{symbol}\rangle\texttt{"+"}\\
\langle\texttt{seq-base}\rangle &::= \langle\texttt{symbol}\rangle \mid \langle\texttt{star}\rangle \mid \langle\texttt{plus}\rangle\\
\langle\texttt{seq}\rangle &::= \langle\texttt{seq}\rangle\langle\texttt{seq}\rangle \mid \langle\texttt{seq-base}\rangle\\
\langle\texttt{pattern}\rangle &::= \langle\texttt{seq}\rangle \mid \langle\texttt{choice}\rangle
\end{aligned}
$$

Figure 3.7: Grammar for the generated patterns, using BNF syntax.

Note that an HTN, even without preconditions or parameters, is still more expressive than regular languages (as presented in Figure 3.8), and thus are more expressive than what our patterns can express. However, remember that this pattern extraction is only a single step in a larger process.



(a) HTN structure.

$\langle\texttt{S}\rangle ::= \langle\texttt{a}\rangle\langle\texttt{S}\rangle\langle\texttt{b}\rangle \mid \langle\texttt{a}\rangle\langle\texttt{b}\rangle$

(b) Equivalent CFG grammar.

Figure 3.8: A HTN without preconditions or parameters and an equivalent grammar. Note that this grammar is not regular, as can be proved using the pumping lemma for regular languages.

Note that it is trivial to map from a regular expression to a hierarchical task decomposition, as presented in Figure 3.9.

3.2.2.2 Substituting Patterns in Demonstrations and Extracting Neighbours

Before describing how patterns are built and selected, we need to explain how they can be integrated in the set of demonstrations and how we can extract HTN candidates taking advantage of them.

The choice of regular expressions as patterns has an advantage from an implementation point of view, as it is easy to use regular expression matching to find and substitute patterns in the demonstration set using off-the-shelf software components. This substitution is therefore simply done using the find and replace operation of a regular expression engine, as shown in Figure 3.10. Here, for a pattern $p = \langle a^{+}b\rangle$, every instantiation of the pattern in the demonstration set D

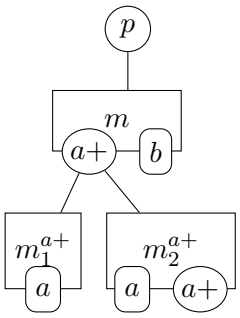


Figure 3.9: Hierarchical representation of the pattern $p = \langle a^+ b \rangle$. Here, $\text{len}(p) = 2$, only counting the a and b symbols.

is then replaced by the symbol corresponding to p. This will allow us to build new patterns iteratively, reusing the previous patterns as building blocks for new ones.

$$D : \left\{ \begin{array}{l} d_1 : \mathbf{a}\ \mathbf{b}\ c\ d \\ d_2 : \mathbf{a}\ \mathbf{b}\ d \\ d_3 : \mathbf{a}\ \mathbf{b}\ c \\ d_4 : d\ \mathbf{a}\ \mathbf{a}\ \mathbf{b} \end{array} \right\} \underset{p}{\mapsto} \left\{ \begin{array}{l} d_1 : \mathbf{p}\ c\ d \\ d_2 : \mathbf{p}\ d \\ d_3 : \mathbf{p}\ c \\ d_4 : d\ \mathbf{p} \end{array} \right\}$$

Figure 3.10: Pattern substitution example for a pattern $p = \langle a^+ b \rangle$.

These patterns can then be used in conjunction with the previously presented HTN-Maker-based operator, applying it on the substituted demonstration set. This allows generating new HTN structures that could not have been generated with only the base demonstrations, as presented in Figure 3.11. Here, we have added a new task p in our hierarchy, whose methods and subtasks have been generated as part of our frequent pattern search.

3.2.2.3 Building a Set of Compressing Patterns

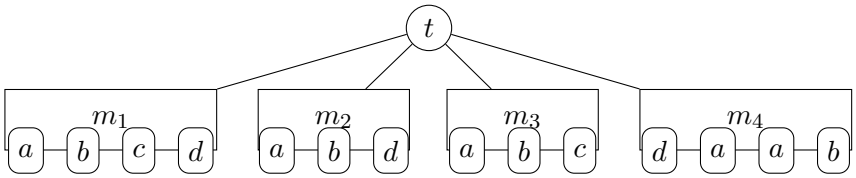
While the first approach that we implemented [HB23] generated a single set of patterns from which to generate neighbours using the HTN-Maker algorithm. However, this relied on an ad-hoc metric of pattern complexity, which was used a proxy for our MDL-based metric (which is itself a proxy for planning performances).

Thanks to improvements in our implementation of the trace matching process used to compute neighbour quality, it became possible to evaluate the quality of a pattern through the quality of the *best* neighbour that can be generated from this pattern.

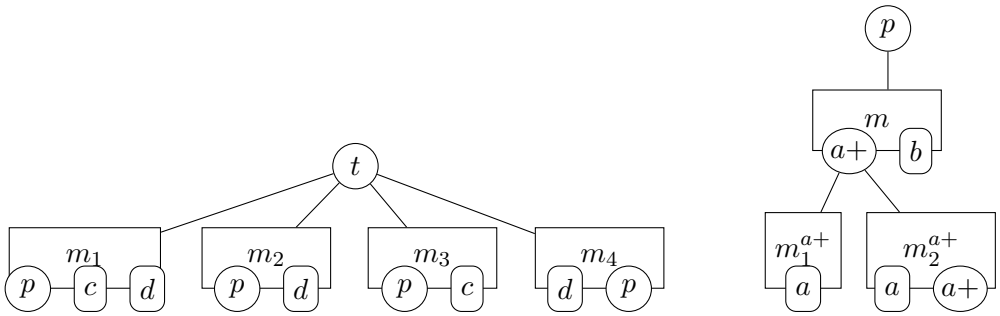
We therefore will present how we can build such a set of candidate patterns to be evaluated.

For a set of demonstrations D, let us write D_i a set of demonstrations abstracted i times, with $D_0 = D$, and with $p_i, i > 0$ the associated pattern. Let us also write $\mathcal{T}_D$ the set of all the primitive and synthetic tasks in D and $\text{win}_k(d), d \in D, k > 2$ the function that generates the set of all sliding windows of size k over a trace d.

The different sets of basic patterns we generate are presented below. Note that every pattern is written as a sequence $\langle \dots \rangle$ to avoid any confusion between multiple patterns and sequential patterns.



(a) A possible HTN using the basic demonstration set.



(b) A possible HTN using the abstracted demonstration set. t and p presented as separate subhierarchies for simplicity.

Figure 3.11: Example of possible HTN structure generation using the HTN-Maker-based operator on a demonstration with the abstracted frequent patterns as shown in Figure 3.10. The learned methods are a trivial lookup of all the demonstrations, but are enough to present the sharing of behaviours afforded by the use of pattern mining.

- $P^b_{D_i}$ is the set of all basic patterns, that is any primitive with a modifier.

$$P^b_{D_i} = \left\{ \langle a^? \rangle, \langle a^* \rangle, \langle a^+ \rangle \mid a \in \mathcal{T}_{D_i} \right\}$$

- $P^{c,k}_{D_i}$ is the set of all choices with k alternatives.

$$P^{c,k}_{D_i} = \left\{ \langle a_1 \mid \cdots \mid a_k \rangle \mid (a_1, \ldots, a_k) \in \mathcal{T}^k_{D_i} \wedge \forall (i,j) \in [1,k]^2, i \neq j \Rightarrow a_i \neq a_j \right\}$$

- $P^{s,l}_{D_i}$ is the set of *sequence patterns* of length $l, l > 2$.

  It is generated from the set of all possible subsequences encountered in the demonstrations, where each element may have a modifier.

$$P^{s,l}_D = \left\{ \begin{array}{l} \langle a'_1, \ldots, a'_k \rangle \mid (a'_1, \ldots, a'_k) \in \prod_{A \in \mathcal{A}} A, \\ \\ \mathcal{A} = \begin{pmatrix} A_1 = \{a_1, a_1^?, a_1^*, a_1^+\} \\ \cdots \\ A_l = \{a_k, a_k^?, a_k^*, a_k^+\} \end{pmatrix}, (a_1, \ldots, a_k) \in \mathrm{win}^d_k, (d,k) \in D_i \times [2,l] \end{array} \right\}$$

3.2.2.4 Searching for the Best Abstraction Set

Let us now turn our attention to the process of finding the best pattern set. Similarly to the GoKrimp algorithm [Lam+14] that we used for inspiration, we use a greedy search algorithm, as described in Algorithm 3.4. In this algorithm, assume that we have access to a function Gen Best Neighbour that we can use to generate the best HTN given a set of abstracted

demonstrations. This function will be detailed at the end of this chapter, and can for now be considered as a subprocedure that uses the previously described HTN-Maker neighbour generation in a greedy local search algorithm to find the best HTN.

Algorithm 3.4 Find Best Patterns(D_i, k, l, H_B)

Input: D_i set of traces, possibly abstracted
k maximal number of choices for a pattern
l maximal length of a sequence pattern
H_B a possibly empty HTN to use as a starting point

$P_D \leftarrow \emptyset$
$H_{P_D} \leftarrow H_B$
$D' \leftarrow D_i$

repeat
  $P_c \leftarrow P^b_{D'} \cup P^{c,k}_{D'} \cup P^{s,l}_{D'}$
  $\mathcal{H}_c \leftarrow \emptyset$
  for all $p \in P_c$ **do**
    $D_H \leftarrow$ Abstract Demos(D', p)
    $H \leftarrow$ Gen Best Neighbour(D_H, H_{P_D})
    $\mathcal{H}_c \leftarrow \mathcal{H}_c \cup \{(H, p)\}$
  $(H_c, p^*) \leftarrow$ Find Best HTN$(\mathcal{H}_c)$
  if Quality(H_{P_D}) < Quality(H_c) **then**
    $H_{P_D} \leftarrow H_c$
    $P_D \leftarrow P_D \cup \{p^*\}$
    $D' \leftarrow$ Abstract Demos(D', p)
until Quality(H_{P_D}) stops improving

return (P_D, H_{P_D})

In the pattern search procedure, we generate, from the current set of demonstrations D', the set of all possible candidate patterns P_c, as described earlier. Then, for each pattern p, we abstract the current set of demonstrations using p, and we try and find the best HTN that can be generated using this new set of demonstrations. The best pattern p^* is then chosen as the one that can generate the best HTN and stored in the set P_D. Once found, we update the current set of demonstrations D', abstracting it using p^*.

This process is repeated until we can no longer generated better HTNs, and we finally return the set of patterns that we built, along with the associated HTN[1].

Note that it is necessary to limit the number of choices k and the length of sequential patterns l so that the problem remains tractable. We can still generate interesting behaviours, especially as more choices can be built hierarchically, with other choices being used as one of the options in further abstraction iterations. Similarly, longer sequences can be generated through multiple abstraction steps, assuming that the subsequence has been selected in a previous step.

3.2.3 Simplifying a Candidate Structure

During the search procedure, it is entirely possible that we end up with complex structures that can produce the same decompositions as simpler ones. The needlessly complex structures that we consider are as follows:

[1]Returning the best HTN is simply a convenience for later in this document.

1. Duplicate tasks with identical decompositions[1].
2. Tasks decomposing into a single method with a single subtask.
3. Duplicate methods.
4. Tasks with only empty methods.

We therefore introduce an HTN simplification operator that handles all these cases, with an application presented in Example 3.4. It merges duplicates tasks into a single one, collapses single tasks methods into their parent method, deduplicates methods of a given task and removes tasks with only empty methods.

Example 3.4 *Example of HTN simplification.*
In this example, each simplification case is presented:

1. t_{dup} and t'_{dup} are deduplicated as a single task t.
2. t_{single} is collapsed into its parent method.
3. m^1_{dup} and m^2_{dup} are deduplicated by keeping only one of them.
4. $t_\emptyset$ is removed.

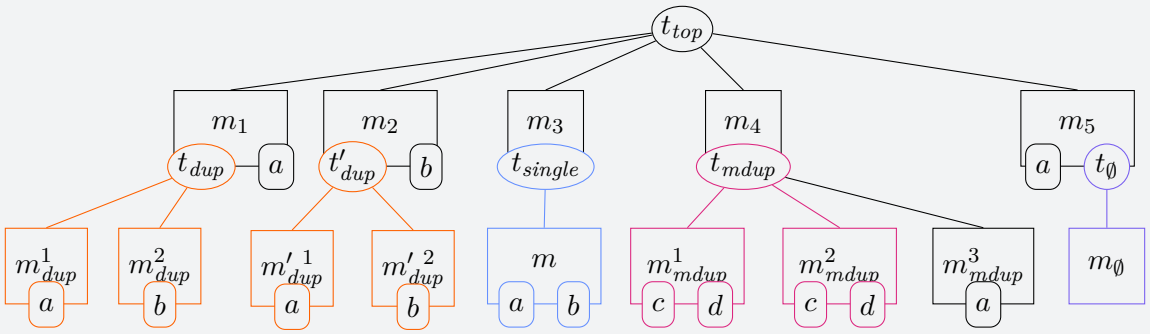


Figure 3.12: HTN before simplification.

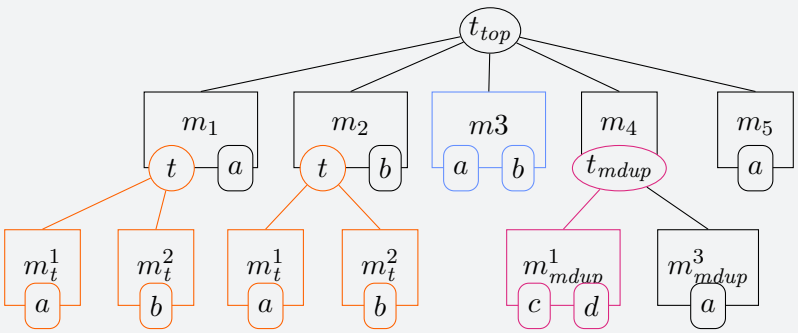


Figure 3.13: HTN after simplification.

It may be argued that empty method tasks could be used to verify that a given set of preconditions hold, as is done for example in HTN-Maker [HMK08] with the introduction of

[1]This first case should not happen with the currently presented algorithm, however it has been considered as it is easy to introduce with random modifications of the structure, for example.

verification tasks. However, because we do not learn parameter at this point in our procedure, this kind of tasks cannot be considered as such. Furthermore, if pre- and post-conditions are learned for some tasks later, this kind of validation tasks can trivially be added as a post-processing step.

3.3 Evaluating Candidate Models

While we have presented the process allowing to generate candidate domain structures and hinted at a process to extract their parameters, we need to address the evaluation of the *quality* of a domain. An obvious solution would be to attempt planning with our candidate domains, in order to find domains that can solve all the problems in a representative test set in the most efficient manner. However, this approach would be too costly to be integrated in the global pipeline: even relatively simple real world problems can require tens or hundreds of seconds to be solved, even with good handcrafted domains. Therefore, each iteration with its hundreds of neighbours would require thousands of seconds to run, and even with techniques to focus the search on relevant parts of the space of possible HTN structures, a domain would not be found in a reasonable time.

Therefore, we need to determine a surrogate metric that will favour general and efficient planning domains. We can have the intuition that such domains will need to be able to generalize to new unseen instances while constraining the possibilities to guide an automated planner's search. As we are working from positive examples only, we took insight from the learning of grammars, and decided to add a simplicity bias. Indeed, our issue is similar, as we can equate the search constraining properties of our domain with limiting incorrect parses in a learned grammar

3.3.1 The MDL Principle for HTN Structures Evaluation

We decided to frame this problem in the context of the Minimum Description Length (MDL) principle[1], as has been done for grammar learning. Note that we use *crude* MDL with an arbitrary two part code. With this vision, we can say that a desirable domain should be able to "compress" the demonstrations, i.e. the domain is able to reproduce the presented demonstrations with a minimal search effort. However, if we only considered this part as our surrogate metric, we would end up with overfitting domains. In the worst case scenario, the domain would end up similar to a lookup table, similar to the one presented in Figure 2.13c, which would obviously have a very poor generalizability. Fortunately, as explained earlier, the MDL principle takes into account the size of the domain in addition to its capability to generate the source data. This will limit the complexity of the domain, and therefore push it towards regions of the space where common patterns are abstracted.

Assuming we have generated a domain candidate H_c as part of our search process, we define:

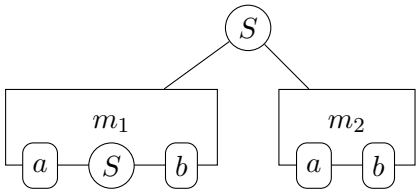
- The description length of the model $L_{\text{mod}}(H_c)$. This represents the size of the HTN, and will increase as we add new methods and tasks to the model.
- The description length of the demonstrations in D knowing H_c $L_{\text{dem}}(D \mid H_c)$. This represents the difficulty of reconstructing the demonstration traces using H_c.

[1] Overview of the MDL principle given in Appendix C.

In order to compute the global description length of the domain and the demonstration set, we combine these two metrics, using a factor α to balance their relative importance, as defined in Equation 3.7.

$$L(H_c, D) = \alpha L_{\text{mod}}(H_c) + L_{\text{dem}}(D \mid H_c) \tag{3.7}$$

The Model Description Length To compute $L_{\text{mod}}(H_c)$, we will use information theory to compute a bound on the length of an optimal encoding of our domain. We propose an encoding of an HTN structure as a set of grammar rules, using a symbol for each task (primitive and abstract), using the symbol | to separate methods and ; to mark the end of the decomposition of task. An example is shown in Figure 3.14. From this, we will be able to compute the frequency of each symbol in the alphabet used to describe these rules in order to extract bounds on the length of a codeword used in the optimal encoding of these rules. This codeword length will be used to compute the global domain description length.



(a) HTN structure.

$S \rightarrow aSb \mid ab$;

(b) Equivalent grammar rule set.

Figure 3.14: An HTN structure and the equivalent set of grammar rules.

Considering a random variable X_{H_c} that takes as possible values the symbols in our alphabet, information theory tells us that the entropy $H(X_{H_c})$ of this variable bounds the expected codeword length L of our optimal code as in the equation below:

$$H(X_{H_c}) \leq \mathbb{E}(L) \leq H(X_{H_c}) + 1 \tag{3.8}$$

Noting $x_1, \ldots, x_n$ the symbols of our alphabet, and their occurrence probability $P(x_1), \ldots, P(x_n)$, the entropy formula is:

$$H(X_{H_c}) = -\sum_{i=1}^{n} P(x_i) \log(P(x_i)) \tag{3.9}$$

As we know the frequency of each symbol in our domain as well as their total number, we can therefore compute $P(x_i)$ and thus compute a bound on the optimal domain encoding length. Because we only want to use this metric to compare two domains, we arbitrarily choose to use the lower bound as the model description length. Considering that a message for describing our given HTN is composed of k symbols, then we have:

$$L_{\text{mod}}(H_c) = k \times H(X_{H_c}) \tag{3.10}$$

The Demonstration Set Description Length To compute $L_{\text{dem}}(D \mid H_c)$, we need to evaluate the cost of encoding a trace $d \in D$ based on our HTN domain. We assume that we have access to a decomposition tree that allows mapping a demonstration can be mapped to a sequence of refinements from the original task to the primitive sequence. Each refinement

of a task constitutes a choice point at which the engine must select one method among the applicable ones in the current state.

We define $L_{\text{dem}}(D \mid H_c)$ as the encoding size of all choice points (Equation 3.11), where $\mathcal{C}$ is the set of choices one has to make to reconstruct d from H_c, and $M_{\text{app},cp}$ is the set of applicable methods at a choice point cp.

$$L_{\text{dem}}(D|h) = \sum_{d \in D} \sum_{cp \in \mathcal{C}} \log\left(|M_{\text{app},cp}|\right) \tag{3.11}$$

Because we actually do not want to compress a dataset, but rather find a good HTN model, we will make several modifications to the L_{dem} term based on our practical observations, presented in Equation 3.12. The $\frac{1}{|D|}$ and $\frac{1}{\text{len}(d)}$ factors are used to be more robust to the length of the demonstrations and their number, while the removal of the logarithm tends to favour models with less complex method choices, leading to simpler planning models in practice.

$$L_{\text{dem}}(D|h) = \frac{1}{|D|} \sum_{d \in D} \frac{1}{\text{len}(d)} \sum_{cp \in \mathcal{C}} |M_{\text{app},cp}| \tag{3.12}$$

The consideration of the applicable methods at each choice point instead of all the possible ones allows to consider the pruning of the search tree imposed by the pre-conditions during the use of the domain, which is mainly relevant if we want to use this metric in a more general setting rather than simply learning HTN *structures*, which do not have any pre-conditions by definition.

Let us now present an example of the computation of this description length for the HTNs presented in the introductory chapter, recalled here in Figure 3.17 for simplicity.

Example 3.5 *Computing the description length of an HTN and associated demonstration set.*
Considering the HTN presented in Figure 3.17b decomposing a task t, we can view this domain as a grammar-like structure, which we can describe with the following rule:

$$t : at \mid bt \mid ct \mid dt \mid ;$$

The frequency table of each symbol is given in the table in figure 3.15a, giving us the entropy $H(X_{H_c})$. Here, $H(X_{H_c}) = 2.40$ bits, which means that to encode our model, each symbol in an optimized encoding contains on average this quantity of information. This is mainly influenced by the total number of symbols, as well as the number of unique symbols in our domain.
Given that there are fifteen symbol occurrences in the rule, this bounds the optimal value of $L_{\text{mod}}(H_c)$ as shown in the equation in Figure 3.15b. As this value will only be used for comparison, we arbitrarily choose the lower bound as L_{mod}.

Symbol	t	a	b	c	d	|	;	Total
Frequency	5	1	1	1	1	4	1	14

(a) Frequency table.

$$
\begin{aligned}
H(X_{H_c}) &= -\left[5\left(\frac{1}{14}\log\left(\frac{1}{14}\right)\right) + \frac{4}{14}\log\left(\frac{4}{14}\right) + \frac{5}{14}\log\left(\frac{5}{14}\right)\right] \\
&= 2.40 \text{ bits} \\
L_{\text{mod}}(H_c) &\in [14 \times 2.40, 14 \times 3.40] = [33.6, 47.6] \text{ bits}
\end{aligned}
$$

(b) Entropy and domain length bounds.

Figure 3.16: Model length calculation for the domain presented in Figure 3.17b.

As an example, we will study the set of demonstration presented in Figure 3.17a using again the domain of Figure 3.17b. For each of the sequences in the demonstration set, we know that we have to choose three times among five methods, therefore:

$$L_{\text{dem}} = \frac{1}{2}\left(\frac{1}{3} \times 4 \times 5 + \frac{1}{3} \times 4 \times 5\right) = 6.67 \tag{3.13}$$

To show how this metric can be used, Table 3.1 presents the different values computed for the examples presented in Figure 3.17, for two different values of α

			$L(H_c, D)$	
H_c	$L_{\text{mod}}(H_c)$	$L_{\text{dem}}(D\|H_c)$	$\alpha = 1$	$\alpha = 0.1$
Fig. 3.17b	33.6	6.67	40.27	10.03
Fig. 3.17c	24.57	0.67	25.24	3.13
Fig. 3.17d	24.57	4.5	29.07	6.96
Fig. 3.17e	29.2	1	30.2	3.92

Table 3.1: Description length for the examples presented in Figure 3.17.

We can observe that the best model is always the "lookup" one. This is due to the fact that it is a toy example with a very simple set of demonstrations, and this behaviour does not show on real domains with a more complex structure. Observe however that when considering the other domains, changing α changes which model may be considered the best by changing the focus from the size of the model to the efficiency with which we can reconstruct the demonstrations.

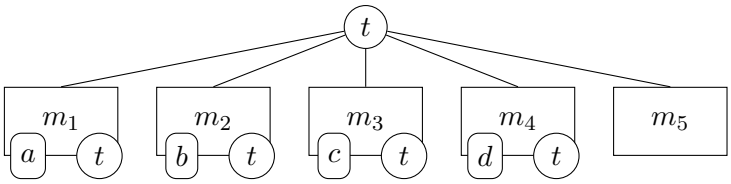
In the example presented previously, we can observe that the size of the domain will be highly correlated with the complexity of the target behaviour. However, the length of the demonstration will be less influenced by the domain complexity. In order to alleviate this problem, we propose to define the *normalized description length* of an HTN, where for a given reference model H_{ref}:

$$
\begin{aligned}
L^{norm}(H_c, D) &= \alpha L^{norm}_{\text{mod}}(H_c) + L^{norm}_{\text{dem}}(D \mid H_c) \\
&= \alpha \frac{L_{\text{mod}}(H_c)}{L_{\text{mod}}(H_{ref})} + \frac{L_{\text{dem}}(D \mid H_c)}{L_{\text{dem}}(D \mid H_{ref})}
\end{aligned} \tag{3.14}
$$

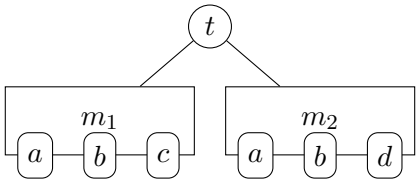
This normalized description length can be interpreted as allowing to improve the quality of one term in some proportion as long as the other term is not worsened by a larger proportion,

$$t \to \langle a, b, c \rangle \qquad t \to \langle a, b, d \rangle$$

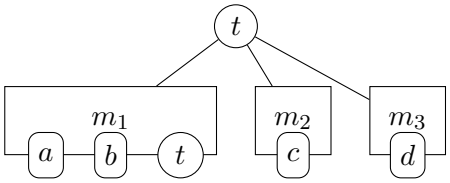
(a) Available demonstrations, showing that t was once achieved with the $\langle a, b, c \rangle$ action sequence and once with the $\langle a, b, d \rangle$ action sequence. Intermediate states as we are considering HTN structures only.



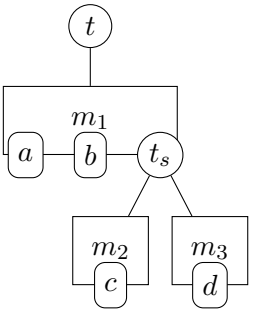
(b) Generic domain where the planner might pick any of the primitive actions and rely on the recursive call to t to continue if needed.



(c) Domain where each demonstration is fully encoded into a dedicated method.



(d) Intermediate domain the common $\langle a, b \rangle$ sequence is grouped. It relies on the recursive call to t in m_1 to produce a full sequence.



(e) Domain where the $\langle a, b \rangle$ sequence is shared, requiring a new abstract task t_s

Figure 3.17: Illustration of the possible structures of the learned domain for a simple learning task with two demonstration of how to perform a task t.

modulated by the α factor. A good reference model is the fully recursive model (as presented in Figure 3.17b), as it can be trivially constructed for any set of primitives and is intrinsically complete.

3.3.2 Obtaining Decomposition Trees from HTN Structures and Action Sequences.

As stated earlier, we need to know the choices that are required while using a candidate domain H_c in order to generate a given demonstration trace d. This means that we need to know the decomposition tree that a planner would have generated using our domain and planning for the task demonstrated in the trace. To this end, we use the technique presented by Höller et al. [Höl+21] for plan verification. With plan verification, we share a common goal: we first need to know if a sequence of actions (the demonstration) is a solution to a given HTN planning problem (the currently considered candidate domain). However, we additionally need to know the full decomposition tree.

Let us briefly present the technique mentioned previously [Höl+21] where the authors propose to compile the plan verification problem to a HTN planning problem by modifying the primitive actions present in the original planning domain. Because we are only concerned with the *structure* of HTNs, we will not focus on the modifications pertaining to pre-conditions and effects, and instead redirect the reader to the original paper for this information. In this paragraph, let us consider a demonstration $d = \langle a_0, a_1, \dots, a_n \rangle$ and an HTN structure $H = (\mathcal{L}, T_P, T_C, M)$ to match with d.

In order to determine which action is to be executed at any given point in the decomposition process, the compilation defines a logic $\mathcal{L}'$ by adding new state predicate symbols $f_0, f_1, \dots f_n, f_{n+1}$ to $\mathcal{L}$. We then define a planning problem $\mathcal{H} = (\mathcal{L}', T_P^{obs}, T_C', M')$:

- T_P^{obs} is the new set of primitive tasks, and is composed of *technical actions*. One technical action a_i^{obs} is generated for each action $a_i \in d, i \in [0, n]$. Their effects and preconditions are defined as
$$\text{pre}(a_i^{obs}) = \{f_i\}$$
$$\text{eff}(a_i^{obs}) = \{\neg f_i, f_{i+1}\}$$
We also add a technical action $obs_\perp$ to handle the case where a primitive task was not used, defined as
$$\text{pre}(obs_\perp) = \{\perp\}$$
$$\text{eff}(obs_\perp) = \emptyset$$

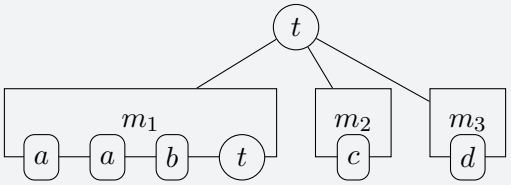
- T_C' is the new set of compound tasks, created by adding one new abstract task a^{cmp} for each primitive $a \in T_P$ and adding them to T_C

- M' the set of methods is generated by modifying all the methods in $m \in M$ so that:
  - Every decomposition into a subtask $a \in T_P$ is replaced by the corresponding a^{cmp}.
  - For each technical action a_i^{obs}, adding a method that decomposes the corresponding task a^{cmp} into a_i^{obs}.
  - If a primitive $a \in T_P$ does not have any technical action associated with it, add a method that decomposes the corresponding task a^{cmp} into $obs_\perp$.

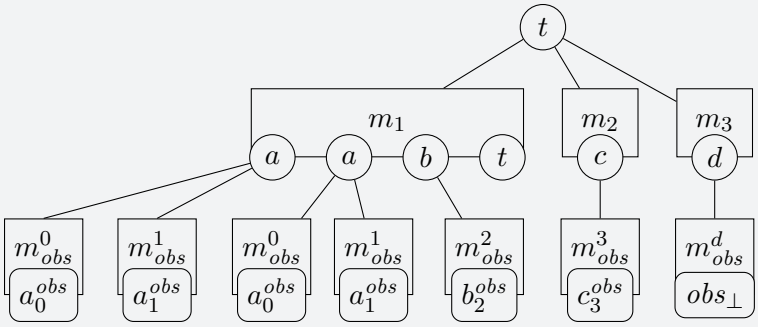
We then define a corresponding planning problem $\mathcal{P} = (\mathcal{H}, s_0, g)$, with $s_0 = \{f_0\}$ and $g = \{f_{n+1}\}$. The goal is necessary to ensure that the solution to the planning problem reproduces the whole demonstration.

We can then obtain the decomposition tree by using an off-the-shelf planner, adding a simple post-processing step to undo the changes pertaining to technical actions and recover the choices that would have been made for planning originally.

Example 3.6 *HTN structure matching using plan verification techniques.*
Let us consider a demonstration $d = \langle a, a, b, c\rangle$ of a task t and a candidate HTN structure as presented in Figure 3.18a. Rewriting the demonstration in terms of indexed observations, we have $d = \langle a_0^{obs}, a_1^{obs}, b_2^{obs}, c_3^{obs}\rangle$, and we can build the HTN with technical actions corresponding to our candidate as presented in Figure 3.18b, with the associated preconditions, effects presented in Figure 3.18c. This last figure also presents the initial and goal state of the planning problem corresponding to our demonstration matching problem.



(a) Base HTN structure candidate.



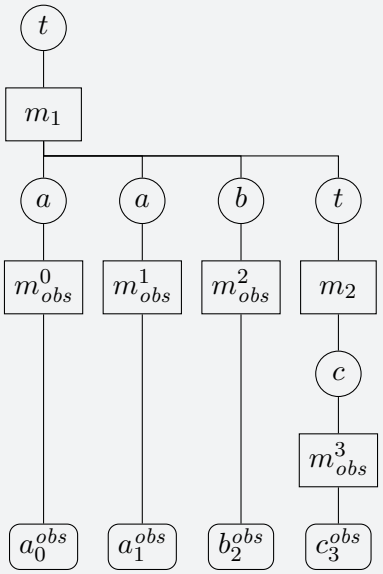
(b) Corresponding HTN structure used in the plan verification problem.

$$s_0 = \{f_0\}$$
$$g = \{f_4\}$$
$$\left\{\begin{array}{ll} \text{pre}(a_0^{obs}) = \{f_0\} & \text{eff}(a_0^{obs}) = \{\neg f_0, f_1\} \\ \text{pre}(a_1^{obs}) = \{f_1\} & \text{eff}(a_1^{obs}) = \{\neg f_1, f_2\} \\ \text{pre}(b_2^{obs}) = \{f_2\} & \text{eff}(b_2^{obs}) = \{\neg f_2, f_3\} \\ \text{pre}(c_3^{obs}) = \{f_3\} & \text{eff}(c_3^{obs}) = \{\neg f_3, f_4\} \end{array}\right\}$$

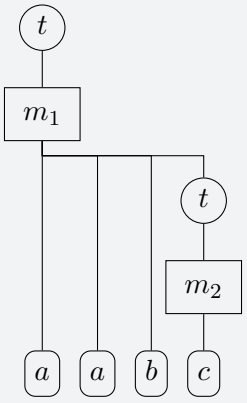
(c) Initial (s_0) and goal (g) states with preconditions of methods and effects of actions in the plan verification HTN.

Figure 3.19: Base HTN and the associated one for plan verification.

Solving this planning problem, we obtain the decomposition tree presented in Figure 3.20a, which can be trivially mapped to a decomposition tree corresponding to the original HTN, as shown in Figure 3.20b. In this decomposition tree, we would have two choice points as described in the previous section, namely where m_1 and m_2 are selected among the applicable methods. In the plan verification HTN, we would have six of these choice points.



(a) Decomposition tree for the plan verification.



(b) Corresponding decomposition tree reverting to the original HTN.

Figure 3.21: Base HTN and the associated one for plan verification.

Note that in this setting, the choices made by the planner during the matching planning process may not be optimal in terms of number of choices. Considering the optimal decomposition tree, the computed demonstration length will be the true value of $L_{\text{dem}}(D \mid H_c)$, while a non-optimal tree one may lead to an overestimation of this value.

A decomposition with a minimal number of choices could easily be obtained using an optimal planner [BHB19; Bit23] by adding fake actions in each method without any preconditions or effects, and optimizing for the solution length. Considering that some choices may be "easier" than others, as they may choose among a smaller set of methods, the number of fake actions for

each method may even be proportional to the choice complexity. However, in practical cases, using satisficing planners is considerably faster than optimal ones. We therefore decided to use a satisficing planner with an option for best-effort plan optimization [Sch21] to obtain reasonable decomposition trees.

Furthermore, parsing based approaches [Li+14; PB23], could help obtain better decompositions, considering the problem of finding an optimal decomposition as that of finding the most probable parse tree. A recent approach [PB23] has shown that for plan recognition, parsing may be faster than using a planning based approach. These results hold particularly well when the unknown suffix of the plan is short, which aligns well with our use case, where it is empty because we are working with complete plans.

3.4 The Complete Structure Search Algorithm

Let us now present the whole structure search algorithm.

Our early work [HB23] showed that a basic greedy hill-climbing search relying only on the HTN-Maker-based candidate generation, a set of frequent patterns extracted using an ad-hoc metric and a simplification procedure to remove unused methods is able to learn good planning models on simple domains from the IPC, requiring only the evaluation of a few thousands of candidates. However, this same work showed that these results were conditioned on the quality of the mined frequent patterns, which requires non-negligible effort from a human expert. Simply adding some random mutation operators to the process obviously did not improve the results, due to the greedy nature of the original algorithm, and moving to stochastic hill-climbing did not either.

However, in this same work we noted that learning parameters during the search process did not improve the results (due to the limited pre-conditions that were extracted), while it incurred a severe computational cost. Removing this step allowed us to consider evaluating a much larger set of possible planning domains, and finding frequent patterns without relying on another surrogate metric, as described earlier.

We then obtained vastly improved results simply finding this set of patterns and keeping as a result the final HTN associated with the best pattern. In the section on pattern finding, we used a black-box function Gen Best Neighbour(D, H) that returned the best HTN for a given set of demonstrations. This function is actually implemented as described in Algorithm 3.5. Note that this procedure should not be too expensive to call compared to the number of patterns to evaluate, as it will be called once for every considered patterns[1].

Here, we generate a set of neighbours using the HTN-Maker algorithm, which we then simplify in two steps:

1. Remove all the methods that are actually unused when matching the traces.

2. Simplify the HTNs using the structure simplification operator presented earlier.

Remember that unused methods may be introduced by our simplified version of the HTN-Maker as detailed in the dedicated section of this chapter. Detecting these methods is straightforward using the decompositions trees generated for the evaluation of the quality of the different HTNs, removing any method that never appears in the decomposition tree.

[1] In practice, because some patterns may be generated several times, our implementation makes heavy use of caching when computing the decomposition trees for evaluating the quality of a given HTN. This caching is also helpful when learning HTNs with several top-level tasks, as some modifications may only affect one task and not the others.

Algorithm 3.5 GEN BEST NEIGHBOUR(H, D)

Input: H initial (possibly empty) HTN
D set of demonstration traces

$H^* \leftarrow H$
while QUALITY(H^*) improves **do**
  $\mathcal{H} \leftarrow$ GENERATE NEIGHBOURS HTN-MAKER(H_c, D)
  $\mathcal{H} \leftarrow \{$REMOVE UNUSED METHODS$(H') \mid H' \in \mathcal{H}\}$
  $\mathcal{H} \leftarrow \{$SIMPLIFY STRUCTURE$(H') \mid H' \in \mathcal{H}\}$
  $H_c \leftarrow$ FIND BEST HTN$(\mathcal{H})$
  if QUALITY$(H^*) <$ QUALITY(H_c) **then**
    $H^* \leftarrow H_c$
return H_c

We can then easily write the global HTN structure search algorithm as in Algorithm 3.6. Note that this algorithm adds a run of a GEN BEST NEIGHBOUR* function as a last step, which is a function that is similar to the GEN BEST NEIGHBOUR, as described earlier, but may use more costly operations, as it only has to be run once. In the currently presented version, this optional additional step is not used, as the tried options did not provide better models.

Algorithm 3.6 FIND BEST STRUCTURE(H, D, k, l)

Input: H initial (possibly empty) HTN
D set of demonstration traces
k maximal number of choices for a pattern
l maximal length of a sequence pattern

$(P_D, H^*) \leftarrow$ FIND BEST PATTERNS(D, k, l, H)
optional
  $D' \leftarrow$ ABSTRACT DEMOS(D, P_D)
  $H_c \leftarrow D' \leftarrow$ GEN BEST NEIGHBOUR$(D \cup D', H^*)$
  if QUALITY$(H^*) <$ QUALITY(H_c) **then**
    $H^* \leftarrow H_c$
return H_*

3.5 Conclusion

In this chapter, we presented a system for learning HTN structures from demonstration traces of action symbols using a greedy search algorithm.

As part of this system, we presented how to adapt the HTN-MAKER algorithm [HMK08] to demonstrations without parameters, and analysed the complexity of generating sets of candidate HTNs using this method. Highlighting the limitations of this approach for generating HTN structures, we proposed an approach leveraging frequent pattern mining for generating multi-level hierarchies using the previously presented operator. We also proposed a procedure to simplify an HTN to an equivalent one by removing redundancies and useless hierarchical features.

In order to evaluate the quality of a given HTN during the search procedure, we designed an MDL-based metric as a computationally efficient proxy.

We finally detailed a complete procedure for generating HTNs combining these different

components. However, when detailing this procedure, we presented an optional search step, allowing for more expensive operations. While it is not used in the implemented version due to unsatisfying results, we tried leveraging it, both through an implementation of a stochastic hill-climbing algorithm and a population-based evolutionary algorithm. The rationale behind this would be to add randomness into our search, in the hope of escaping local optima, even though we did not obtain better results, even at the cost of vastly longer learning times. However, given the success of evolutionary approaches for learning tree-like structures, as presented in the introductory chapter, we expect it to be a promising avenue for improving our search algorithm.

Another promising avenue concerns the pattern generation. While the current approach limits itself to patterns expressible as regular expressions, this is mainly due to the simplicity of using find and replace operations with regular expressions. However, using more complex systems for these find and replace operations, it would be possible to use non-regular patterns, such as the "balanced parenthesis" pattern, which would allow generating new hierarchical structures not reachable in the current implementation.

Lastly, we now generate alternation patterns by enumerating all possible combinations, which obviously quickly becomes costly. An option for generating candidates for choices in a smarter way would be to use the · (any) regular expression operator in other sequential patterns, using the different subsequences matched by the · part of the pattern as elements of an alternation pattern.

Chapter 4

Learning Hierarchical Task Networks Parameters from Demonstrations

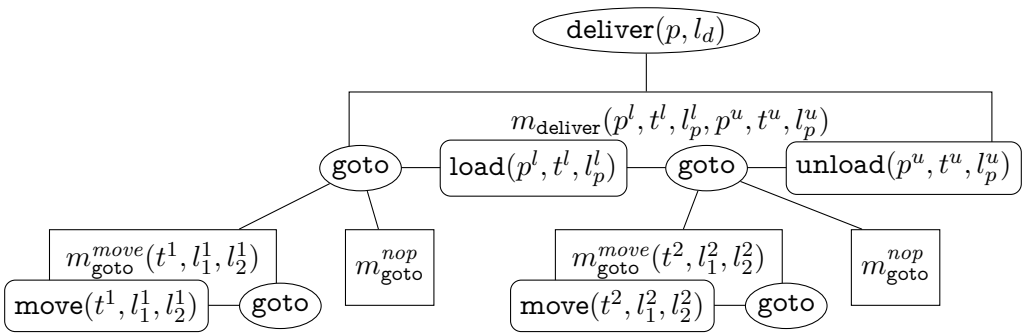
Contents

4.1 Introduction . . . 65
4.2 Learning Symbolic Parameters: Existing Approaches . . . 67
4.3 Generating Correct and Usable Parameters for a Given HTN Structure . . . 68
4.3.1 Generating a Finite Candidate Parameter Set . . . 69
4.3.1.1 Algorithm for Generating a Candidate Parameter Set . . . 69
4.3.1.2 Candidate Parameter Set Generation as Walks on a Graph . . . 73
4.3.2 Simplifying the Generated Candidate Sets . . . 74
4.3.2.1 Parameter Unification . . . 74
4.3.2.2 Parameter Removal . . . 77
4.4 Handling Recursive Task Definitions: the "*Loop-Until*" Pattern . . . 79
4.4.1 Ensuring Consistency with the HTN Structure . . . 84
4.4.2 Ensuring Compatibility with the Demonstrations . . . 84
4.4.3 Minimizing Method Parameters Through Unification . . . 85
4.5 Conclusion . . . 86

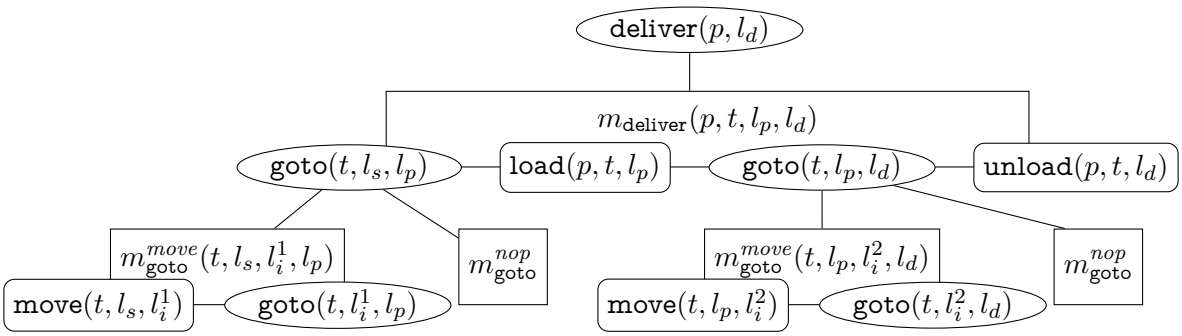
4.1 Introduction

As we have highlighted in our survey of the HTN learning systems, approaches that introduce new abstract tasks are either limited to grounded HTNs [Che+21; HS16; Li+14], have strong limitations on the structure of the learned tasks [GMK18; LJ16] or have scalability issues with the size or structure of the domain [SPF17]. However, lifted models are necessary to generalize the learned knowledge to new objects, and therefore we want to learn parameters for the newly added tasks and methods. The goal of our parameter learner will be to capture the relationship between the arguments of the subtasks of the hierarchy, both vertically (across levels) and horizontally (between siblings tasks in a single method), in an idea similar to *generalization* as described by Plotkin [Plo70]. This will in turn enable a solver to efficiently use the model for planning, and may even be used to extract useful preconditions.

Note that because we consider that the primitive tasks and the demonstration tasks are known, a domain with a trivial parameterization can always be generated from an HTN structure as described in the previous chapter. An example of the difference between an HTN with such trivial parameterization and one with ideally learned parameters is presented in Figure 4.1. Note that for the sake of conciseness, whenever the context is unambiguous in this chapter, we will use "HTN without parameters" or "non-parameterized HTN" instead of "HTN without *learned* parameters".



(a) HTN without learned parameters. The only parameters present are the ones in the primitive tasks and in the demonstrated top-level tasks, as well as parameters trivially added to the methods to stay consistent with our earlier definitions.



(b) HTN with ideal parameters. Note how these parameters transfer information both horizontally and vertically in the hierarchy.

Figure 4.1: An example of HTN without learned parameters and the same HTN with ideal parameters.

In this chapter, we consider as input an incomplete HTN model $\mathcal{H}_S$ and a set of demonstrations D:

- $\mathcal{H}_S$ is a tuple $(\mathcal{L}, T_I \cup T_L, T_P, M_I \cup M_L)$, similar to a hierarchical planning domain.
  - $\mathcal{L}$ is the associated logic.
  - T_P is the set of (known) primitive tasks, with their parameters, preconditions and effects.
  - $T_I \cup T_L$ is the set of abstract tasks, with T_I the set of initially known tasks and T_L the set of learned tasks. The tasks in T_I are complete (known parameters and post-conditions if any), while only the names of the tasks in T_L are known. For clarity, we will call the additional tasks in T_L *synthetic tasks*.
  - $M_I \cup M_L$ is the set of methods. Similarly, M_I is a (potentially empty) set of methods to decompose T_I, and both their pre-conditions and parameters are fixed. On the other hand, for the methods in M_L, only their parent task and their subtasks are known, with the parameters to be determined
- D is a set of solution traces with associated top-level task instance, as defined in the overview section of this manuscript.

Furthermore, we also consider that for each demonstration $d \in D$, we can obtain a decomposition tree that shows how the HTN structure H_S can match the demonstration d, ignoring any parameters. The preceding chapter detailed in Section 3.3.2 how such decomposition trees can be obtained from the input data. For simplicity in this chapter, we will also use args$(\cdot)$ to designate the set of parameters of a task or method.

This chapter will be structured as follows: first, we will detail how other HTN-learning approaches deal with this issue, and present some other related approaches. Then, we will present a first approach to learn the parameters before addressing its limitations in the face of recursive tasks. We will therefore propose a solution to deal with these issues in the case of a specific but frequent recursion pattern.

For simplicity, we will consider that the structure of the HTN model is non-recursive in the demonstrated tasks, except for HTN-Maker-induced direct recursions.

4.2 Learning Symbolic Parameters: Existing Approaches

In previous HTN learning approaches, parameter learning remains an unaddressed issue.

In the case of grounded HTN learning [Che+21; HS16; Li+14], it is obviously considered a non-issue, at the cost of a loss of generalization power. Similarly, when all the possible intermediate abstract tasks are given as input [HMK08; ZMY14], parameters are obviously already given for the abstract tasks, and only their unification with the method parameters is to be found. HTNLearn [ZMY14] uses a very simple form of anti-unification[1], mainly targeted towards finding method pre-conditions, replacing constants by variables and removing duplicate predicates. HTN-Maker [Hog11] uses unification to map method parameters with task and precondition parameters.

The approach by [SPF17] does not require intermediate abstract tasks as input, propagating parameters from the primitives, but does not study how well this approach scales in larger domains nor how to handle recursive task definitions. On the other hand, approaches that

[1] The process of generalizing two symbolic expressions.

learn goal-equivalent tasks [GMK18; LJ16] have their parameters intrinsically defined by their definitions of the learned tasks and methods, making the problem similar to when abstract tasks are given.

In the context of learning lifted Linear Temporal Logic (LTL) formulas from natural language [Hsi+22], assuming a *contextual query* similar to a top level abstract tasks in our context, the anti-unification is done through a simple lifting procedure similar to that in HTNLearn [ZMY14].

In our case, because our new abstract tasks may represent arbitrary behaviours, we need to both learn the structure of the parameters of the tasks and methods, and how they can be unified together to avoid tasks with hundreds of parameters. It can be noted that all these approaches (ours included) are domain specific. This issue plagues generalization, which lacks a common formal framework, as highlighted in a recent survey [CK23]. Classifying these approaches and ours in the author's proposed framework could lead to more efficient methods. However, the inherent structural variadicity and interdependence of our symbols' parameterizations makes this a non-trivial task.

4.3 Generating Correct and Usable Parameters for a Given HTN Structure

The two main roles of parameters in a HTN can be seen as:

- Passing values from the top down to the bottom of the hierarchy.
- Restrict the possibilities of instantiation across siblings, in particular across subtasks of a given method.

One could argue that a third property should be added: the parameters of a method should allow for the extraction of useful pre-conditions. However, this is out of the scope of our current approach and is left for future works.

Of course, the parameterization should remain simple enough so that an automated planner does not run into a non-tractable combinatorial explosion of method instantiations.

These high-level properties led us to propose a two-step algorithm for parameter learning:

1. Identify the set of candidate parameters for all non-parameterized abstract tasks and methods in the domain.
2. Simplify this set of parameters. This step is itself subdivided in two sub-steps:
   (a) Extract possible unifications of candidate parameters from the usage patterns of the hierarchy.
   (b) Remove useless parameters from the hierarchy.

The first step allows finding parameters that *can* propagate information in the hierarchy, especially vertically. However, this may lead to numerous parameters, which contradicts our second property, hence the second step. The unification procedure will also provide the additional benefit of propagating information horizontally across sibling tasks.

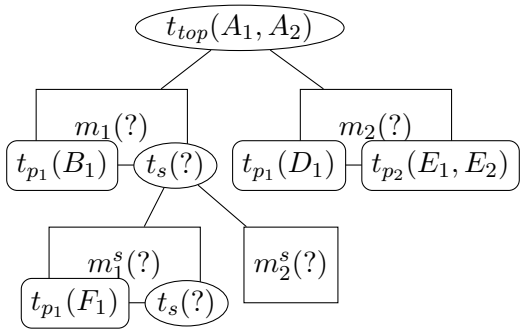
4.3.1 Generating a Finite Candidate Parameter Set

To identify the superset of possible parameters, we define the following properties to determine this set:

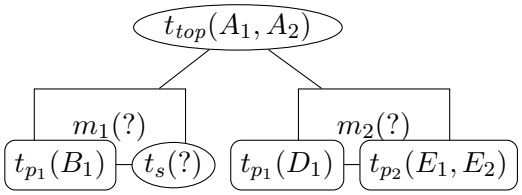
- The set of parameters of a method m must contain the parameters of all its subtasks and must be finite. This comes naturally from the definition of an HTN method where the parameters of each task are imposed.
- Each parameter of a synthetic task t must be used in at least one of its methods. This property is used to ensure that all the parameters of t are actually useful to pass information down the hierarchy.

4.3.1.1 Algorithm for Generating a Candidate Parameter Set

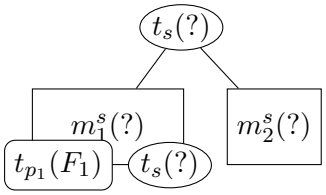
In order to extract a set satisfying these conditions, we propose to reuse the idea of propagating arguments upwards from the primitives presented by Segura-Muros, Pérez and Fernández-Olivares [SPF17]. However, this approach clearly requires defining a single top-level task for the whole HTN, which is difficult, if not impossible, in the presence of recursive task definitions. To solve this issue, we propose to split our HTN into *subhierarchies*, which are HTN-like structures limited to a top-level task and its associated methods. This equivalence between HTN and subhierarchies is presented graphically in Figure 4.2, and is defined more formally in Definition 4.1.



(a) Example task hierarchy, showing how t_{top} can be decomposed using the methods m_1 and m_2.



(b) Subhierarchy for t_{top}.



(c) Subhierarchy for t_s.

Figure 4.2: Example HTN structure and corresponding subhierarchies. Here, t_{top} is a demonstrated abstract task and t_s is a learned abstract tasks, while the other tasks are primitive. This example presents the case where we have an incomplete HTN domain where only the primitive and demonstrated abstract tasks' arguments are known, with ? used to denote task and methods where the parameters are unknown.

Definition 4.1 (Subhierarchy)**.** For a given HTN $H = (\mathcal{L}, T_C, T_P, M)$, the subhierarchy associated with task $t \in T_C$ is defined as a tuple $h_t = (\mathcal{L}, T_C^{h_t}, T_P^{h_t}, M^{h_t})$ where:

- $M^{h_t} = \{m = (c, tn_m, \text{pre}(m) \in M \mid c = \text{head}(t))\}$ is the set of methods that refine t.
- $T_C^{h_t} = \{t' \in T_C \mid t' = t \vee \exists m \in M^{h_t}, t' \in tn_m\}$ is the set of compound tasks that appear in the subtasks of the methods decomposing t.
- $T_P^{h_t} = \{t' \in T_P \mid \exists m \in M^{h_t}, t' \in tn_m\}$ is the set of primitive tasks that appear in the subtasks of the methods decomposing t.

With this definition of subhierarchies, we can express the properties required of our candidate set in a more concise manner. For a given subhierarchy h^t:

- For a given method m of t, we have:

$$\text{args}(m) = \bigcup_{t_s \in \text{subtasks}(m)} \text{args}(t_s)$$

- If t is a synthetic tasks, then:

$$\text{args}(t) \subset \bigcup_{m \in M^{h_t}} \text{args}(m)$$

  Note that this leaves the ability have methods parameters that are *not* bound to a parameter of the top level task t.

Furthermore, thanks to this decomposition of a structure with a single top-level task for each subhierarchy, the definition of a bottom and a top for these structures is trivial. Note that we can see each subhierarchy with top level task t as a kind of *reference definition* for t, with its parameters and the parameters of its methods. We can therefore develop a procedure for propagating the arguments upwards in these new structures: instead of propagating from the bottom of the hierarchy, we take each subhierarchy independently and propagate the arguments upwards in this structure and update the methods and top-level task's arguments accordingly. Then, we update all the instantiations of an abstract task *as a subtask in a subhierarchy* according to their now updated definition in their reference subhierarchy. This process can then be repeated until it reaches a fixed point, as presented in Algorithm 4.1. Note that we *never* modify the definition of a demonstrated task (the arguments of its methods may be modified, however).

Algorithm 4.1 Parameter Superset Generation

Input: $\mathcal{H}_S$ an incomplete HTN model

```
H_subs ← the subhierarchies from H_S
repeat
    Π ← args(H_subs)                                  ▷ Existing parameters
    for all h ∈ H_subs do
        PROPAGATE ARGS UPWARDS(h)
    Π_new ← args(H_subs) \ Π                          ▷ New parameters
    for all h ∈ H_subs do
        UPDATE SUBTASKS ARGS(h, Π_new)
until P_new = ∅
```

However, note that in the presence of recursive tasks, such as t_s in our example, this process would never terminate, because new parameters would be created at each iteration. To enforce termination of the algorithm, we need to detect this behaviour during the propagation, including

when the recursions are indirect. To this end, we need to keep track of which methods a parameter has been propagated through during the propagation process. Then, we can limit the number of times a parameter can cross a given method to solve the issue. To this end, we augment each parameter p of a task or method in a subhierarchy with the set $\widehat{M}_p$ of methods through which it has been propagated upwards, so that $\hat{p} = (p, \widehat{M}_p)$.

This extended propagation process is presented in Algorithm 4.2. The mapping between this algorithm and the high level explanation is straightforward: line 3 propagates all the arguments from the subtasks of m into args(m), while line 6 propagates the arguments of the methods into the top level task's, provided it satisfies the method's filtering condition (line 4).

Algorithm 4.2 PROPAGATE ARGS UPWARDS(h_t)

Input: h_t a subhierarchy with top-level task t

```
1: for all m ∈ M^{h_t} do
2:     for all p̂ ∈ args(SUBTASKS(m)) do
3:         args(m) ← args(m) ∪ {p̂}
4:         if m ∉ M̂_p then
5:             p̂' ← (p, M̂_p ∪ {m})
6:             args(t) ← args(t) ∪ {p̂'}
```

The filtering condition guarantees the termination of the algorithm because a parameter can only be added to a top level task by going through a method. This filtering is motivated by the idea that a given task can only set the parameters of its direct subtasks, but in case of a recursive call, the method must be able to handle the parameters of the next instantiation of the subtask. Because there is a finite number of methods in our set of subhierarchies, after some number of iterations, a given parameter will be stopped by the filter even in the presence of recursive tasks. This remains true even if this recursion is indirect. We argue that it is a reasonable limitation as an HTN planner can:

1. Parameterize all the non-recursive subtasks.
2. For each recursive subtask instantiation, choose whether the parameters are the same as the parent task or not.

Finally, Algorithm 4.3 presents the procedure to update the subtasks, called after each round of argument propagation. It is a straightforward procedure that is used to keep a consistent parameterization of every abstract task.

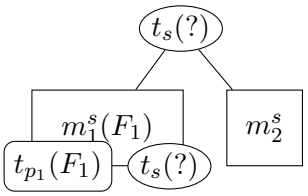
Algorithm 4.3 UPDATE SUBTASKS ARGS(h_t, Π_{new})

```
1: for all t_s ∈ SUBTASKS(h_t) do
2:     Π_new^{t_s} ← {p̂' = (p', M̂_p) | p̂ ∈ args(t_s) ∧ p̂ ∈ Π_new, with p' a new variable}
3:     args(t_s) ← args(t_s) ∪ Π_new^{t_s}
```

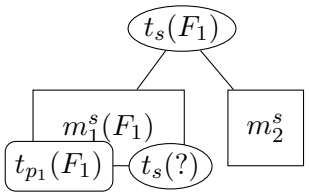
Figure 4.3 illustrates the parameter generation using Algorithm 4.1, focusing specifically on the subhierarchy for t_s from the example presented Figure 4.2, as it is independent of any other subhierarchies. Figures 4.3a and 4.3b shows the effect of the function PROPAGATE ARGS UPWARDS while Figure 4.3c shows the update of the subtasks. Note that due to the recursive nature of t_s, the added parameter during the subtasks update is F_1', as it may or may not be bound to F_1. This process is then repeated, as shown in Figure 4.3d. This time however, the filtering condition for the argument propagation (Alg. 4.2, line 4) is triggered by F_1', preventing

it from being added to the parameters of t_s. As no new changes can be made to the subtasks of t_s (even considering the subhierarchy for t_{top}), all the possible arguments of this subhierarchy have been extracted.

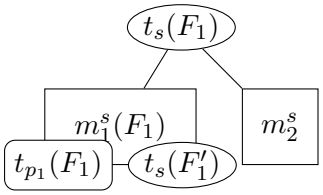
Figure 4.4 shows the resulting parameters for the task t_{top} after applying the same parameter extraction procedure.



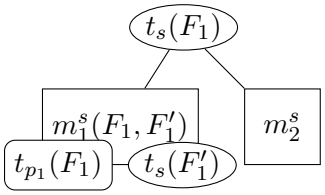
(a) Iter. 1, upwards propagation, step 1.



(b) Iter. 1, upwards propagation, step 2.



(c) Iter. 1, subtasks update.



(d) Iter. 2, upwards propagation, step 1. Final subhierarchy for t_s.

Figure 4.3: Example of argument superset generation for t_s.

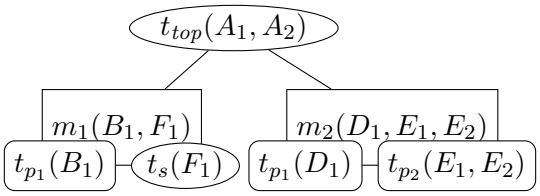
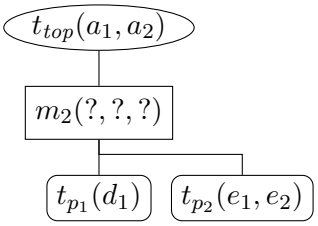
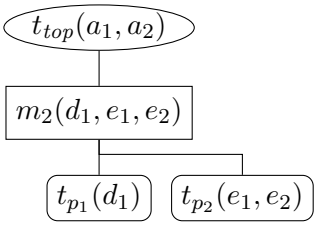


Figure 4.4: Extracted parameters for t_{top}

From this parameterized HTN and the structure of the decomposition trees mapping them to the demonstration traces, we can easily extract parameterized decomposition trees by replacing argument instantiations in the primitive actions and the demonstrated top level tasks and propagating these substitutions throughout the tree. A basic example of decomposition tree is given in Figure 4.5, where a_1, a_2, d_1, e_1 and e_2 represent constants. These decomposition trees will be used to simplify the set of task and method parameters from the demonstration examples.



(a) Before substitutions.

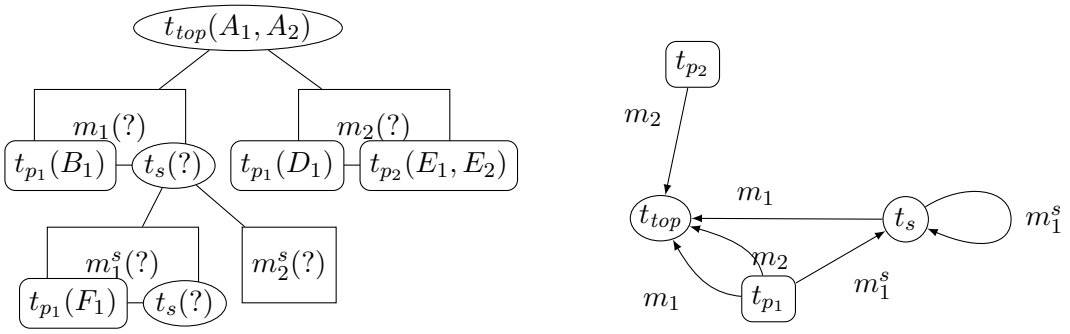


(b) After substitutions.

Figure 4.5: Example of argument propagation in a decomposition tree for a demonstration of $t_{top}(a_1, a_2)$ as the sequence $\langle t_{p_1}(d_1), t_{p_2}(e_1, e_2)\rangle$

4.3.1.2 Candidate Parameter Set Generation as Walks on a Graph

We can observe that this propagation procedure can be represented as a directed graph, where each task in the HTN model is a node, and methods are mapped to labelled edges such that for $(t, t') \in (T_P \cup T_C)^2$, there is an edge labelled m from t to t' if and only if t is a subtask of t'. We will call this graph the *propagation graph*. Considering the HTN presented in Figure 4.6a (repeated from Figure 4.2a), the corresponding propagation graph is presented in Figure 4.6b.



(a) HTN structure. (b) Propagation graph.

Figure 4.6: Propagation graph for an HTN.

Using this graph, it is possible to compute the number of arguments that a task or method will have at the end of the propagation. For a task, this can be achieved by counting the number of *walks*[1] from a primitive task to a target task where all labels are distinct. For a method m, after computing the number of arguments of all tasks, we can sum the number of arguments of all source tasks for the edges labelled m.

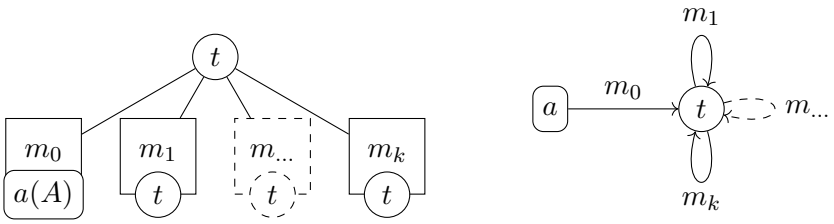
This observation allows us to compute bounds on the number of parameters in a given task. In the absence of any recursions, our graph is directed and acyclic, and for any abstract task t we can extract a directed graph G the node t is a source and the sinks are the primitive tasks with known parameters. Given an (incomplete) HTN $H = (\mathcal{L}, T_C, T_P, M)$, in the worst case, the internal nodes of G are comprised of all the compound tasks except t, and the maximum number of edges from one node to another is the maximum number of methods, written m_{max}. Then, the maximal length of any path is $|T_C| + 1$, and each node may at most generate m_{max} subpaths, which gives:

$$\max_{t \in T_C} |\operatorname{args}(t)| = (|T_C| + 1)^{m_{max}} = O(|T_C|^{m_{max}}) \tag{4.1}$$

However, in practice, this bound is rarely reached and the number of arguments remains tractable.

When recursions are involved, this bound does not remain valid, as it was built on the fact that G was acyclic. Let us focus on an example, where we have a task as shown in Figure 4.7, where we have k different methods that are recursive and a method with a single primitive subtask a. It is easy to see that we have k walks of length two with distinct edge labels. Then, for each of these walks, we can add any $k-1$ more edges to generate walks of length three, and so on until we reach walks of length $k+1$. It is thus easy to see that the number of parameters in this simple case is $O(k!)$. However, this bound is in practice not an issue, as such problematic structures do not tend to appear in planning domains.

[1] A sequence of edges which joins a sequence of vertices.



(a) HTN structure. (b) Propagation graph.

Figure 4.7: A HTN with multiple recursions in different methods, and the corresponding propagation graph

However, we can analyse the cause for this number of path: it stems from the fact we decided that a task must be able to set the parameters for the next *instantiation* of a given subtask. It should therefore be possible to limit this phenomenon by considering that a task must only be able to set parameters for the subtasks in its own direct next instantiation, not that of its subtasks'. This would probably have a limited impact on the parameterization quality, as recursive tasks handling with our current method presents issues anyway, as will be detailed in Section 4.4.

Using another example (Figure 4.8), we can also observe that direct recursions appear as self-loops in the graph, and indirect recursions as Strongly Connected Components (SCCs). We can also detect a layered structure to this graph, which is especially apparent after condensing the SCC formed by the indirect recursion t_1, t_2, as shown in Figure 4.8c: here, we can see that the parameters of t_{top} depend on the parameters of b, t_1 and t_2, the latter depending on a. This observation will be useful in the part of our work dedicated to improving the handling of recursions, currently still in progress at the time of writing this manuscript and presented in Appendix B.

4.3.2 Simplifying the Generated Candidate Sets

Now that we have described a way to extract a set of possible parameters for a given HTN, we need to identify how parameters are passed to its methods and from a method to its subtasks. This is done in a simplification step where we unify parameters from distinct sources.

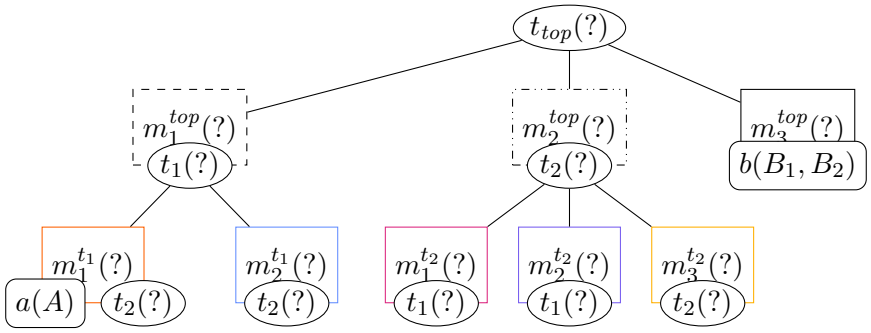
We wish for the set of parameters to be general enough to be able to cover all the examples (and hopefully generalize well to new instances) while still restricting the decomposition possibilities to limit the search effort required of the solver. We propose to achieve this simplification through two main procedures:

1. Parameter unification, where parameters are unified with one another according to the examples.

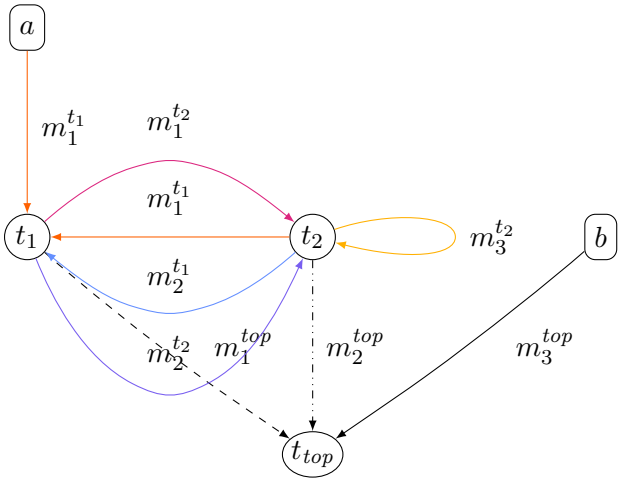
2. Parameter removal, where parameters that do not propagate information (either vertically or horizontally) are dropped.

4.3.2.1 Parameter Unification

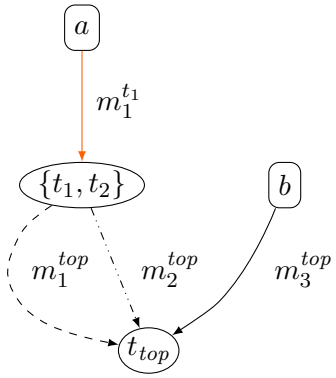
We want to unify as many parameters as possible from the examples given as input. This is motivated by the fact that it will *i)* reduce the number of parameters to instantiate in the model and *ii)* constrain the parameters of the subtask of a given method, allowing them to refer to



(a) An example incomplete HTN. This structure presents an indirect recursion, with t_1 and t_2, as well as two an abstract task t_{top} that uses the recursive tasks. No parameters are known here except the ones in the primitive tasks a and b.



(b) The propagation graph corresponding to the HTN.



(c) Condensed graph corresponding to the propagation graph.

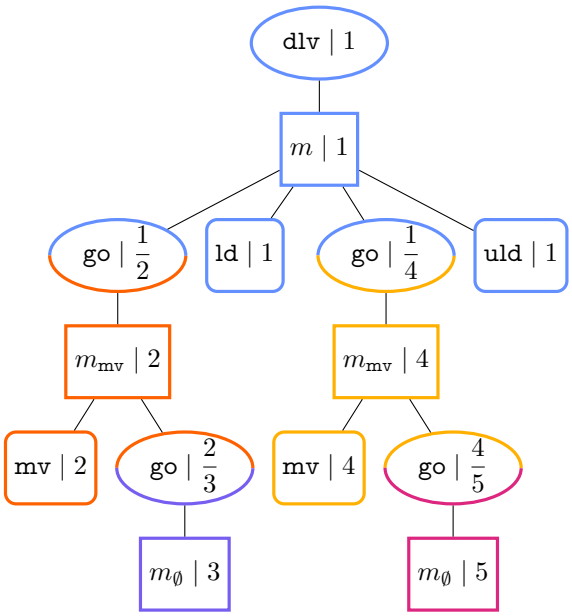
Figure 4.8: An incomplete HTN and the corresponding propagation graph. Colours and line styles used in all subfigures are used to clarify the edge labelling in the propagation graph.

the same constant for the whole method without requiring the planner to infer that this is the best parameterization.

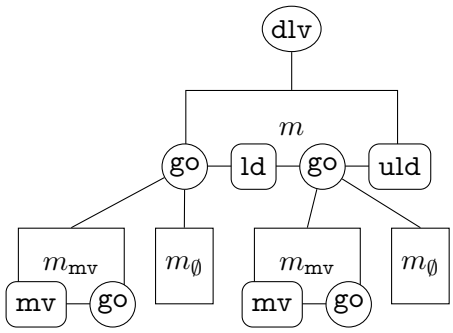
To achieve this unification, we frame the problem as MAX-SMT with the theory of equality and uninterpreted functions. We define args(x) as the function that returns the ordered set of arguments of x, where x may be a method, a task, a subhierarchy or a set of subhierarchies and $\arg_i(x)$ the function that returns the ith argument of x.

Let us write H_{subs} the set of subhierarchies corresponding to a HTN candidate $\mathcal{H}_C$, and $h \in H_{subs}$ a single subhierarchy. We define, for every demonstration $d \in D$ and corresponding decomposition tree, the set I_d of identifiers for the instantiations of the subhierarchies $h \in H_{subs}$ in the corresponding decomposition tree. Similarly to an HTN, we define the sets of compound tasks T_C^h, primitive tasks T_P^h and methods M^h. We additionally write $t^h \in T_C^h$ the top level abstract in a subhierarchy h.

Furthermore, for a variable $?p$, its grounding in the instantiation $i \in I_d$ is written as p_i, which is defined only if such a grounding exists. If a subtask t is abstract and appears in an instantiation $i \in I_d$, we note $\underline{i}_t$ the next instantiation through t. For example, in Figure 4.9, $i = 2$ implies that $\underline{i}_{\texttt{go}} = 3$.



(a) Decomposition tree with identifiers.



(b) Corresponding HTN.

Figure 4.9: Example of decomposition tree with instantiation identifiers for a demonstration $\texttt{dlv} : \langle \texttt{mv}, \texttt{ld}, \texttt{mv}, \texttt{uld} \rangle$ and presented HTN structure.

In order to express our MAX-SMT problem, we introduce the following data types and function:

- $h \in H_{subs}$, a set $\mathcal{V}_h$ of uninterpreted constants corresponding to args(h), the argument variables in h.
- $\forall d \in D$, an enumerated data type $\mathcal{G}_d$ corresponding to $\cup_{t \in d}$ args(t), the constants in a given example.
- For each couple $(h, d) \in H_{subs} \times D$:

- An enumerated data type $\mathcal{I}_d^h$ to represent which instantiation of h in d is currently considered. It is a restriction of I_d to the instantiations of h.
- A function $\text{gnd}_d^h : \mathcal{V}_h \times \mathcal{I}_d^h \to \mathcal{G}_d$ which associates an argument with a grounding in an instantiation of h.

Thanks to the one-to-one mapping from base elements (HTN variables, example constants and instantiations identifiers) to elements of the sets of SMT data types ($\mathcal{V}$, $\mathcal{G}$ and $\mathcal{I}$), we use notations from the original sets to define the constraints for simplicity.

The system of constraints is presented in Table 4.1. Each constraint can be seen has fulfilling a specific role: the *No-Inst* constraint states that an argument that has never been instantiated in the demonstrations should not be unified with any other, to limit overzealous simplifications. The *Def-Consistency* constraint is used to ensure that if we unify two parameters in the reference definition of a synthetic task t, then they must be unified in every instantiation of t as a subtask. *Gnd-Evidence* and *Gnd-Consistency* are used to consider the grounding of the arguments. The first one simply associates an argument with its grounding in the instantiation i of the subhierarchy h_i, while the second one ensures that we do not have inconsistency at the boundary between two instantiations. Finally, *Unif-Examples* is used to add soft constraints (our optimization objective) to try and satisfy as many encountered unifications as possible.

4.3.2.2 Parameter Removal

Once the unification process has taken place, the HTN model may still contain abstract tasks with a large number of parameters, leading to methods with many parameters which will be difficult to instantiate for the solver. Therefore, we propose to remove the parameters that will hinder a solver's performance rather than improve it. We propose to define "useful" parameters as:

1. Parameters enabling the transfer of information from the top of the hierarchy down towards the primitive tasks.

2. Parameters enabling parameter unification across sibling subtasks.

To determine which arguments to remove, we define two functions: $\textsc{Parents}(p)$ and $\textsc{Has Siblings}(p)$.

$\textsc{Parents}(p)$ returns the set of parameters that are used as parents of a given parameter p in method's subtask, allowing to implement rule 1. Using the examples in Figure 4.10, we can give the set of parents for the parameters of the task t, in the different structures presented. In the structure Figure 4.10a, we can see that $\textsc{Parents}(X) = \emptyset$ and $\textsc{Parents}(X) = \emptyset$, while in Figure 4.10b, $\textsc{Parents}(Z) = \{Z_{t_{top}}\}$.

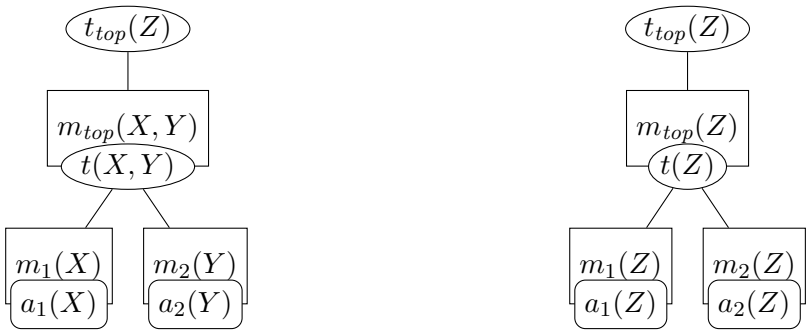
$\textsc{Has Siblings}(p)$ returns $\textsc{True}$ or $\textsc{False}$ depending on whether p is used in two subtasks of the same method, allowing to implement rule 2.

We also define a set of protected parameters for a given set of subhierarchies H, $P_H^\dagger$, which contains all the parameters of the primitive and of the demonstrated tasks, which can obviously never be removed. We can then define the set of simplified arguments of the subhierarchies as:

$$\forall h \in H_{subs}, \text{args}(h) = \left\{ p \in \text{args}(h) \;\middle|\; \begin{array}{l} \textsc{Parents}(p) \neq \emptyset \\ \vee \textsc{Has Siblings}(p) \\ \vee\, p \in P_H^\dagger \end{array} \right\} \tag{4.2}$$

	Name	Condition	Constraint
Hard	No-Inst	$\forall h \in H_{subs}, \forall ?p \in \text{args}(h), \forall d \in D, \forall i \in I_d, \nexists p_i$	$\forall ?p' \in \bigcup_{h' \in S} \text{args}(h') \setminus \{?p\}, ?p \neq ?p'$
	Def-Consistency	$\forall h \in H_{subs}, T = \left\{ t \in \bigcup_{h' \in S} T_C^{h'} \mid \text{sym}(t) = \text{sym}(t^h) \right\}, \forall t \in T \setminus t^h$	$\forall (i,j) \in \lvert \text{args}(t^h) \rvert^2, \text{arg}_i(t^h) = \text{arg}_j(t^h)$ $\Rightarrow \text{arg}_i(t) = \text{arg}_j(t)$
	Gnd-Evidence	$\forall d \in D, \forall i \in I_d, \forall ?p \in \text{args}(h_i), \exists p_i$ with $h_i \in S$ the subhierarchy instantiated at i	$\text{gnd}_d^{h_i}(?p, i) = p_i$
	Gnd-Consistency	$\forall d \in D, \forall i \in I_d, \forall t \in T_C^{h_i} \setminus t^{h_i}$, $h_t \in S, \text{sym}(t) = \text{sym}(t^{h_t}), \forall i_t, \forall j \in \lvert \text{args}(t) \rvert$	$\text{gnd}_d^{h_i}(\text{arg}_j(t), i) = \text{gnd}_d^{h_t}(\text{arg}_j(t^{h_t}), i_t)$
Soft	Unif-Examples	$\forall h \in H_{subs}, \forall (?p, ?q) \in \text{args}(h)^2, ?p \neq ?q$, $\forall d \in D, \forall i \in I_d, \exists p_i, \exists q_i, p_i = q_i$	$?p = ?q$

Table 4.1: Constraints used in the MAX-SMT problem. Conditions are fully expanded to remove quantifiers, and a constraint is added for each expansion of the corresponding condition. Quantifiers are also expanded in each constraint as the chosen solver does not support optimization with quantifiers. Upper constraints are hard while the lower one is soft.



(a) Before unification. (b) After unification.

Figure 4.10: Example of extended subhierarchy used for downward information propagation.

Algorithm 4.4 implements the simplification procedure, where parameters are removed from the tasks and methods until a fixed point is reached.

Algorithm 4.4 Parameter Removal

```
repeat
    P_drop ← ∅
    for all p ∈ args(H_subs) do
        if PARENTS(p) = ∅ ∧ ¬HAS SIBLINGS(p) ∧ p ∉ P_H^† then
            P_drop ← P_drop ∪ {p}
    args(H_subs) ← args(H_subs) \ P_drop
until P_drop = ∅
```

4.4 Handling Recursive Task Definitions: the "*Loop-Until*" Pattern

The extraction and substitution procedure described in the previous section would actually lead to poor parameterization in the case of recursive tasks definitions, allowing the top level arguments to only refer to the first or second instantiation in the recursion. Therefore, we will discuss a generalization of this procedure, starting with a common pattern used to encode loops.

A common usage of recursive task definitions in HTN domains is to encode the "do *something* until *condition*" pattern, which would be difficult to parameterize without considering the last step of the recursion. The ubiquitous $goto(L_1, L_d)$ pattern, presented Figure 4.11, is an example of such a pattern used in many planning domains, used to move an agent from a location L_1 to a location L_d. This is done recursively by chaining *move* actions through intermediate locations until the agent arrives at L_d, mainly to obey location connection preconditions. As can be seen in this example, the L_i parameter is used to constrain the next instantiation of *goto* and the L_d parameter constrains all of them. However, our original method for argument unification cannot generate this parameterization: Figure 4.12 shows the parameter set that we can generate (Figure 4.12b) and the decomposition trees that we could obtain for a given trace (Figure 4.12c), which highlights the impossibility to refer to the end of the decomposition.

To solve this issue, we propose a small modification of the extracted parameters for recursive subhierarchies, presented Figure 4.13, as well as a preprocessing step leveraging the demonstrations, before extracting the full parameterized decomposition trees presented earlier.

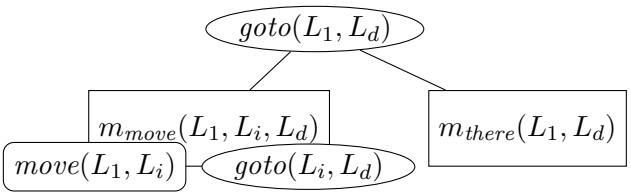
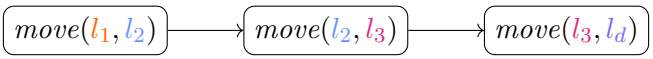
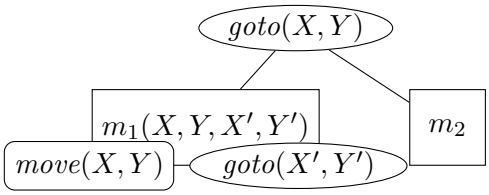


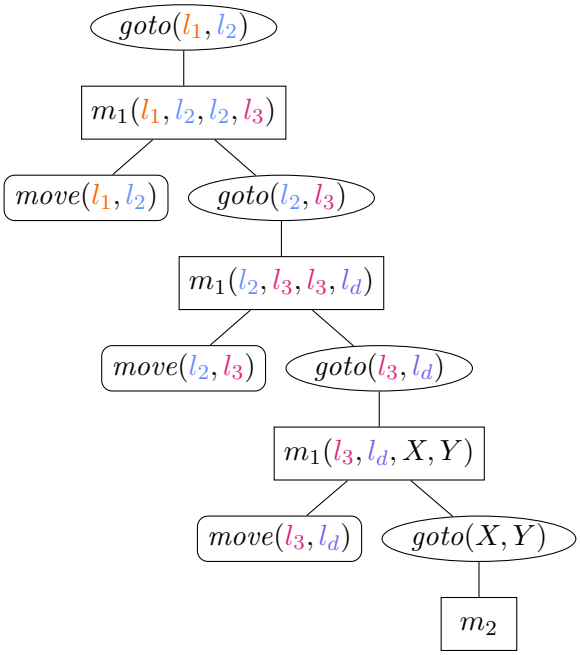
Figure 4.11: Subhierarchy for a *goto* pattern. Preconditions omitted for clarity.



(a) Demonstration trace.



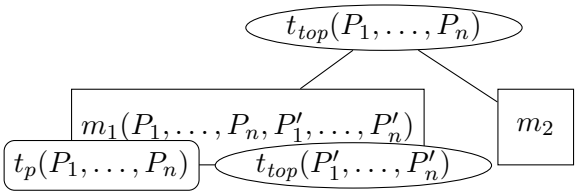
(b) Structure with parameters candidate set generated using the presented method.



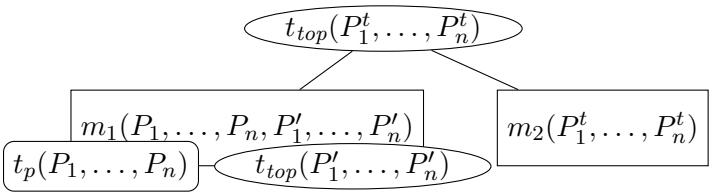
(c) Decomposition tree.

Figure 4.12: Possible example trace and corresponding decomposition tree example for the *goto* task, with the subhierarchy parameters extracted using the method presented in previous section. Colours are used to highlight identical constants.

This process will be illustrated using a simple subhierarchy structure, but could be easily generalized to any task with a single recursive subtask. While this covers many of the use cases, more work is needed for this to work on arbitrary task hierarchies. The main idea is to be able to map parameters of the top task of a given recursive subhierarchy to parameters from the demonstrations' primitive actions, while considering that recursive tasks should be able to refer to parameters at the end of a recursion.



(a) Basic recursive hierarchy.



(b) Modified recursive hierarchy.

Figure 4.13: Parameters modification for recursive tasks.

We first modify the parameters to keep track of the non-recursive parameters from which the recursive one has been generated, modifying the extracted structures from the previously extracted one, presented Figure 4.13a, into the one presented in Figure 4.13b. In this example the P^t_i parameters shows that this parameter originated from the P_i parameter of the task t_p, but we do not know whether it should refer to the immediate instantiation of P_i or if it needs to refer to its last instantiation. The P'_i are the instantiation of the parameters of the task t_{top} in the recursion chain, generated in the same way as in the example Figure 4.3.

We then can substitute the ground parameters in the non-recursive subtasks in each recursion chain, as presented in Figure 4.14 where all the p^j_i represent constants.

Applying this process to a *goto* task for which we want to learn the parameters, we obtain the subhierarchy presented in Figure 4.15. A decomposition tree for a given example demonstration trace is given in Figure 4.16. This task will be used as a running example to illustrate the remainder of this section.

From this new set of parameters, we then need to determine, $\forall i \in [1, n]$, if P^t_i is bound to P_i or P'_i, or to the parameters of the last step of the chain, noted P^L_i. Furthermore, we want to know if some parameters are bound together in the method, transferring information from one step of the chain to the next. We note P^+_i the instantiation of the parameter P_i in the next step and $\mathcal{P}$ the set of all arguments in the example and the subhierarchy. $\mathcal{P}_m \subset \mathcal{P}$ represents the set of arguments of a method m in the considered subhierarchy and $\mathcal{P}_{\text{top}} \subset \mathcal{P}$ the set of arguments of the top level task.

Leveraging the structure of the subhierarchies and the demonstrations, we cast the problem of grouping parameters together as a MAX-SMT problem with the goal of optimizing the size of the groupings of method parameters and the number of top level task parameters bound to their last instantiation in a recursion chain. We can divide the set of constraints that will define our SMT problem into two categories: constraints that ensure consistency with the previously defined HTN structure, and constraints that ensure consistency with the given examples.

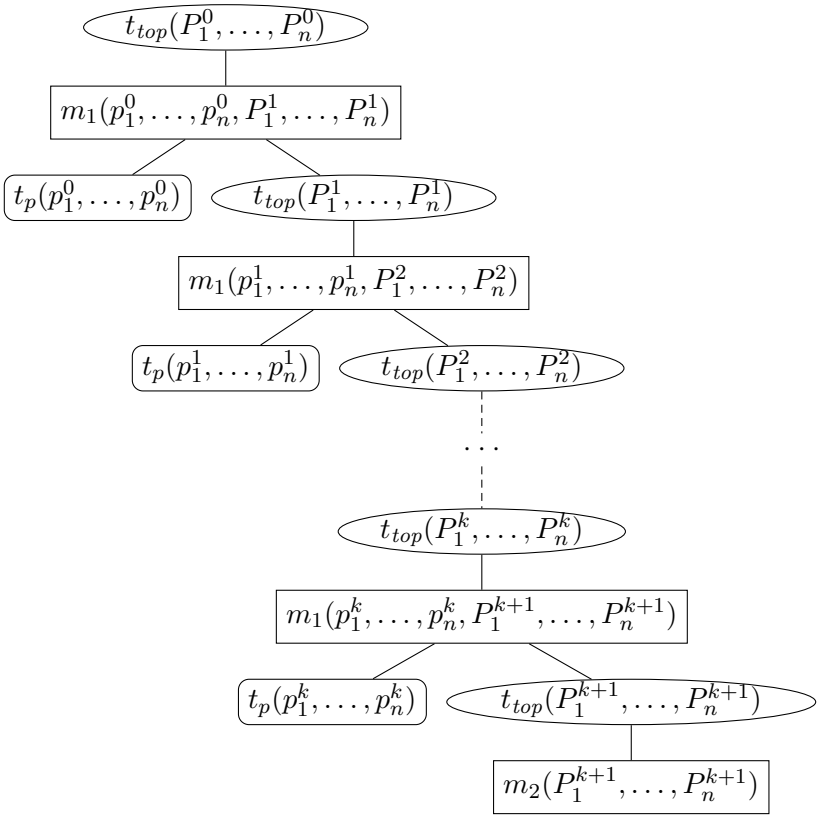


Figure 4.14: Generic decomposition tree for a recursive hierarchy.

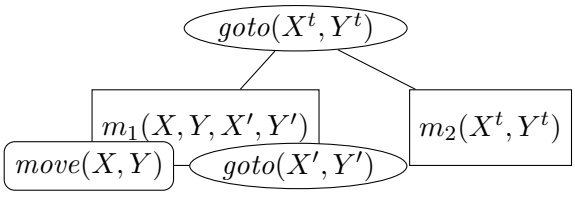
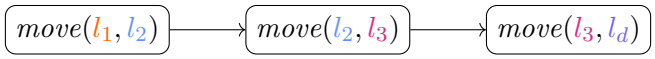
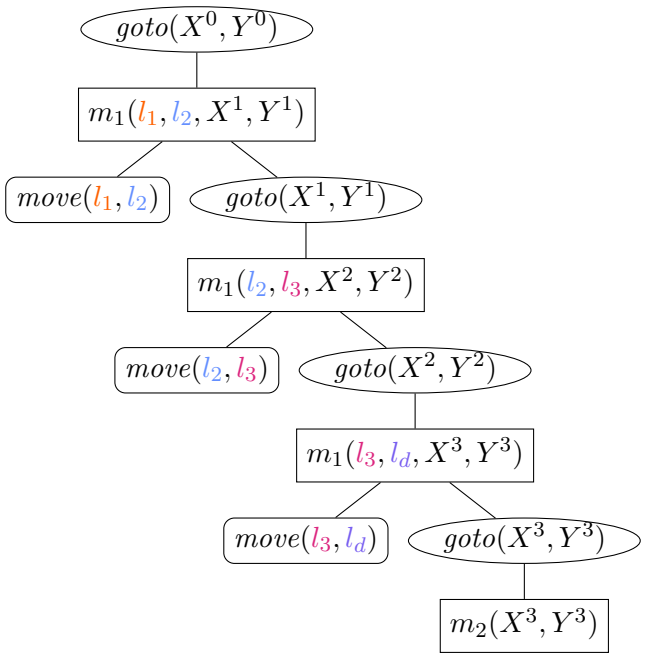


Figure 4.15: Extracted subhierarchy for a *goto* task before recursion processing.



(a) Demonstration trace.



(b) Decomposition tree.

Figure 4.16: Possible example trace and corresponding decomposition tree example for the *goto* task. Colours are used to highlight identical constants.

4.4.1 Ensuring Consistency with the HTN Structure

From the presented model in Figure 4.13b, we can extract the following structural constraints where hard constraints represent properties that must hold for the model to be consistent:

$$\forall i \in [1, n]$$

$$\textsc{Hard}(P_i^t = P_i \vee P_i^t = P_i') \tag{4.3a}$$

$$\textsc{Hard}(P_i^t = P_i' \Rightarrow (P_i^t = P_i^L \vee P_i' = P_i^+)) \tag{4.3b}$$

$$\textsc{Hard}(P_i^t = P_i^L \Leftrightarrow P_i' = P_i^L) \tag{4.3c}$$

$$\textsc{Hard}(P_i^t = P_i \Leftrightarrow P_i' = P_i^+) \tag{4.3d}$$

$$\textsc{Soft}(P_i^t = P_i^L) \tag{4.3e}$$

Constraints 4.3a and 4.3b are used to enforce consistency with the origin of a top task parameter P_i^t. Constraint 4.3a enforces the fact that the top level task argument either comes from the non-recursive subtasks (left) or the recursive instantiation (right) while constraint 4.3b enforces the fact that a top level parameter may only propagate information towards the next step in a recursion or towards the last one.

The constraints 4.3c and 4.3d are used to enforce consistency within the model generated by the constraint satisfaction solver.

The soft constraint 4.3e encodes the desirable, but not necessary, property that an argument is always passed recursively which avoids the need for the planner to non-deterministically choose its value.

4.4.2 Ensuring Compatibility with the Demonstrations

Let us now turn our attention to the constraints that must hold to ensure that parameter passing is consistent with the demonstrations in a given hierarchy.

We can also extract the constraints defined in Equations 4.4∗ from each example with a structure as presented in Figure 4.14. We note $\forall i \in [1, n], \forall s, s' \in [0, k] \cup \{L\}, p_i^{s',s}$ the argument $p_i^{s'}$ considered at step s, in order to allow parameters to refer to independent constants at each step.

$$\forall i \in [1, n]$$

$$\textsc{Hard}(P_i^L = p_i^{k,L}) \tag{4.4a}$$

$$\forall s \in [0, k[, \textsc{Hard}(P_i^+ = p_i^{s+1,s}) \tag{4.4b}$$

$$\forall s \in [0, k]$$

$$\textsc{Hard}(P_i = p_i^{s,s}) \tag{4.4c}$$

$$\textsc{Hard}\begin{pmatrix}\forall s' \in [0, k] \cup \{L\}, \forall j \in [1, n], \\ \text{sym}(p_i^{k,L}) \neq \text{sym}(p_j^{s,s'}) \Rightarrow p_j^{k,L} \neq p_j^{s,s'}\end{pmatrix} \tag{4.4d}$$

$$\textsc{Hard}\begin{pmatrix}\forall s', s'' \in [0, k], \forall j \in [1, n], \\ \text{sym}(p_i^{s',s}) \neq \text{sym}(p_j^{s'',s}) \Rightarrow p_i^{s',s} \neq p_j^{s'',s}\end{pmatrix} \tag{4.4e}$$

When considering our *goto* task, some possible constraints that ensure consistency with the example presented in Figure 4.16 are presented in Example 4.1.

Example 4.1 *Some Instantiations of the Demonstration-Based Constraints*
Equation 4.4a defines the binding for the last instantiation of each top task parameter as presented in the following equation:

$$\{X^L = l_3^L \quad Y^L = l_d^L\} \tag{4.5}$$

Equation 4.6 shows the bindings from steps 0 and 1 in the decomposition tree, showcasing the effect of the equations 4.4b and 4.4c.

$$\begin{Bmatrix} X = l_1^0 & X' = l_2^0 \\ Y = l_2^0 & Y' = l_3^0 \end{Bmatrix}_0 \quad \begin{Bmatrix} X = l_2^1 & X' = l_3^1 \\ Y = l_3^1 & Y' = l_d^1 \end{Bmatrix}_1 \tag{4.6}$$

Finally, equation 4.7 shows the action of constraint 4.4d, preventing unsound unifications involving the last instantiation of a given task parameter.

$$\{l_3^L \neq l_1^0 \quad l_3^L \neq l_2^0 \quad l_3^L \neq l_2^1 \quad l_3^L \neq l_d^1 \quad l_3^L \neq l_d^L\} \tag{4.7}$$

4.4.3 Minimizing Method Parameters Through Unification

To determine which parameters are bound together during the optimization process, we define a set $\mathcal{G}$ of potential groups for each $p \in \mathcal{P}$ and a function PGroup (equation 4.8a) which maps each unique parameter to a single group (equation 4.8c). We also define a function NotCountG (equation 4.8b) which will be used in the definition of the optimization objectives and is defined through the constraint presented in equation 4.8d.

$$\text{PGroup} : \mathcal{P} \to \mathcal{G} \tag{4.8a}$$

$$\text{NotCountGroup} : \mathcal{G} \to \{0, 1\} \tag{4.8b}$$

$$\textit{Hard} \quad \begin{pmatrix} \forall p_1, p_2 \in \mathcal{P} \\ \quad \text{PGroup}(p_1) = \text{PGroup}(p_2) \Rightarrow p_1 = p_2 \end{pmatrix} \tag{4.8c}$$

$$\textit{Hard} \quad \begin{pmatrix} \forall g \in \mathcal{G}, \text{NotCountG}(g) \\ \quad \Leftrightarrow \begin{cases} \nexists p \in \mathcal{P}_{m_1}, \text{PGroup}(p) = g \\ \vee \exists p \in \mathcal{P}_{\text{top}}, \text{PGroup}(p) = g \end{cases} \end{pmatrix} \tag{4.8d}$$

We define the objectives for our optimization problem in equation 4.9 with $\mathcal{C}_{\text{Soft}}$ designating the set of soft constraints. These objectives are considered in lexicographic order.

The first optimization objective (Eq. 4.9a) is used to satisfy two of our objectives: i) grouping method arguments together, to allow transferring information from one step of the recursion to the next and ii) binding subtask arguments to top level tasks arguments.

The second optimization objective (Eq. 4.9b) is used to satisfy the constraints binding top level arguments to the instantiation of arguments in the last step of recursion (Eq. 4.3e) in order to transfer information throughout the recursion.

$$\max \sum_{p \in \mathcal{P}_{m_1}} \textsc{NotCountG}(\textsc{PGroup}(p)) \tag{4.9a}$$

$$\max \sum_{c \in \mathcal{C}_{\text{Soft}}} \textsc{Satisfied}(c) \tag{4.9b}$$

Solving this problem will generate a set of equivalence classes. We then replace each of these classes in the modified subhierarchy with a single new parameter, unifying parameters with their right instantiation.

Considering again our *goto* task example, solving the associated MAX-SMT problem will produce the equivalence classes presented equation 4.10. Replacing each equivalence class in the subhierarchy Figure 4.15 with a parameter using the naming scheme shown under the classes will yield the same structure as presented in Figure 4.11, which is the expected result.

$$\underbrace{\{X, X^t\}}_{L_1} \qquad \underbrace{\{Y, X'\}}_{L_i} \qquad \underbrace{\{Y', Y^t, Y^L\}}_{L_d} \qquad \{X^L\} \tag{4.10}$$

While we tried to expand these ideas to arbitrary recursive structures, such as ones containing indirect and multiple recursions, this work is not yet complete. An overview of what has been achieved, some preliminary results as well as the remaining challenges is given in Appendix B.

4.5 Conclusion

In this chapter, we presented a MAX-SMT approach for learning the parameters for a given HTN for which we only know the parameters of the primitive actions and potentially some abstract tasks.

In the process, we first highlighted the difficulty of propagating the parameters in the presence of recursive methods and proposed a solution to this problem that generates a correct set of parameters.

We then proposed to use a MAX-SMT approach to unify parameters together using evidence from the demonstration in order to provide guidance to an automated planner using the HTN. However, we noted the limitations of this approach, in terms of parameterization quality, in the presence of recursions in the hierarchy. We then proposed to detail how these issues could be solved in the case of a specific recursive structure, the *loop until* construct, using a similar MAX-SMT approach to the previously presented one.

A promising avenue for future research would be to try and adapt this procedure to any arbitrary recursive structure. While we have given some thought to the subject, with some insights presented in Appendix B, this problem remains unsolved. The two main challenges that we still face are to handle the large number of candidate parameters that can be generated, as well as expressing the problem as a set of constraints that an optimizing SMT solver can handle.

Chapter 5

Experimental Evaluation

Contents

5.1 Introduction . . . 87
5.2 Planning Domains Presentation . . . 87
5.3 Environment and Datasets . . . 88
5.3.1 Environment . . . 88
5.3.2 Datasets . . . 90
5.4 Planning Performance . . . 91
5.4.1 Rovers . . . 91
5.4.2 Logistics . . . 96
5.4.3 Childsnack . . . 99
5.4.4 Satellite . . . 102
5.4.5 Woodworking . . . 105
5.5 Learning Times . . . 108
5.6 Conclusion . . . 113

5.1 Introduction

In order to evaluate the performance of the learned models, we have performed an experimental analysis on several planning domains, mainly from the International Planning Competition (IPC). We will start this chapter by presenting the domains from which we will learn, before focusing on different metrics for evaluating the quality of the learned models, namely coverage, planning speed and plan length.

The code and experimental data are available as a repository in the PLAN@LAAS GitHub organization (`https://github.com/plaans`).

5.2 Planning Domains Presentation

Let us now present the domains that we evaluated our learner on, with their main characteristics as well as that of the training and test sets for each. In every domain, demonstration traces where obtained using the Lilotane planner [Sch21] with a time limit of five minutes and the optimization setting turned on with a factor of ten. This means that for every plan found in n seconds, the planner was free to spend $10n$ seconds searching for an optimal plan. While this does not guarantee that an optimal plan will be generated, we always manually inspected some of the generated plans and never found a non-optimal one.

Each of the domains considered will have a reference model, which will be used to compare our learned Hierarchical Task Networks (HTNs) against. In domains from the IPC, it will be the HTN from the competition and otherwise a handcrafted one.

Reference domains, example instances and notable learned domains are presented in Appendix D.

Logistics Logistics is a delivery domain, where packages can be transported in a city using trucks, and across cities using planes. Note that we are using the version from the HTN-Maker evaluation dataset rather that ones from the IPC. This version was modified to use types rather than predicates, so that it would be closer to the hierarchical IPC version and our parameterization procedure would be able to use it. For simplicity reasons, the airport location where left as being specified using predicates. We however modified the IPC domain to be usable on the HTN-Maker instances in order to use it for comparison.

We initially intended to use the original HTN-Maker-learned domains for comparison, but were not able to convert it to be compatible with planners that take HDDL as input in a timely fashion.

Rovers The Rovers domain models problems where rovers have to obtain samples and communicate associated data as part of three distinct tasks: sample rock, sample soil or get image data. We are using instances from the IPC, which have the particularity of including several features used to model advice, including in the primitive actions. These "advising" actions can be divided into two categories:

- No-ops that do not affect the state of the world (namely `nop`, `visit` and `unvisit`), which are removed from demonstrations and plan cost calculation in the evaluation.
- Different versions of actual actions with specific parameterization used to guide the planner (such as `communicate_rock_data1` and `communicate_rock_data2`). These are kept-as is, in order to be able to use the reference model unmodified.

Satellite Satellite is a domain where the agent has to use a set of satellites to gather some information. This information gathering involves turning the satellite towards a target object and then using an instrument with the right capabilities to gather the information. Not all satellites are equipped with the same instruments.

Childsnack Childsnack is a domain where the goal is to make and serve sandwiches to children, taking into consideration a possible gluten allergy. It is a simple domain for which the reference domain contains a single task that can be achieved with one of two non-recursive methods.

Woodworking Woodworking is a domain where the goal is to process wooden planks. The main difficulty stems from the large number of possible values that the parameters can take.

5.3 Environment and Datasets

5.3.1 Environment

All the experiments in this section were run on the LAAS-CNRS HPC cluster, with each run task being allocated 12 CPU cores and 32 GB of RAM. Each task was run in an Ubuntu

20.04-based Apptainer container. Learned models planning capabilities were evaluated using the Lilotane planner with plan length optimization disabled.

Note that during the learning phase, for all the domains, the trace matching algorithm presented in Chapter 3 leverages multiple cores, matching up to 12 traces in parallel. The parameter learning phase is single threaded, and the multiple cores were instead leveraged to run multiple parameterization instances in parallel to reduce experiment time. As most of the structure learning time is spent matching the demonstrations, this should be kept in mind when considering the results presented in this chapter.

Learning Hyperparameters Let us briefly present the different learning parameters that will define the different possible configurations in Table 5.1. The symbols presented in this table will be used to generate unique labels to name different combinations of parameters, especially useful in graphics presenting results for different parameterizations.

Parameter	Sym.	Domain	Description
α	α	$\mathbb{R}_+^*$	α factor in the description length.
Simplify HTN	S	$\mathbb{B}$	Apply the simplification operator to the generated HTNs.
Allow Repeated Patterns	$*$	$\mathbb{B}$	Allow generating patterns containing the $*$ and $+$ modifiers.
Generate Choice Patterns	C	$\mathbb{B}$	Allow generating choice patterns.
Sequence Pattern Max Length	Kk	$\mathbb{N}_+$	Maximal length k of the generated sequential patterns.
HTN-Maker Method Generation Mode	Nbx	$\left\{ \begin{array}{l} \texttt{Rec}, \\ \texttt{Lge}, \\ \texttt{RcLge} \end{array} \right\}$	Set of methods generated using our HTN-Maker-based algorithm.

Table 5.1: The different parameters that govern the search procedure and their domains.

Before going moving to the labelling process, let us detail some more the "*HTN-Maker Method Generation Mode*" parameter. Remember how in Chapter 3, we showed that the set of possible methods generated using an adapted HTN-Maker algorithm to structure-only demonstrations can be represented as a triangular matrix, reproduced in Equation 5.1 for clarity.

$$\begin{array}{c} \\ 0 \\ 1 \\ \vdots \\ n-1 \\ n \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & \dots & n-1 & n \end{array} \\ \begin{bmatrix} \langle a_0, t\rangle & \langle a_0, a_1, t\rangle & \langle a_0, \dots, t\rangle & \langle a_0, \dots, a_{n-1}, t\rangle & \langle a_0, \dots, a_n\rangle \\ & \langle a_1, t\rangle & \langle a_1, \dots, t\rangle & \langle a_1, \dots, a_{n-1}, t\rangle & \langle a_1, \dots, a_n\rangle \\ & & \langle \dots, t\rangle & \langle \dots, a_{n-1}, t\rangle & \langle \dots, a_n\rangle \\ & & & \langle a_{n-1}, t\rangle & \langle a_{n-1}, a_n\rangle \\ & & & & \langle a_n\rangle \end{bmatrix} \end{array} \tag{5.1}$$

In our implementation, in order to reduce candidate HTN structure trace matching-time, and thus model learning time[1], we only generate one or two neighbours per (abstracted) trace, as follows:

Most Recursive This corresponds to the value `Rec`, where we generate the neighbour that correspond to the set of all methods on the diagonal of the matrix.

[1] A more detailed analysis of learning times is given in Section 5.5.

Largest This corresponds to the value `Lge`. Here, we generate a neighbour that contains *all* possible methods in the matrix (ignoring duplicates), assuming that we will be able to filter out unused methods if need be.

Most Recursive and Largest Corresponding to the value `RcLge`, we generate both the previously presented neighbours for each abstracted trace.

Let us now detail how the labels will be formed using the symbols presented earlier. For symbols representing boolean values, the symbol will be absent if the parameter is set to false. Let us take as example a model learned with $\alpha = 0.1$, with unused method filtering and repeated patterns, sequential patterns of maximal length $k = 3$ and no choice patterns, with neighbours generated using the most recursive mode. The label would then be α0.1`_F_*_K3_NbRec`. Note that whenever the neighbour generation is limited to a single mode, the corresponding symbol will be omitted for clarity.

We also append some details to the label. The first labels that we can append are related to the learned parameters of the model: if the model has learned parameters, we append `Prm` to the previous label. If the parameters where learned in such a way that a direct recursion to a known abstract task was enforced to keep the same parameters as the parameters, the label `Prm_Rec` is added instead.

Note that when presenting the results, the reference HTN will always be presented for comparison under the label `REF`. For the Logistics domain, we have two reference domains: a handmade one, which leverages the structure of the domain to be as efficient as possible, and the learned model that was present in the dataset from IPC 2020. These models will be respectively labelled `REF` and `REF_LEARNED` when presenting the results.

Number of Training Demonstrations and Neighbour Generation Mode We also append the label `NT`k to show the number of training demonstrations used to learn this domain, where k is the number of demonstrations. For every domain, learning was attempted with $k \in [5, 10, 20, 40, 80, 160]$, limited to the maximal number of generated demonstrations.

Neighbour generation modes other than *most recursive* were only used with $k \leq 10$, in order to keep computation times reasonable.

5.3.2 Datasets

For the domains presented earlier, let us describe the datasets: how they were generated, and how they were used for training and evaluation.

For every domain, we generated demonstrations using the Lilotane planner with optimization turned on and a time limit of 10 seconds, using the reference domain as a source. Every solved instance solution was then split in order to generate one trace per task. Sets of k training instances were then generated through random sampling of the generated demonstrations.

Furthermore, for every domain from the totally ordered track of the hierarchical 2020 IPC (i.e. every domain but Logistics), we increased the number of instances by shuffling the tasks in the initial task network, once for each instance. The set of testing instances is then always taken as the whole set of base and augmented IPC instances. Note that due to the limited number of instances in the IPC domains, when a large number of training instances was used, the learned models may exhibit some degree of overfitting.

The number of generated demonstrations for each domain is presented in Table 5.2.

	Instances		
Domain	Base	Augmented	Generated demonstrations
Childsnack	30	30	600
Logistics	100	–	162
Rovers	30	30	74
Satellite	20	20	323
Woodworking	29	29	812

Table 5.2: Number of IPC planning instances and generated demonstration traces for each domain.

5.4 Planning Performance

In this section, whenever HTNs are ranked against one another, the metric used for ranking was the coverage of the HTN, with ties broken by the cumulative planning time on successful instances.

5.4.1 Rovers

Let us first focus on the Rovers domain. We evaluated learning with the set of parameters presented in Table 5.3.

Parameter	Possible Values
α	$0.001, 0.01, 0.1$
Filter Unused Methods	$\top$
Simplify Structure	$\bot, \top$
$*$	$\bot, \top$
Gen. Choice Patterns	$\bot, \top$
Seq. Pattern Max Length	$3, 4$
Planning	120s, 2500 MB

Table 5.3: Learning evaluation parameters for the Rovers domain.

The resulting coverage percentage from this learning phase is given in Table 5.4, limited to the best models for clarity.

In order to obtain more insight into the performance of the different models, let us look at the empirical cumulative distributions of the planning times presented in Figures 5.1 and 5.2.

We can observe that parameterized domains present better performances, both in terms of coverage and planning time. We can more specifically note that parameterized domains tend to be faster than the reference domain, while non-parameterized ones tend to not scale to the more complex instances.

Analysing the domains in a more qualitative manner, we can observe that the best structure (Appendix D, Figure D.2) does make little use of recursions: only in the `navigate_one_or_more` synthetic task (pattern `navigate`+), which is a task with a very local effect and few instantiations in the whole model. Due to this limited use of recursions, our parameter learning algorithm is able to unify many parameters among sibling tasks, as well as transfer

Label	Solved Instances (%)
REF	61.7
α0.001_*_K4_C_NbRec_Prm_Rec_NT74	75.0
α0.01_S_*_K4_C_NbRec_Prm_NT74	75.0
α0.1_S_*_K4_C_NbRec_Prm_Rec_NT60	73.3
α0.1_S_*_K4_C_NbRec_Prm_NT60	73.3
α0.1_S_K3_NbRec_NT74	56.7
α0.001_*_K4_NbRec_NT74	56.7
α0.001_K3_NbRec_NT74	56.7
α0.1_K4_NbRec_NT74	56.7

Table 5.4: Coverage for the ROVERS domain, restricted to the best four parameterized and non-parameterized domains.

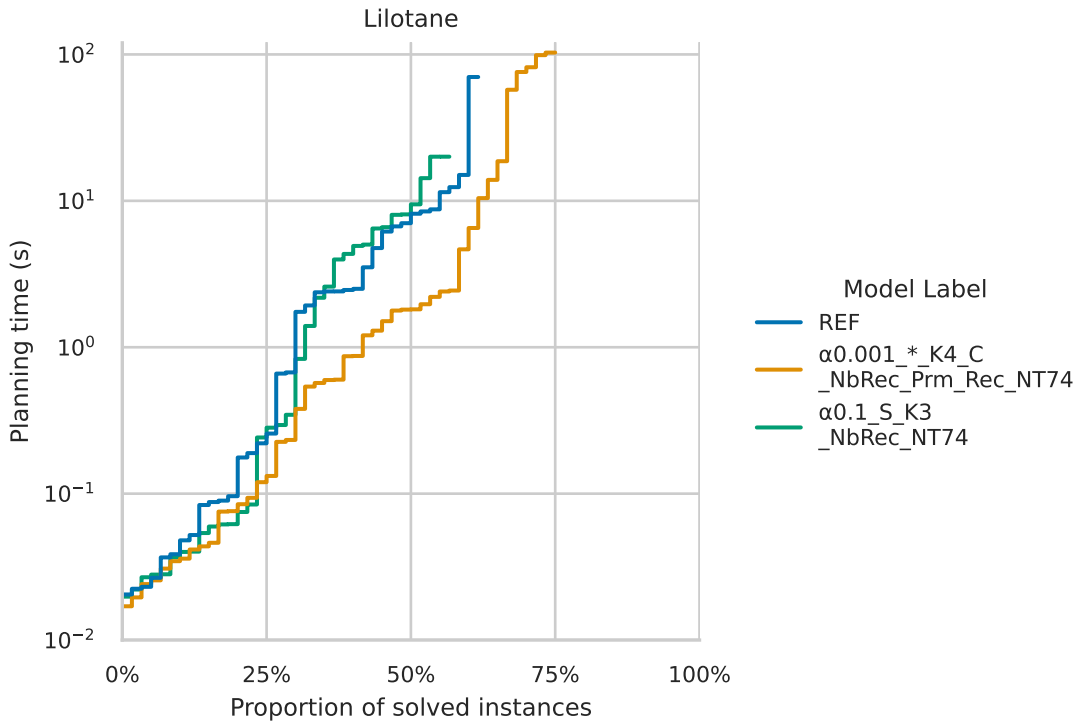
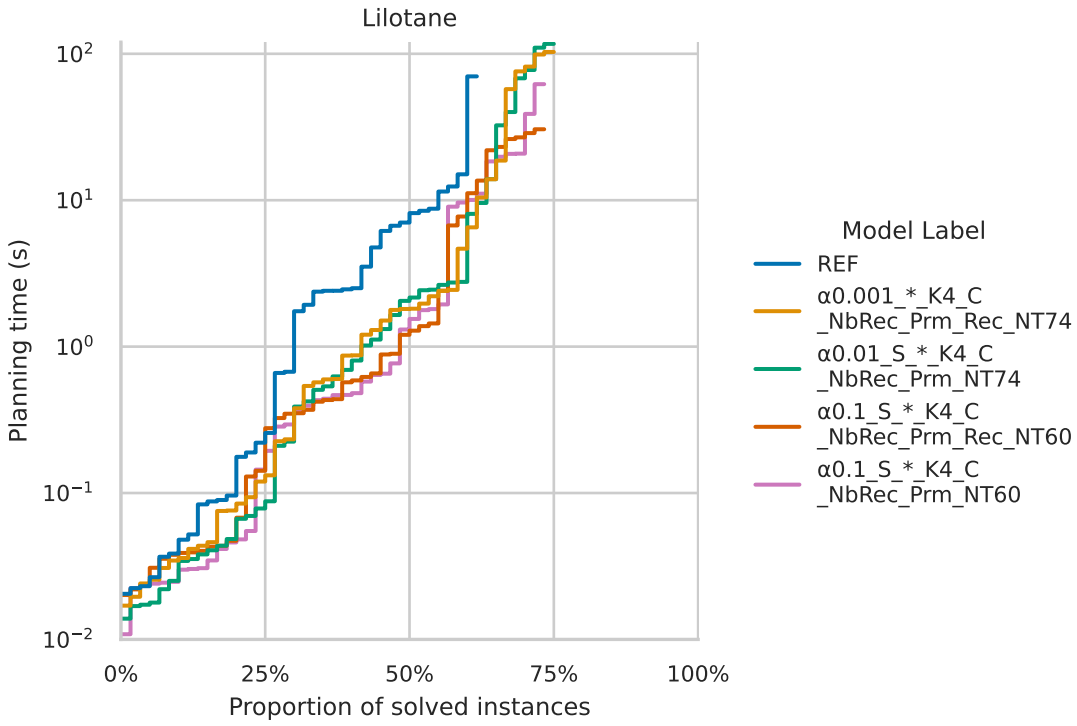
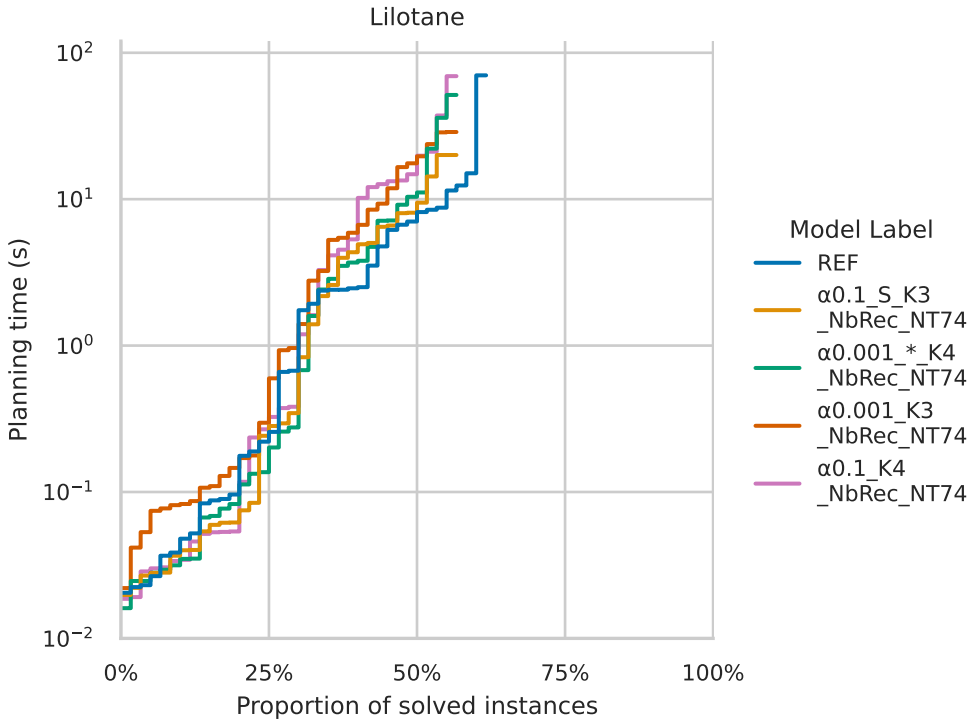


Figure 5.1: Planning time cumulative distribution for the ROVERS domain, for the best parameterized and non-parameterized domains.



(a) Planning time distribution, limited to parameterized models.



(b) Planning time distribution, limited to non-parameterized models.

Figure 5.2: Planning time cumulative distribution for the ROVERS domain, focusing on parameterized and non-parametrized HTNs.

information from the top-level tasks down to the primitives.

Focusing on the best and the reference models only, we can compare their planning performances on the set of commonly solved instances (37/60). Figure 5.3 shows that the planning time is consistently better for the learned models.

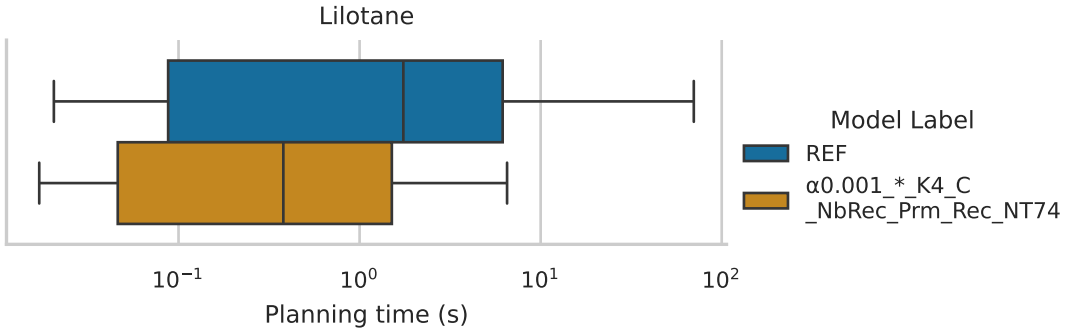


Figure 5.3: Planning time box plot, limited to the best and reference models and the instances solved by both of them.

We can do a similar analysis on the plan length: our learned models tend to consistently generate shorted plans, as presented in Figure 5.4. This which may be explained by the limitation imposed by the sequential structures found in the learned method, where unified parameters allow the planner to quickly progress towards relevant solutions. We conjecture that this is especially due to the more limited navigation capabilities of our model, which allows the planner to avoid useless movement actions.

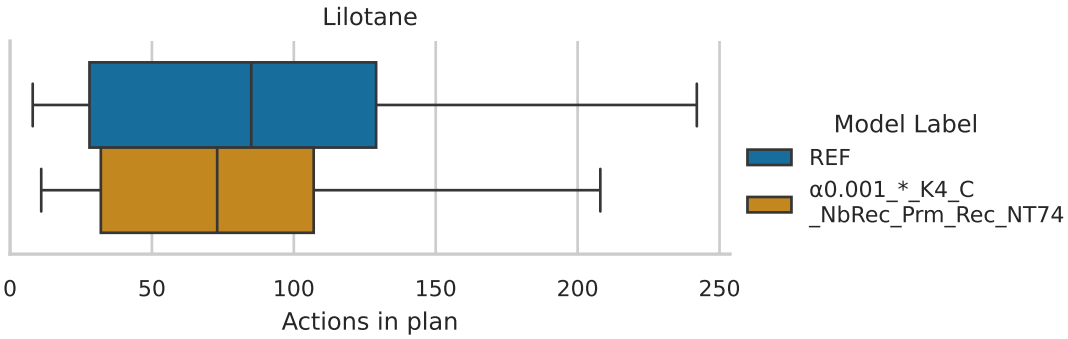
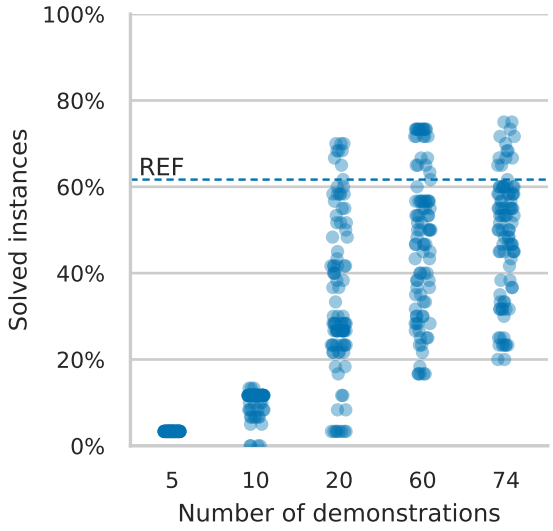


Figure 5.4: Plan length box plot, limited to the best and reference models and the instances solved by both of them.

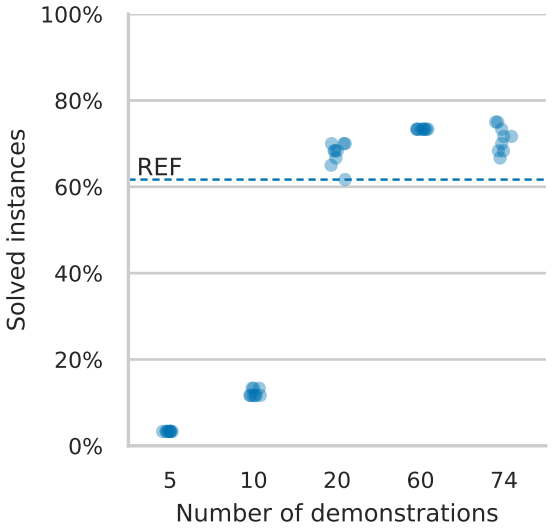
Let us now focus on the impact of the number of demonstrations[1] on the planning performances, as presented Figure 5.5.

We can observe on Figure 5.5 that while the best models make use of all 74 demonstrations, the reference model is outperformed by the best model learned with as little as 20 demonstrations. Furthermore, focusing on Figure 5.5c, we observe that the non-parameterized models tend to exhibit a similar improvement trend as the parameterized ones, however with a distribution that has less great and poor models bur rather unremarkable ones. We conjecture that

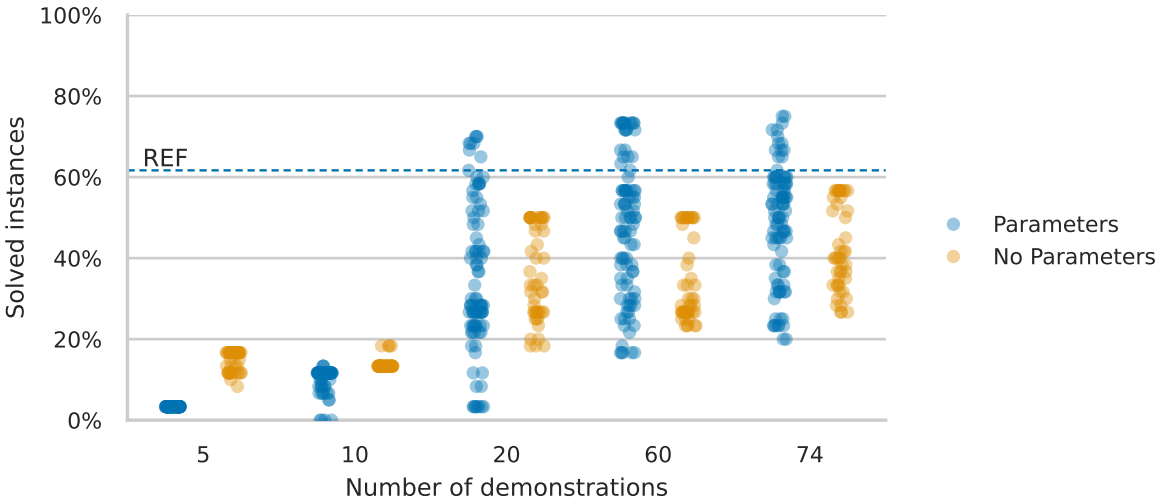
[1]Due to a crash during the experimental run, no data is available for a training set containing 40 demonstrations.



(a) Coverage against number of training demonstrations for all models. Limited to parameterized models.



(b) Coverage against number of training demonstrations for the top 10% learned models. Limited to parameterized models.



(c) Scatter plot of the coverage against the number of training demonstrations.

Figure 5.5: Impact of training demonstration set size on coverage. Note that the x-axis scale is not linear.

this behaviour is due to the fact that the absence of parameterization allows for more flexibility at the cost of a more expensive search procedure. This flexibility allows poor structure to generalize well-enough, but the search cost associated with the larger search space causes the planner to fail on complex instances.

5.4.2 Logistics

Let us now turn our attention to the Logistics domain. Learning parameters and coverage results are presented in Tables 5.5 & 5.6, respectively.

Parameter	Possible Values
α	$0.001, 0.01, 0.1$
Filter Unused Methods	$\top$
Simplify Structure	$\bot, \top$
$*$	$\bot, \top$
Gen. Choice Patterns	$\bot, \top$
Seq. Pattern Max Length	$3, 4$
Planning	10s, 2500 MB

Table 5.5: Learning evaluation parameters for the Logistics domain.

Label	Solved Instances (%)
REF	100.0
IPC_LEARNED	100.0
α0.1_S_*_K3 _NbRec_Prm_Rec_NT20	100.0
α0.1_S_*_K3 _NbRec_Prm_Rec_NT40	100.0
α0.1_S_*_K3 _NbRec_Prm_NT60	100.0
α0.1_S_*_K3 _NbRec_Prm_Rec_NT160	100.0
α0.1_S_*_K3_C _NbLge_NT10	100.0
α0.1_S_K4_C _NbRcLg_NT10	100.0
α0.1_S_*_K3_C _NbRcLg_NT10	100.0
α0.01_S_*_K3_C _NbLge_NT10	100.0

Table 5.6: Coverage for the Logistics domain, restricted to the best four parameterized and non-parameterized domains.

Observing the planning time distribution in Figures 5.6 and 5.7, we notice that the impact of parameterization is similar to that observed in the Rovers domain: they perform better than the non parameterized ones and scale better to more complex instances.

Focusing now on the best domains, both parameterized and non parameterized, in Figure 5.6, we can make several finer observations:

- The best learned models (parameterized or not) outperform the learned model from the IPC.
- No model outperforms the reference handmade HTN.

Interestingly, we can also note that the best non-parameterized model was generated using the *Large* neighbour generation mode, whereas the best parameterized model was generated using

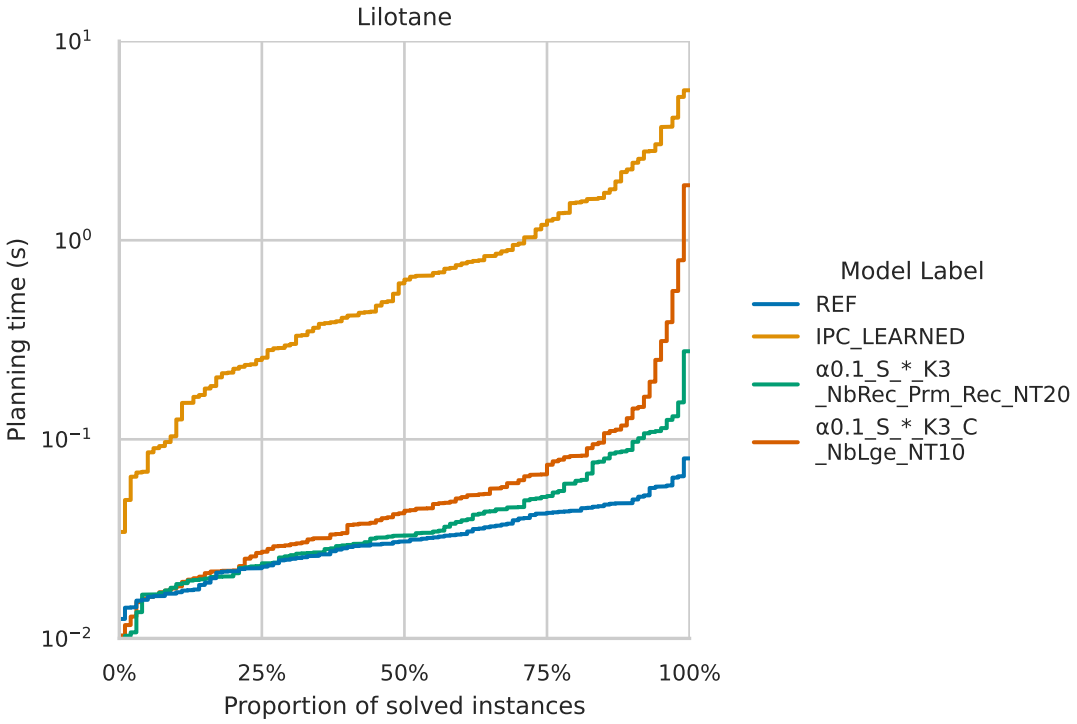


Figure 5.6: Planning time cumulative distribution for the LOGISTICS domain, for the best parameterized and non-parameterized domains.

the recursive neighbour generation[1]. This is not surprising: the *Large* generation will tend to favour models that are flatter, with highly sequential methods and therefore a low number of possible choices, even without a good parameterization. On the other hand, the models generated using the *Most Recursive* mode will often offer more freedom when not constrained by their parameters, thus increasing the search effort.

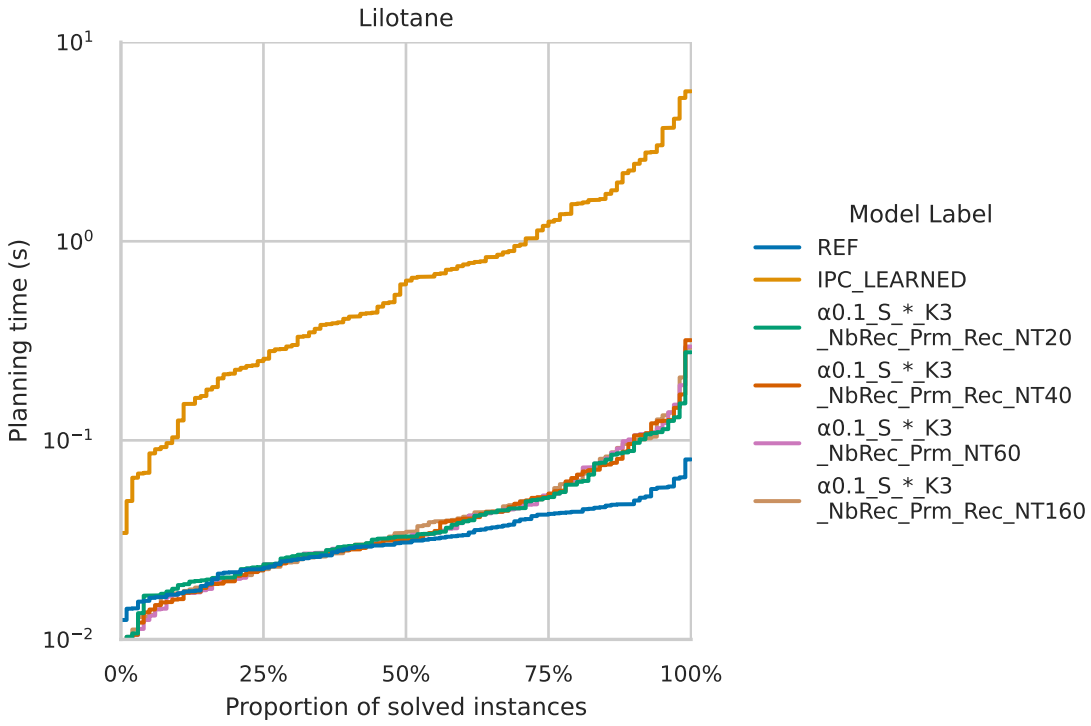
The performance of the best parameterized one (Appendix B, Figure D.5) appears to be due to its highly sequential structure, with little recursivity, which similarly to the ROVERS case, allows for numerous unification across the parameters and therefore good information propagation in the hierarchy.

We can also note that both our best learned domain tend to present a sharp increase in planning time for some specific instance of our test set. Analysing the raw data more finely shows that this spike is caused by a different instance for each learned model. They all are among the largest instances in our set, and therefore running the same tests on these models with more complex instances would be interesting to better assess the scaling behaviour of each model.

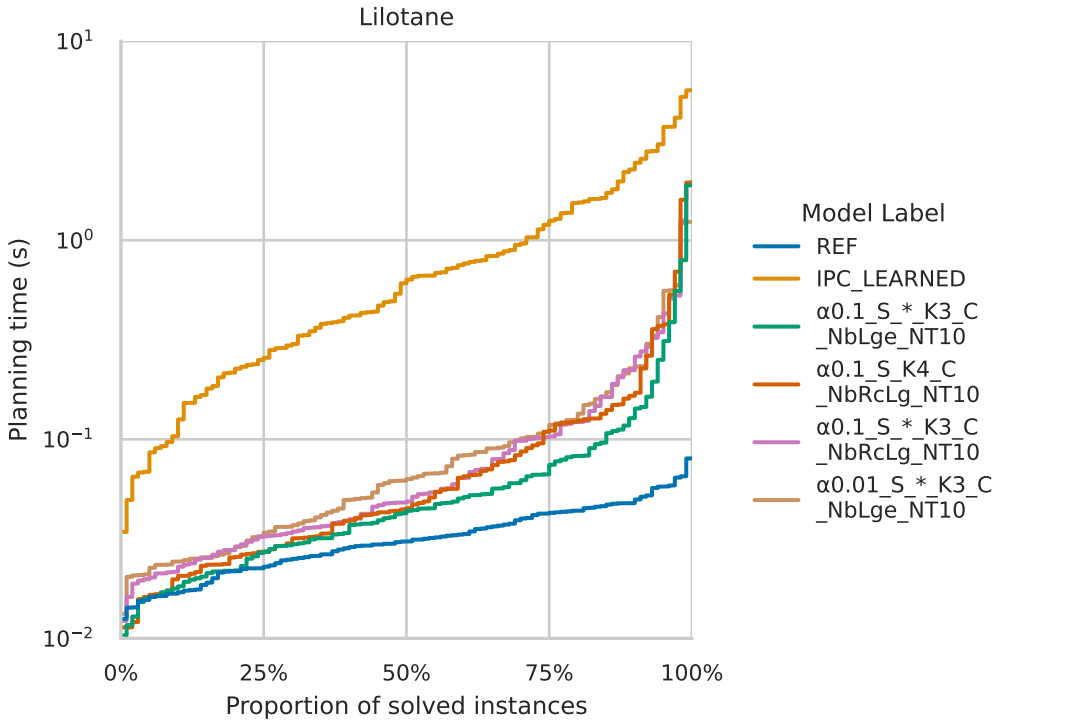
Let us now move on to the analysis of the plan length, comparing the two best domains (parameterized and non-parametrized) with the reference domains, as presented in Figure 5.8. We can observe that our best parameterized learned model consistently produces shorter plans than the IPC domain, mostly on par with the handmade domain, and that a similar behaviour can be observed for the non-parameterized domains. This is easily explained by their sequential structure which make it difficult to add useless actions in the plan.

Finally, let us analyse the impact of the number of training demonstrations on the per-

[1]This remains true even if we restrict ourselves to models learned with up to 10 demonstrations.



(a) Planning time distribution, limited to parameterized models.



(b) Planning time distribution, limited to non-parameterized models.

Figure 5.7: Planning time cumulative distribution for the Logistics domain, focusing on parameterized and non-parametrized HTNs.

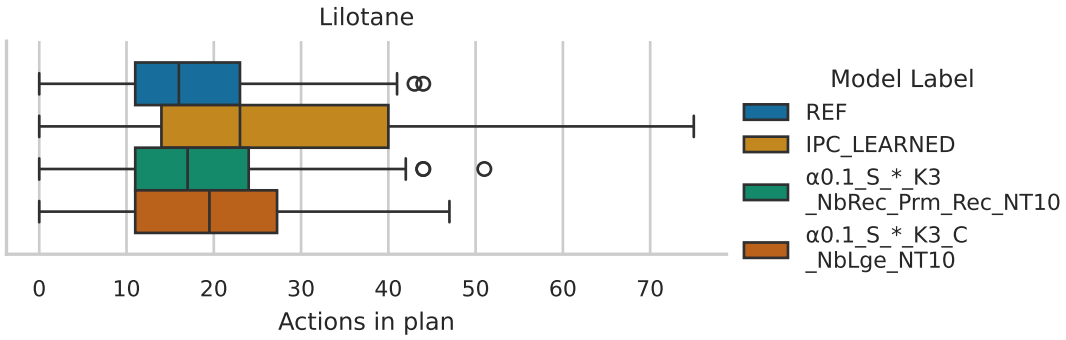


Figure 5.8: Distribution of the plan length over the set of commonly solved instances (100/100) in the Logistics domain.

formances of the learned models, presented in Figure 5.9. This figure shows us the structural simplicity of this domain: even though the global coverage of the learned models improves when going from 5 to 10 training demonstrations, we observe that at least one domain solves all the instances in every case, even in the non parameterized one.

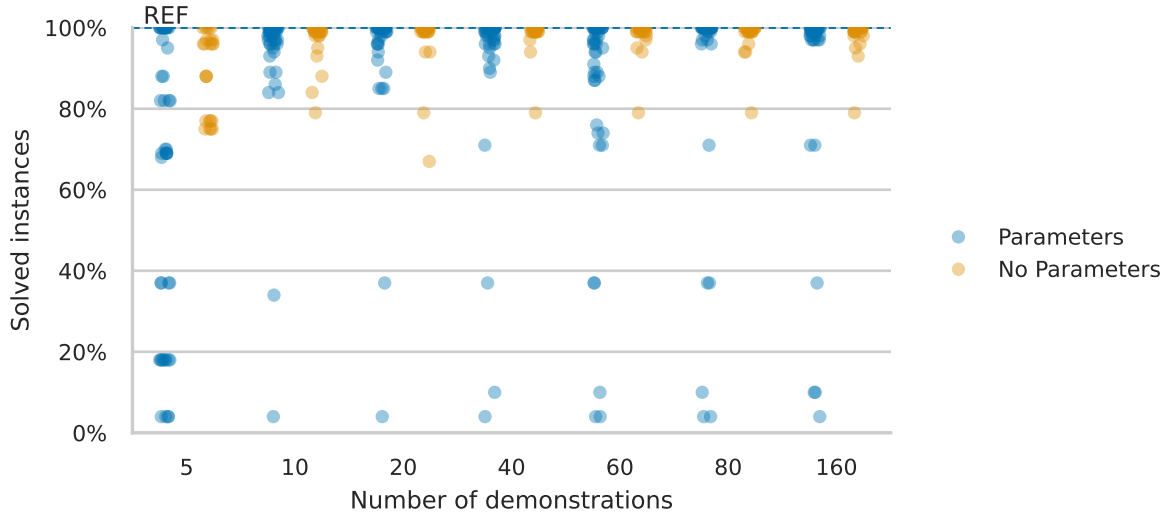


Figure 5.9: Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear.

5.4.3 Childsnack

Learning parameters and coverage results for the Childsnack domain are presented in Tables 5.7 & 5.8, while Figures 5.10 & 5.11 present the distribution of planning times for the different instances.

Once again, we can observe that parameterization is what allows the learned models to scale to more complex instances. Interestingly, we note that the best performing models, even parameterized are the one generated using modes that included the creation of *large* neighbours. This can easily be explained by the simple structure of the domain, where the original model only has two methods, which is a structure that can be more easily reached by generating all

Parameter	Possible Values
α	$0.001, 0.01, 0.1$
Filter Unused Methods	$\top$
Simplify Structure	$\bot, \top$
$*$	$\bot, \top$
Gen. Choice Patterns	$\bot, \top$
Seq. Pattern Max Length	$3, 4$
Planning	40s, 2500 MB

Table 5.7: Learning evaluation parameters for the CHILDSNACK domain.

Label	Solved Instances (%)
REF	85.0
α0.01_S_K4_C _NbLge_Prm_NT5	76.7
α0.1_S_*_K3 _NbRcLg_Prm_NT5	76.7
α0.01_S_*_K4_C _NbLge_Prm_Rec_NT5	76.7
α0.1_S_*_K4 _NbRec_Prm_Rec_NT80	76.7
α0.01_S_*_K3_C _NbRcLg_NT5	68.3
α0.001_S_K4 _NbLge_NT5	66.7
α0.1_S_*_K4_C _NbLge_NT5	66.7
α0.01_S_K4_C _NbLge_NT5	66.7

Table 5.8: Coverage for the CHILDSNACK domain, restricted to the best four parameterized and non-parameterized domains.

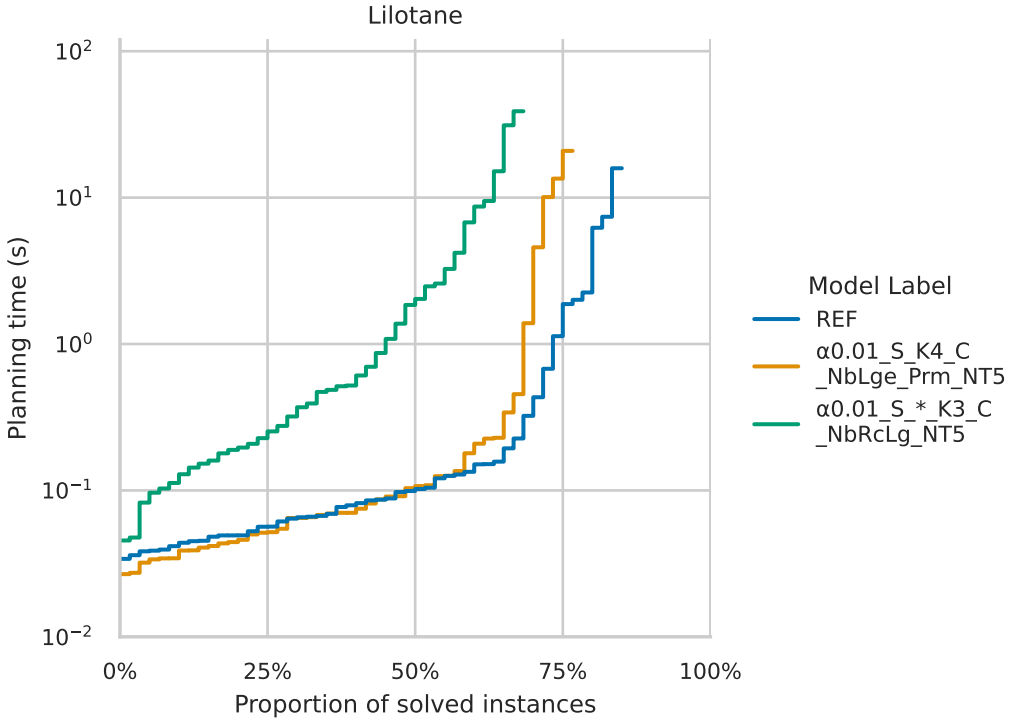
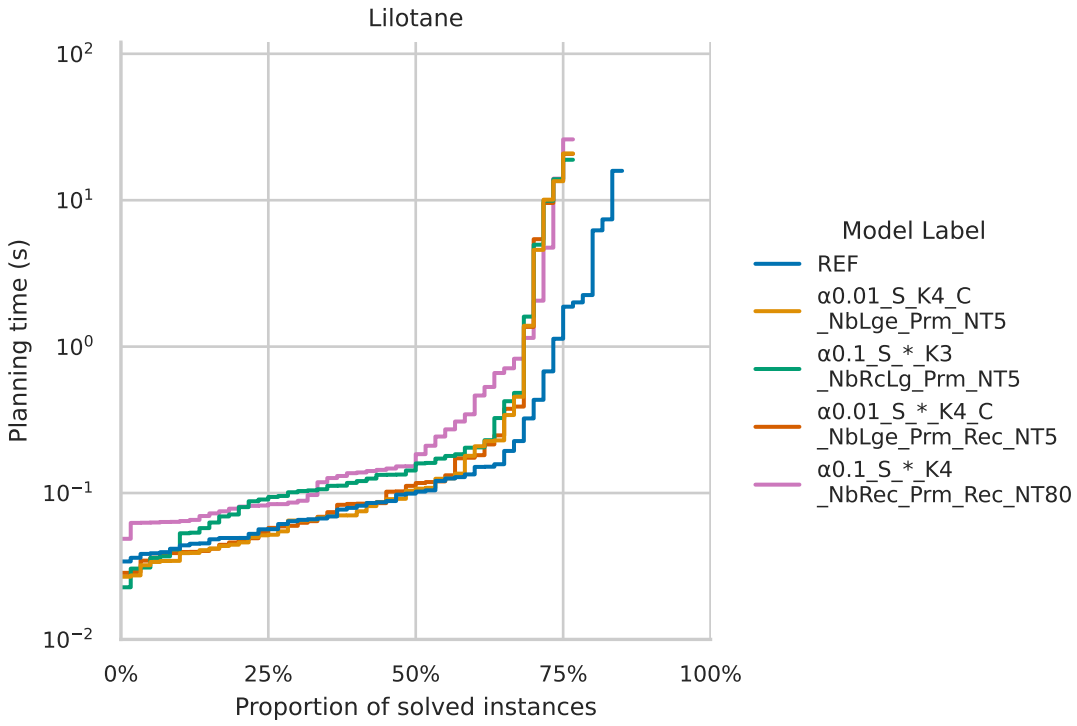
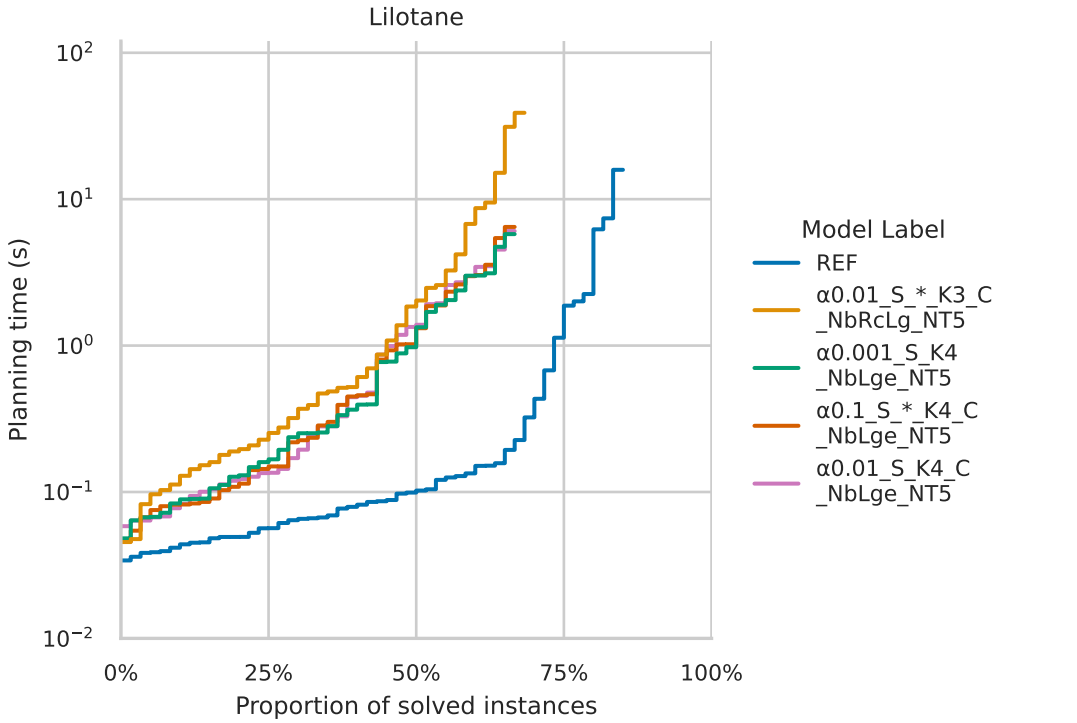


Figure 5.10: Planning time cumulative distribution for the CHILDSNACK domain, for the best parameterized and non-parameterized domains.



(a) Planning time distribution, limited to parameterized models.



(b) Planning time distribution, limited to non-parameterized models.

Figure 5.11: Planning time cumulative distribution for the CHILDSNACK domain, focusing on parameterized and non-parametrized HTNs.

the (two) possible structures in the demonstrations, rather than building the model abstraction after abstraction step.

Note that we are able to attain *exactly* the structure of the original domain (Appendix D, Figure D.7). Because we have the same structure and still do not scale as well as the reference domain, we conjecture that this difference is due to the lack of method preconditions in our learned domains.

We do not present the effect of the training instances on the planning performance for this domain, as it is identical for all sets of demonstrations due to the simplicity of the domain.

5.4.4 Satellite

Learning parameters and coverage results for the Satellite domain are presented in Tables 5.9 & 5.10. Figures 5.12 & 5.14 present the distribution of the planning times over the instances.

Parameter	Possible Values
α	$0.001, 0.01, 0.1$
Filter Unused Methods	$\top$
Simplify Structure	$\bot, \top$
$*$	$\bot, \top$
Gen. Choice Patterns	$\bot, \top$
Seq. Pattern Max Length	$3, 4$
Planning	30s, 2500 MB

Table 5.9: Learning evaluation parameters for the Satellite domain.

Label	Solved Instances (%)
`REF`	60.0
α`0.1_S_*_K4_C _NbRec_Prm_NT80`	70.0
α`0.1_S_*_K4_C _NbRec_Prm_Rec_NT80`	70.0
α`0.01_S_K4_C _NbRec_Prm_Rec_NT20`	67.5
α`0.01_S_K4 _NbRec_Prm_NT20`	67.5
α`0.1_S_K3_C _NbRec_NT80`	40.0
α`0.1_S_K3 _NbRec_NT40`	40.0
α`0.001_S_*_K3_C _NbRec_NT80`	40.0
α`0.001_S_*_K4_C _NbRcLg_NT5`	40.0

Table 5.10: Coverage for the Satellite domain, restricted to the best four parameterized and non-parameterized domains.

We note once again that parameterization is required to scale to more complex instances.

However, we note that Satellite is also a very simple domain to learn, with as little as 5 demonstrations being required to learn a model able to outperform the reference domain from the IPC.

The dip in performance observed when learning with 10 demonstrations is due to an issue with training demonstration set generation, where one task (`do_turning`) had no demonstrations in the resulting set. It can therefore safely be ignored.

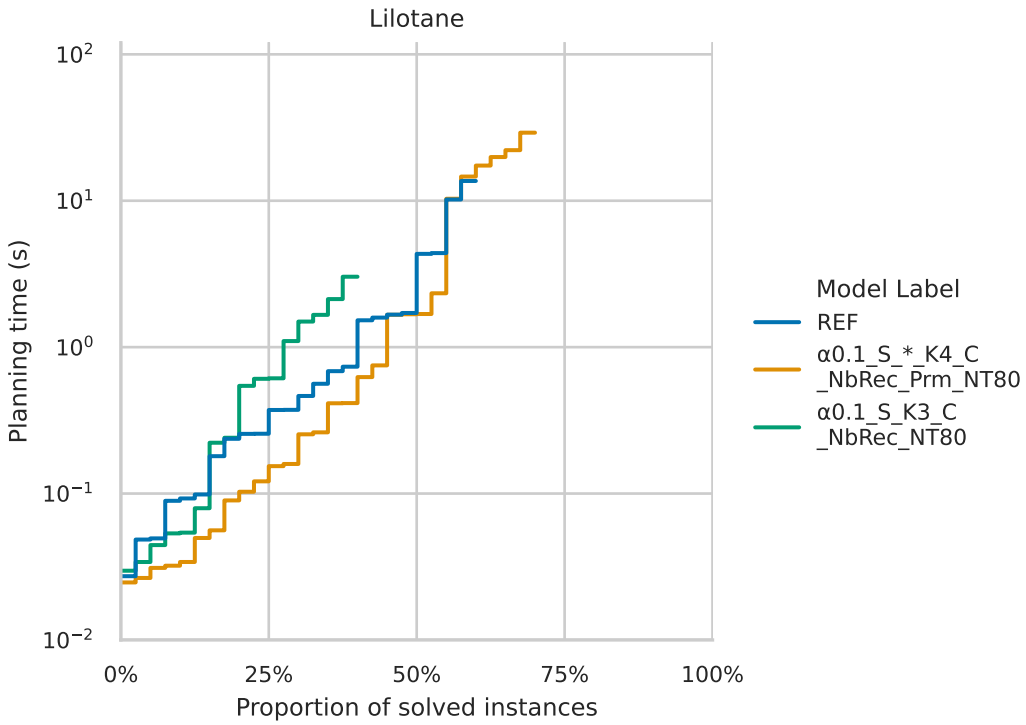


Figure 5.12: Planning time cumulative distribution for the SATELLITE domain, for the best parameterized and non-parameterized domains.

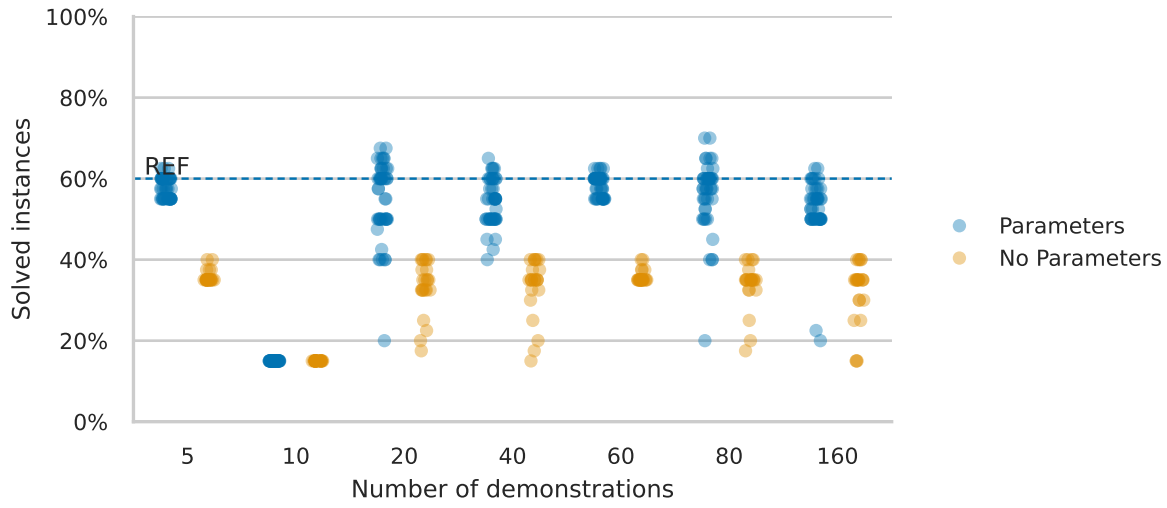
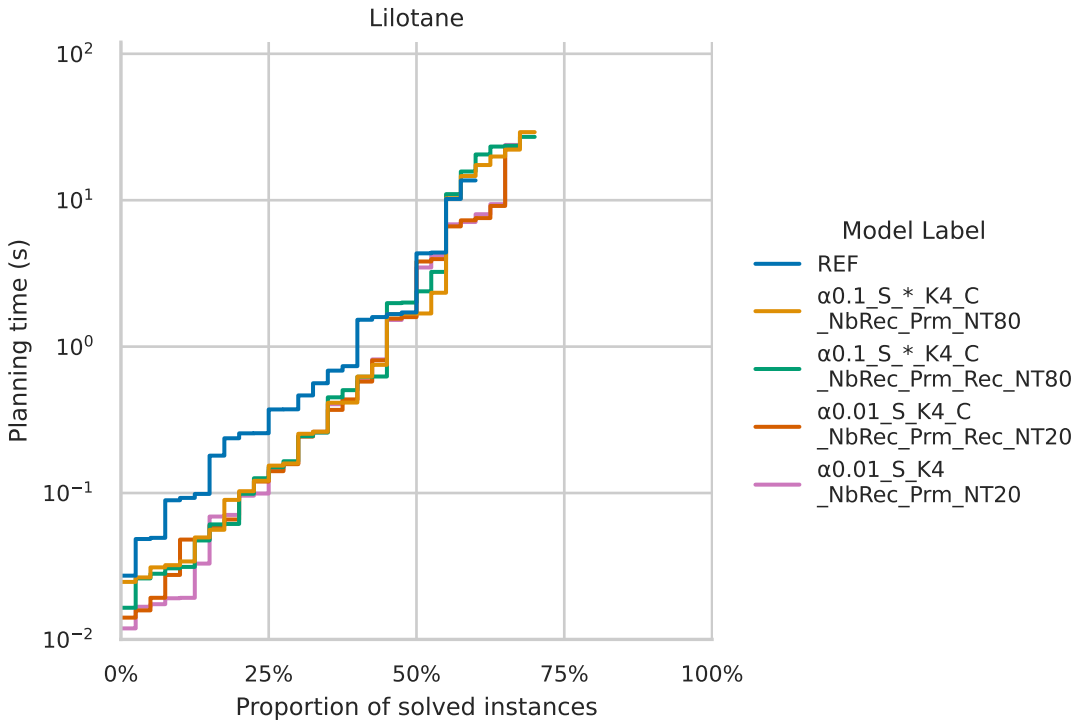
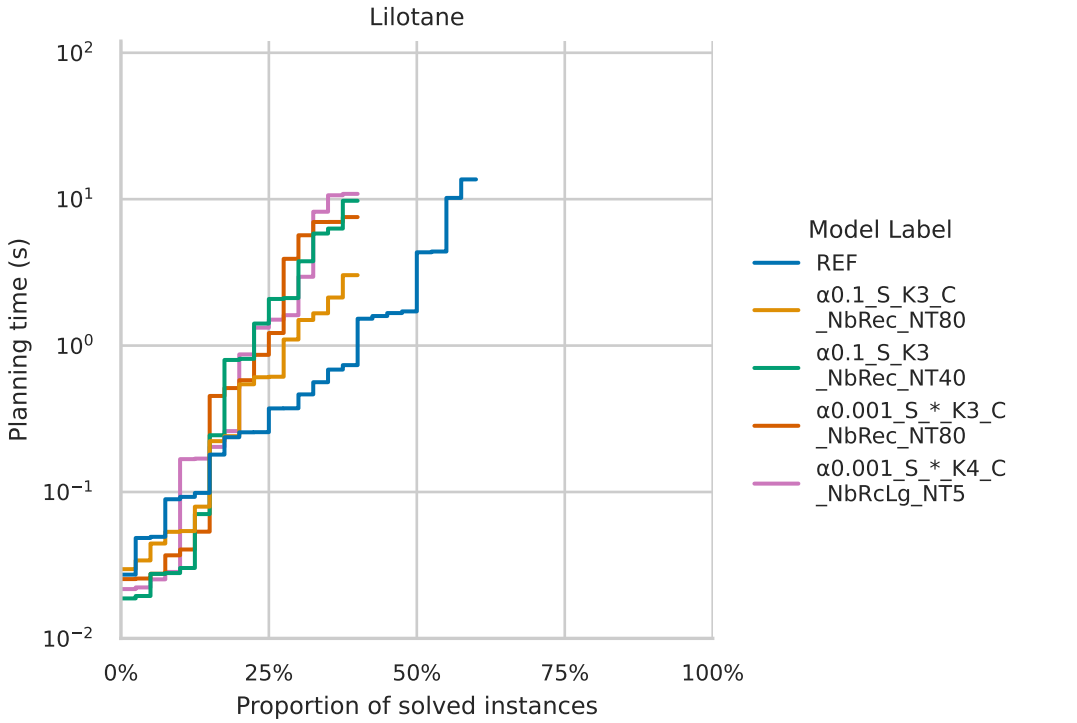


Figure 5.13: Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear.



(a) Planning time distribution, limited to parameterized models.



(b) Planning time distribution, limited to non-parameterized models.

Figure 5.14: Planning time cumulative distribution for the SATELLITE domain, focusing on parameterized and non-parametrized HTNs.

5.4.5 Woodworking

Learning parameters and coverage results for the Woodworking domain are presented in Tables 5.11 & 5.12, while Figures 5.15 & 5.17 present the distribution of planning times for the different instances.

Due to identical results for parameterizing with recursion detection or without it in this domain, we only show the best models for the one without direct recursion detection, in order to showcase more diverse models.

Parameter	Possible Values
α	$0.0001, 0.001, 0.01$
Filter Unused Methods	$\top$
Simplify Structure	$\top$
$*$	$\bot, \top$
Gen. Choice Patterns	$\bot, \top$
Seq. Pattern Max Length	$3, 4$
Planning	60s, 2500 MB

Table 5.11: Learning evaluation parameters for the Woodworking domain.

Label	Solved Instances (%)
REF	72.4
α0.0001_S_*_K3_C _NbRec_Prm_NT20	75.9
α0.001_S_K3_C _NbRec_Prm_NT20	75.9
α0.01_S_K4 _NbRec_Prm_NT40	72.4
α0.0001_S_*_K4 _NbRcLg_Prm_NT10	72.4
α0.0001_S_*_K4_C _NbLge_NT10	36.2
α0.001_S_K4_C _NbLge_NT10	34.5
α0.001_S_K4_C _NbRcLg_NT10	34.5
α0.01_S_K3 _NbRec_NT10	32.8

Table 5.12: Coverage for the Woodworking domain, restricted to the best four parameterized and non-parameterized domains.

We again obtain similar results as in the other domains, with parameters being necessary to scale to more complex instances.

Observing the coverage compared to the number of training demonstrations (Figure 5.16), we observe that we reach the performance of the reference domain with as little as 10 demonstrations.

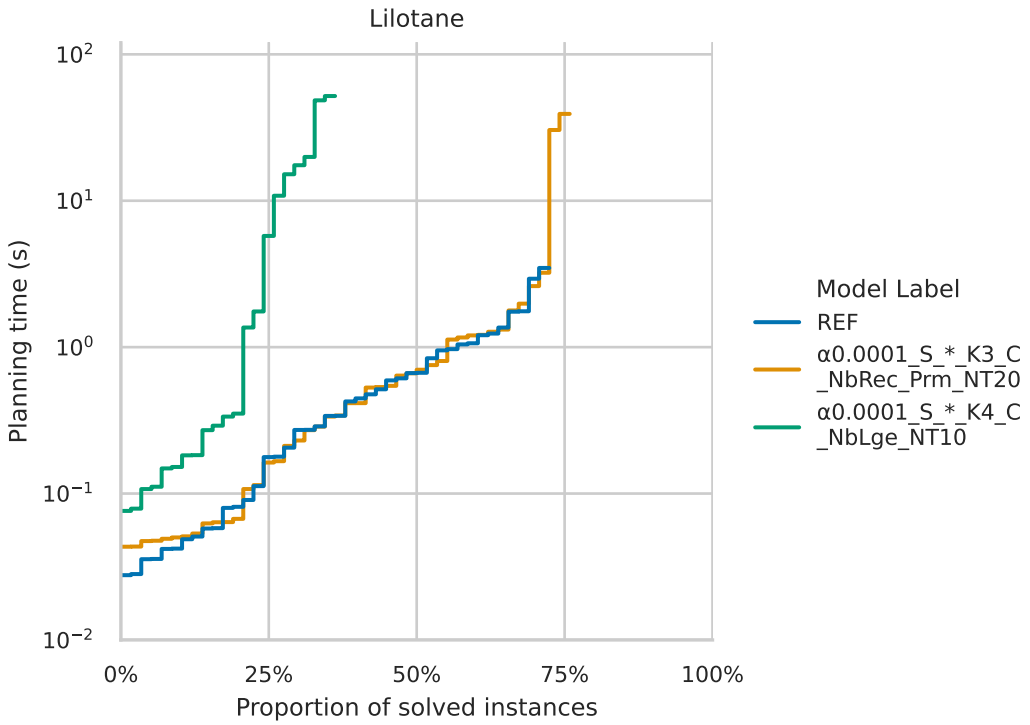


Figure 5.15: Planning time cumulative distribution for the WOODWORKING domain, for the best parameterized and non-parameterized domains.

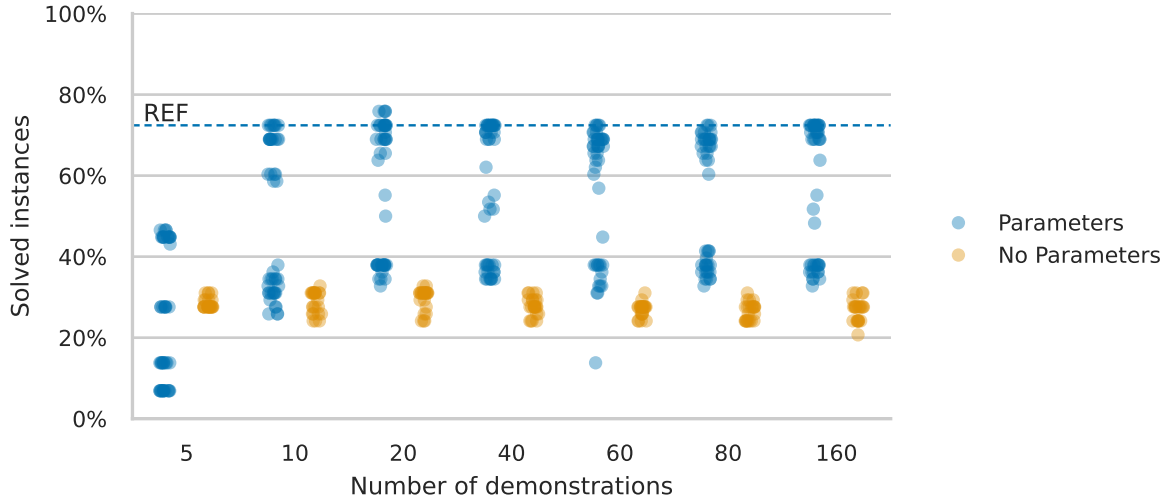
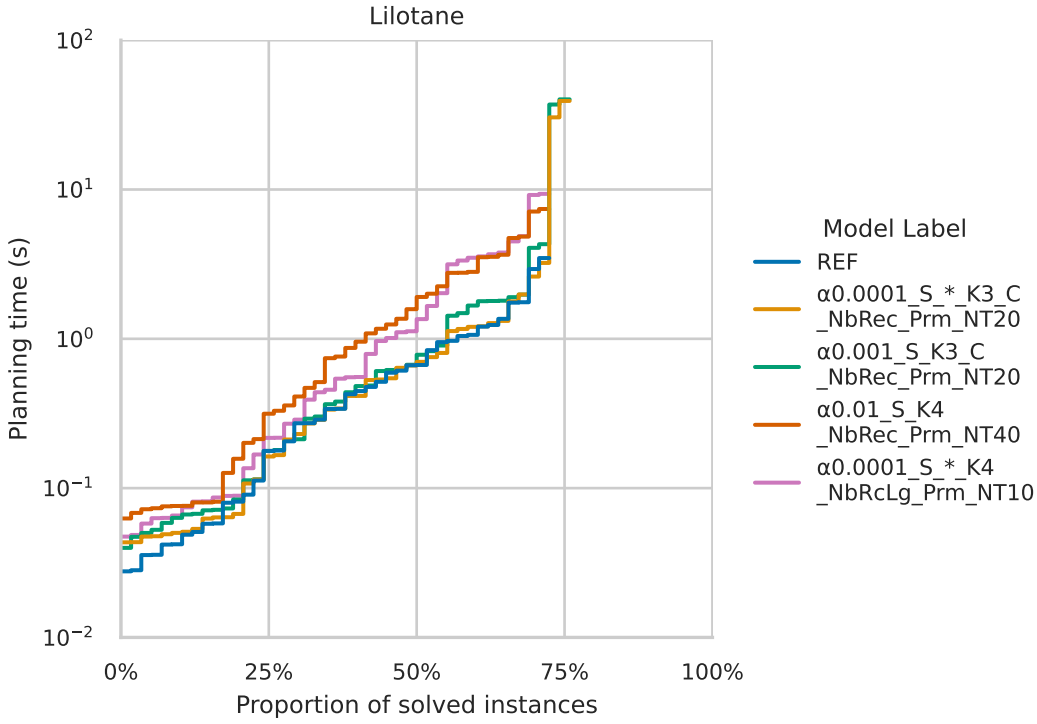
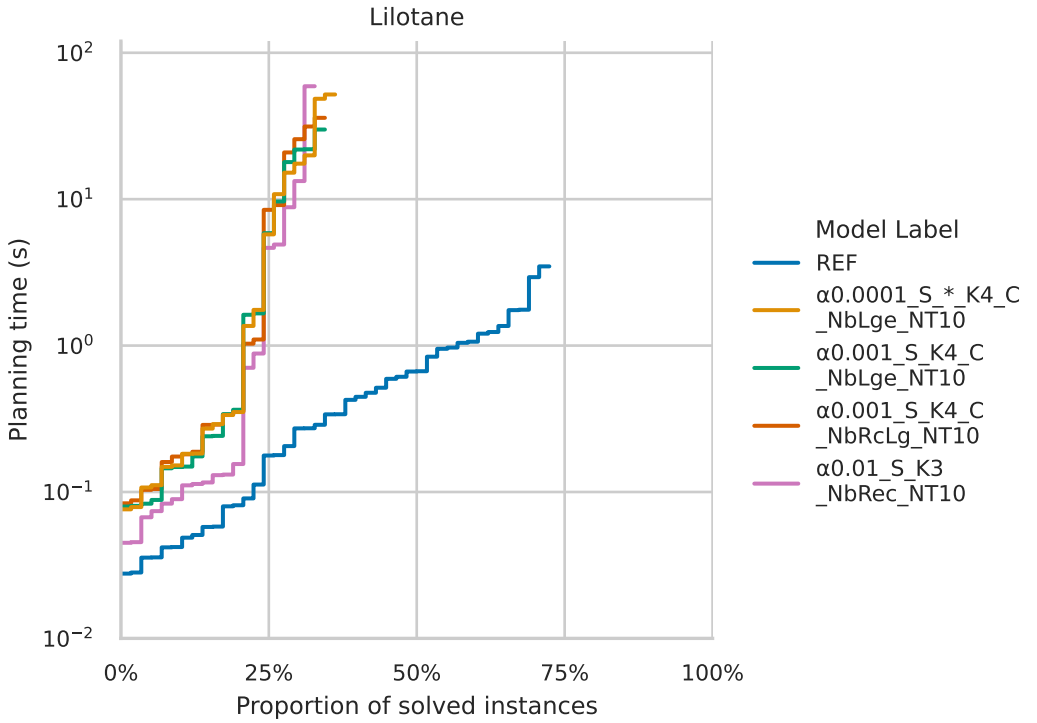


Figure 5.16: Scatter plot of the coverage against the number of training demonstrations. Note that the x-axis scale is not linear.



(a) Planning time distribution, limited to parameterized models.



(b) Planning time distribution, limited to non-parameterized models.

Figure 5.17: Planning time cumulative distribution for the WOODWORKING domain, focusing on parameterized and non-parametrized HTNs.

5.5 Learning Times

Figures 5.18 & 5.19 present the structure learning and parameterization times for every domain, for the recursive neighbour generation mode only. Figures 5.20 & 5.21 present the same results but for every neighbour generation mode where applicable. Because the ROVERS domain has at most 74 demonstrations, we grouped it with the 80 demonstrations results of the other domains for comparison.

The first insight that can be obtained from these results is that *structure* learning is a lot more expensive than *parameter* learning. Furthermore, the structure learning and parameterization times appear loosely correlated: domains with a difficult to learn structure tend to require more time for parameterization.

More qualitatively, the difficulty to learn a given domain structure is linked to the:

- The number of candidate HTN structures that can be generated during the search.
- The difficulty of matching these candidate structures to the demonstrations.

This possibly explains why the LOGISTICS domain is more complex to learn even though the ROVERS final domain is more complex: the large number of optional actions in the LOGISTICS lead to a large number of possible matchings (without parameters) compared to the more focused structure in the ROVERS domains.

Focusing now on the *large* neighbour generation mode learning times, we observe that this mode is often ten times slower than the *recursive mode*. As it only appears useful when we have very simple domain structures, and as even in these cases the *recursive* mode comes close to its performance, we argue that this cost is not worth it.

Globally, the learning times are relatively fast: less than an hour in the worst case, often in the range of a few minutes for complex domains with numerous demonstrations. This could be taken advantage of in order to search over the space of hyperparameters, learning multiple models and then evaluating them on a set of test instances.

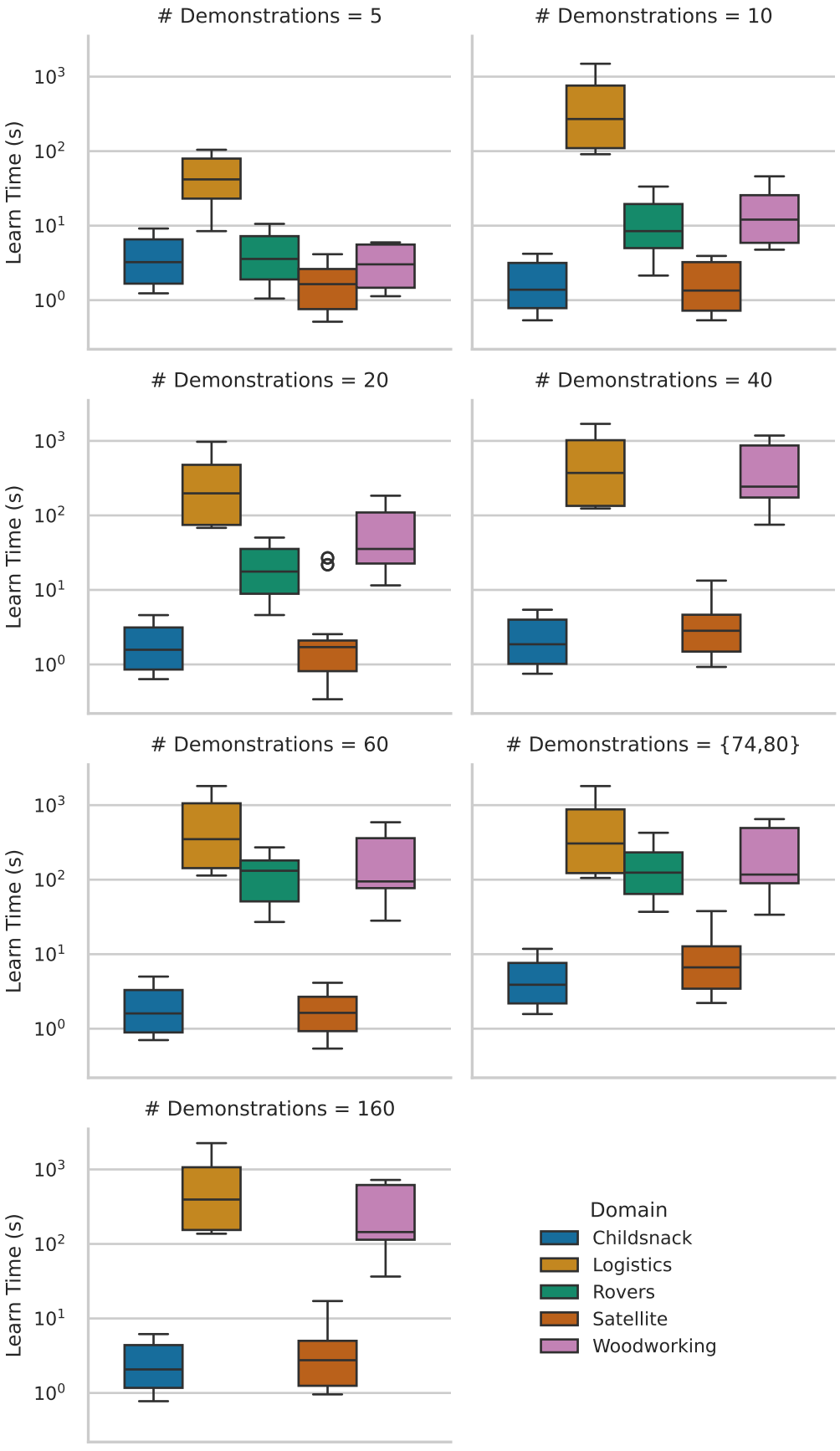


Figure 5.18: Structure learning time for all the tested domains, limited to the *most recursive* neighbour generation.

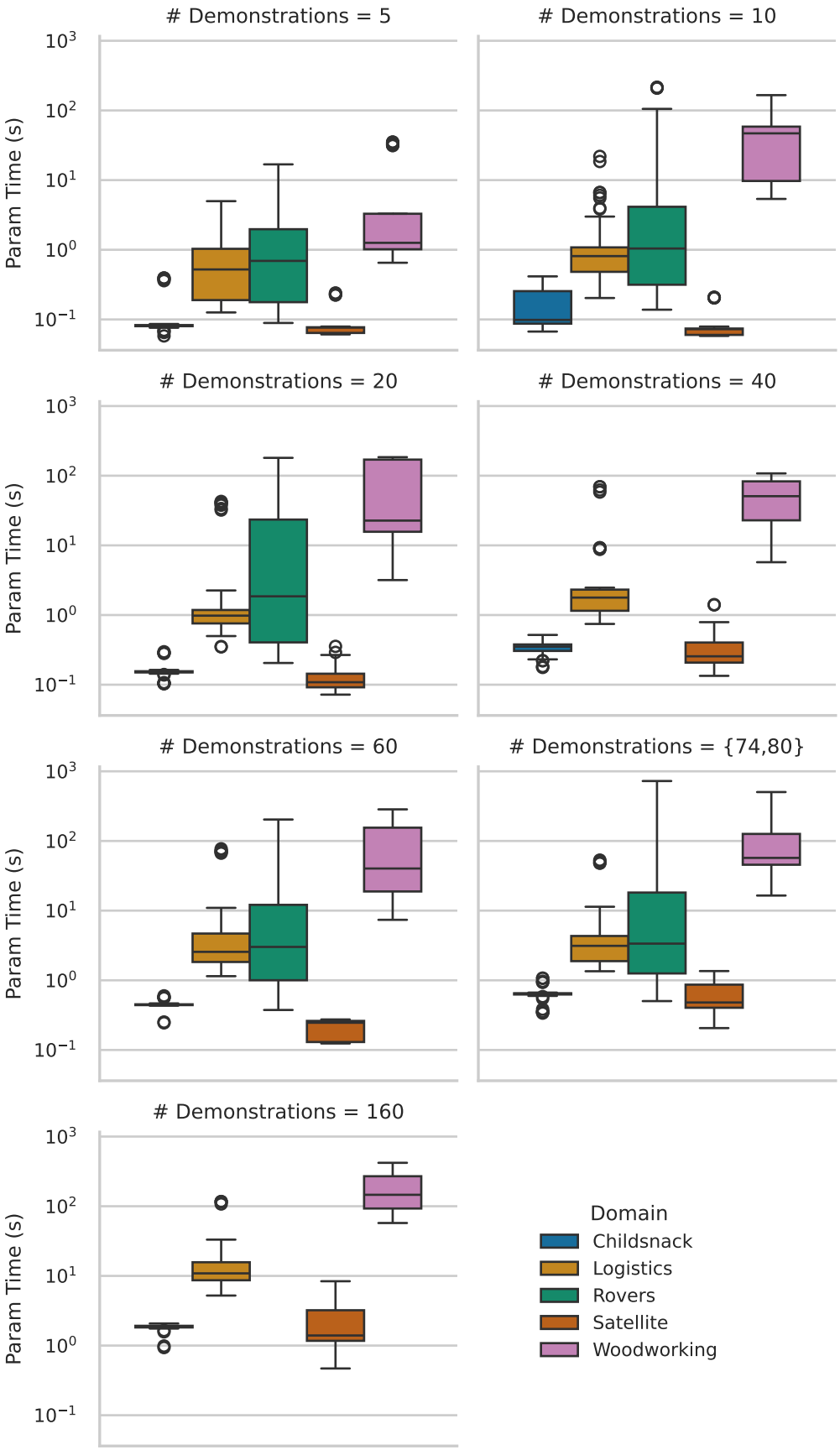


Figure 5.19: Parameterization time for all the tested domains, limited to the *most recursive* neighbour generation.

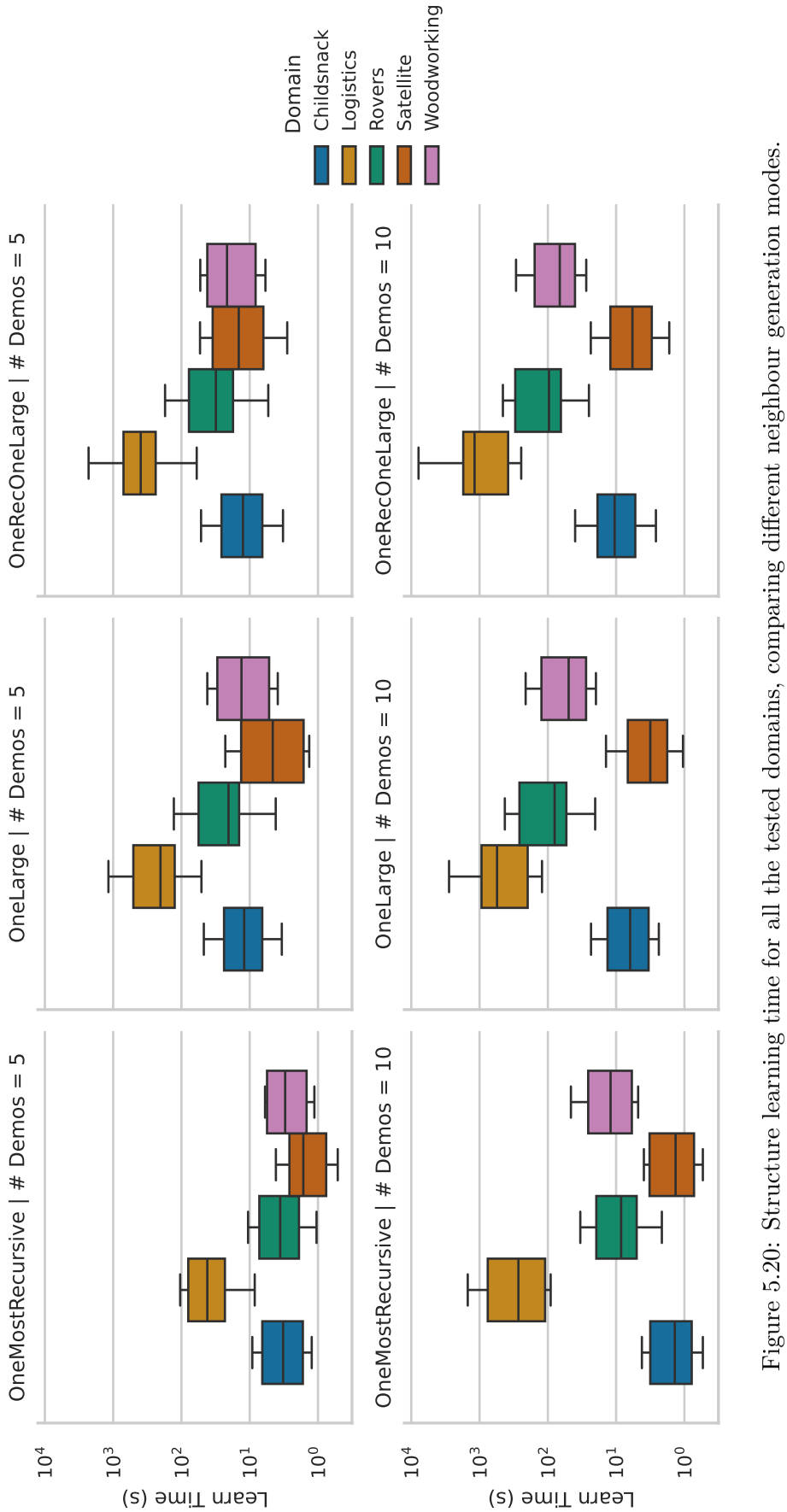


Figure 5.20: Structure learning time for all the tested domains, comparing different neighbour generation modes.

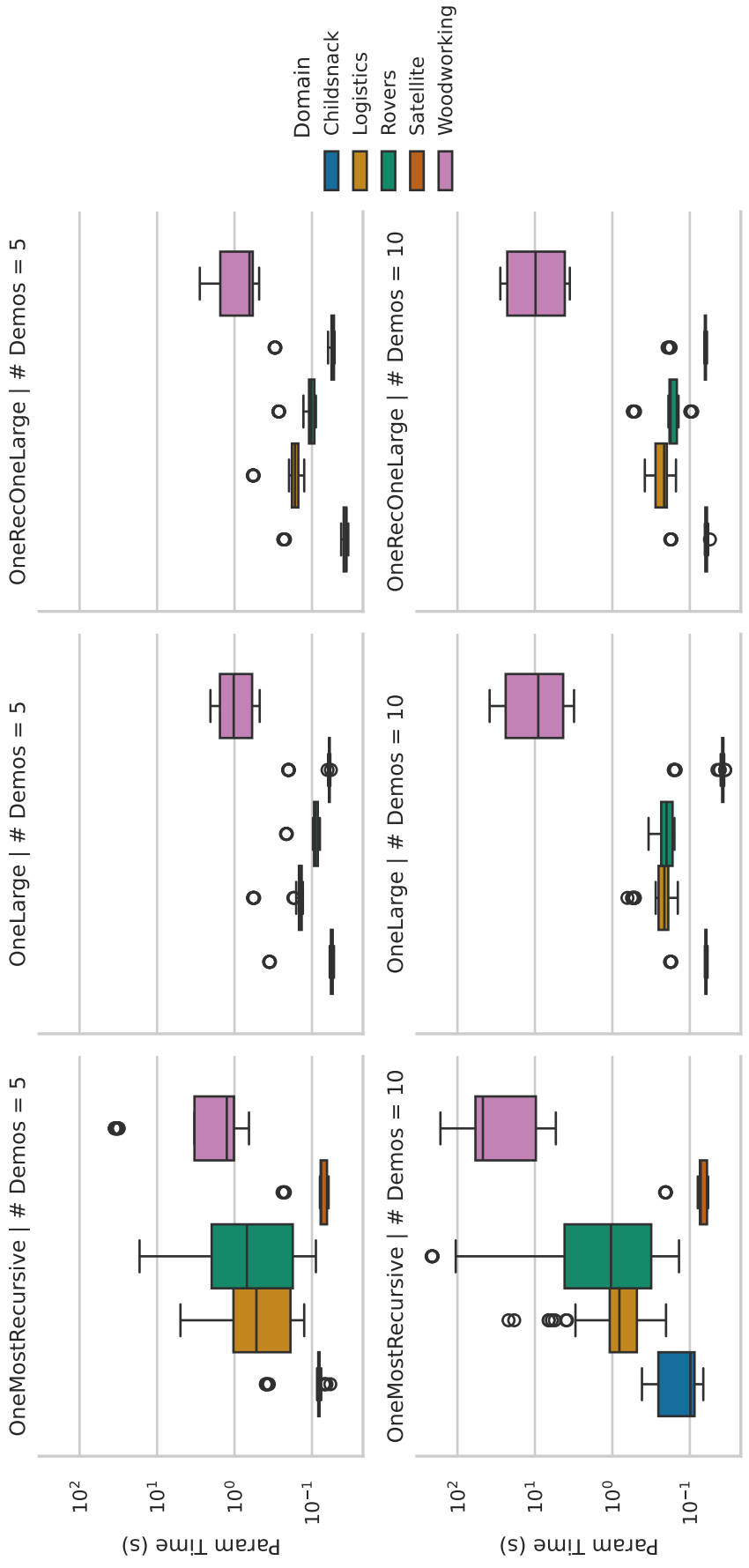


Figure 5.21: Parameterization time for all the tested domains, comparing different neighbour generation modes.

5.6 Conclusion

In this chapter, we have analysed the performance of our learned domains in five well-known planning domains, comparing them with handmade domains that include advice to guide the planner as well as a domain obtained from another learning approach in the case of the LOGISTICS domain.

The main insights that we can obtain from these experiments are as follows:

- Our approach to learning from demonstrations allows learning models that are competitive with the state of the art in largely reasonable times and with a small set of demonstrations.
- A good parameterization is important for the model to scale to more complex instances, however parameterizing a low quality model may have the opposite effect.
- As was expected when developing our Minimum Description Length (MDL)-based metric, model quality weight should be given less importance compared to the demonstration cost.

While these results are highly encouraging, a more in-depth analysis would be interesting. First, the analysis should be run on the same domains under more generous computational limits and with even larger instances, in order to better assess the scalability capabilities of the learned domains. Furthermore, we would like to include more domains in our analysis to better assess the generalizability of our observations. Finally, it would be instructive to study in more details the impact of each learning hyperparameter on the quality of the resulting models.

Note also that, we have only presented the evaluation of our learned models using a single planner, namely Lilotane [Sch21], as it was used in other parts of our pipeline for its optimization feature. Trying to evaluate the learned models with another planner (HyperTensioN [MMdS21]), we obtained abysmal results on every domain but SATELLITE, where we easily outperformed the reference model, even more so than with Lilotane. However, the fact that even on the CHILDSNACK domain we had poor performances – even though we learned the same structure, with similar parameters but without preconditions – tells us that this kind of models does not work well with this planner. A more in-depth analysis of this planner-dependent performance would be interesting, first to improve our learner, but also possibly to help assess the impact of different kind of models on the performances of different HTN planners.

Chapter 6

Conclusion

In this thesis, we have developed a system for learning Hierarchical Task Networks (HTNs) from a small set of demonstrations. We have presented how such a system can be split into two parts, separating the learning of the *structure* of the HTN from its *parameters*.

In order to build such a system, we developed a novel metric based on the Minimum Description Length (MDL) principle to use as an efficient proxy for planning performance. This allowed us to develop a greedy search algorithm leveraging frequent pattern mining, which we coupled with the HTN-Maker learning algorithm to learn HTN structures.

On the topic of parameter learning, we proposed a system to learn a sensible parameterization of an HTN, provided we know at least the definition of its subtasks. This procedure is based on argument propagation from the subtasks, to generate a set of parameters that *can* be useful for constraining the search space of planner, and a MAX-SMT approach for argument unification, where the unification allows to actually transfer information across the hierarchy to guide the planner's search.

Future Works Several possibilities for extending this work come to mind.

Firstly, the structure search relies on a set of generated patterns to generate good candidate HTNs. This generation is for now done using exhaustive generation over a set of specific patterns, which was mainly designed to accommodate implementation constraints. It would be interesting to explore the possibility of generating this set of patterns in a smarter way, in order to balance the possibly generated HTNs and the computational cost of learning, for example by leveraging information given by the arguments of the actions. Furthermore, considering more expressive patterns may pave the road towards better learned models.

It should also be noted that the bottleneck of the structure learning algorithm is the cost of matching candidate structures and demonstration traces. Given the recent success of parsing-based approaches for plan recognition, these may be better suited to this goal than the current planning-based approaches.

Secondly, regarding parameter learning, while the proposed approach is correct and usable in the presence of recursions in the HTN, the resulting parameterization is of low quality and may hinder a planner more than help it. While we have provided potential pointers on how to better handle recursive structures, this remains an open problem. Furthermore, the parameters could be used to extract method pre-conditions, which may improve the performance of the learned domains.

Finally, we have only considered totally-ordered, deterministic domains. While our algorithm should easily be applicable to non-deterministic domains, thanks to its ability to detect looping constructs and its ability to build decomposition trees, the absence of method pre-conditions and task post-conditions to assess a decomposition's success may lead to poorly performing models in the current state. The extension to partially-ordered domains appears trickier, as the demonstrations may contain interleaved actions from other tasks, and could therefore be an interesting research topic.

Annexe A

Résumé en Français

Pour agir de façon délibérée dans leur environnement, les agents autonomes ont généralement besoin d'effectuer des tâches de haut niveau en s'appuyant sur un ensemble de *skills* leur permettant d'effectuer des actions dans leur environnement.

Chaque skill représente une opération élémentaire, comme attraper un objet ou le déposer. Ces skills peuvent potentiellement abstraire les plus basses couches des primitives de contrôle de l'agent (comme les commandes envoyées aux moteurs dans le cas d'un robot). Il ne s'agit donc pas nécessairement d'une séquence fixe de ces primitives. En effet, il peut être nécessaire qu'un skill s'adapte à son contexte d'exécution : à titre d'exemple, nous pouvons considérer utiliser différents mouvements pour attraper une tasse vide ou bien un bol rempli de liquide. Ces skills doivent ensuite être combinés afin de réaliser les comportements de haut niveau souhaités.

Pour donner un exemple de tâche de haut-niveau qui peut être réalisée par une combinaison de skills, considérons un agent voyageant de Paris à Toulouse pour se rendre à une conférence. Deux options sont disponibles : voyager en train ou en avion. Nous pouvons définir l'ensemble de skills suivant pour notre agent : {`BuyTrainTckt`, `BuyPlaneTckt`, `WalkTo`, `TakeCab`, `TakeTrain`, `TakePlane`}. Ces skills peuvent être paramétrisés afin de spécifier leur contexte d'application, par exemple la destination pour le skill `WalkTo`.

Nous pouvons donc imaginer choisir la séquence de skills suivante pour atteindre notre but de haut niveau, à savoir se rendre sur le site de la conférence à Paris :

$$\langle \texttt{BuyTrainTckt}(Paris), \texttt{WalkTo}(Station), \texttt{TakeTrain}(Paris), \texttt{TakeCab}(Venue) \rangle$$

S'il est possible de simplement combiner les skills de notre agent de manière *réactive* pour obtenir une telle séquence, en utilisant par exemple de simples politiques comme des machines à états, des réseaux de neurones ou des *behaviour trees*, cette approche peut s'avérer inefficace dans le cas où il est nécessaire de considérer un horizon éloigné pour effectuer une certaine tâche.

Dans notre exemple de voyage, certaines étapes nécessitent de planifier la suite des actions avant de les exécuter : par exemple, utiliser l'action d'acheter un billet de train nécessite de connaitre certaines futures actions, comme savoir que l'on ne prendra pas ici l'avion pour se rendre à la conférence. Nous nous plaçons donc dans le paradigme de l'*action délibérée*, telle que présenté par Ghallab, Nau et Traverso [GNT14]. Dans ce cadre, les auteurs définissent une action comme quelque chose que l'agent effectue pour changer son état ou son environnement, tandis que la délibération est un processus de raisonnement qui mène l'agent à choisir une action plutôt qu'une autre en considérant ses buts à long terme, souvent au travers de techniques de *planification*.

Dans l'absolu, nous pouvons considérer un acteur interagissant avec son environnement de la manière présentée en Figure A.1. Ici, nous pouvons séparer une *plateforme d'exécution* et un composant de *délibération*. Cette plateforme a pour rôle de transformer les commandes des skills venant du composant de délibération en primitives de l'acteur pour permettre d'exécuter ces commandes dans son environnement. Elle est aussi chargée de convertir les données brutes

des capteurs en représentations utilisables par les fonctions de délibération.

Le composant de délibération peut être divisé en deux sous-composantes : *planification* et *action*. Le composant de planification principalement reçoit les activités de haut-niveau à réaliser et génère des stratégies à long terme pour les mener à bien, dans le but de guider le composant d'action. Il considère généralement un environnement abstrait. Le composant d'action est chargé de réaliser l'exécution des stratégies générées par la planification, surveillant l'exécution des actions pour faire face aux événements extérieurs et aux échecs dans l'exécution des skills.

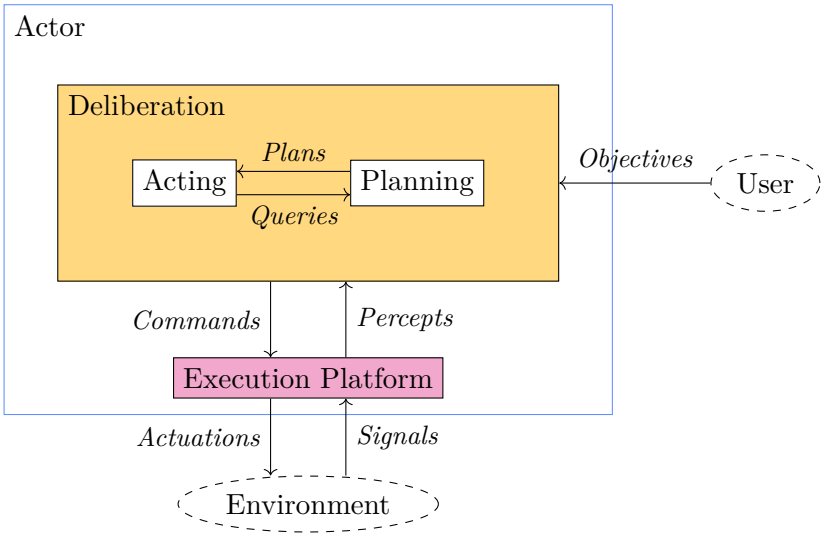


FIGURE A.1 : Une vue simplifiée de l'architecture d'un acteur, adapté de [GNT14].

Notons que les architectures réellement utilisées pour un acteur sont généralement plus complexes. Par exemple, l'architecture présentée par LEMAIGNAN et al. [Lem+17], possédant plusieurs composants dans la couche de délibération. Nous pouvons toutefois toujours distinguer les deux couches, ainsi que les composants de planification et d'action.

En observant différents systèmes d'exécution pour la robotique [DI00 ; Ing+96 ; MCA22 ; SdSP06 ; TB22], nous pouvons noter qu'ils s'appuient sur des *modèles* des tâches et des actions à réaliser. Ces modèles sont bien souvent de nature hiérarchique, de façon à décrire efficacement des tâches complexes. Réutilisant notre exemple de voyage, des modèles des skills pourraient être utilisés pour spécifier leurs conditions d'applicabilité et leurs effets. Considérons par exemple notre skill `WalkTo`(*Station*). Il pourrait être défini de manière à n'être applicable seulement s'il existe un cheminement piéton pour se rendre à la gare, et que celle-ci soit située à moins de trois kilomètres du point de départ de notre agent. L'effet de ce skill serait défini tel que notre agent se trouve à la gare à la fin de son exécution. Si nous voulions un modèle de notre tâche de haut niveau, nous pourrions considérer une structure telle que présentée en Figure A.2, dans laquelle nous avons deux options différentes pour réaliser la tâche, à savoir voyager en train ou en avion.

Des formalismes hiérarchiques ont été développés pour décomposer une tâche (ou un but) de haut niveau en skills disponibles pour un agent. Ces modèles sont cependant complexes à définir manuellement. Si l'apprentissage pour les agents autonomes s'est grandement développé ces dernières années, la plupart des approches restent focalisées sur la couche d'exécution [CSL21 ; Kle+20]. En comparaison, l'apprentissage de modèles pour des tâches complexes de haut niveau n'a pas reçu la même attention. L'apprentissage de ces modèles sera le cœur de cette thèse,

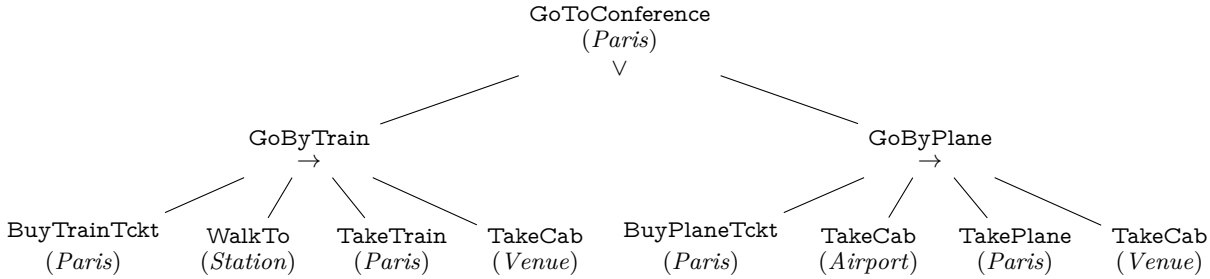


FIGURE A.2 : Un domaine hiérarchique simplifié pour la tâche de haut niveau de notre exemple de voyage.

spécifiquement dans le contexte suivant :

- *Disponibilité des skills* : nous supposerons que l'ensemble des capacités primaires de l'agent (skills) est connue. Ces skills peuvent être appris par des approches de l'état de l'art, ou bien programmés manuellement.
- *Basé démonstration* : nous supposerons qu'un ensemble de démonstrations de la manière d'effectuer la tâche à apprendre est donné à l'agent. Ces démonstrations prendront la forme de séquences de skills, montrant une combinaison donnée de ceux-ci réalisant la tâche à apprendre dans un certain état.

Nous nous focaliserons spécifiquement sur l'apprentissage de modèles hiérarchiques, ceux-ci étant plus aisés à comprendre pour des experts humains. De plus, ils permettent de contraindre l'espace – potentiellement immense – de tous les plans possibles qu'un agent pourrait choisir de suivre, tout en permettant de reproduire les comportements démontrés. De plus, nous nous placerons dans un contexte où les démonstrations sont couteuses à obtenir, comme dans le cas de l'apprentissage pour des agents robotiques, ce qui nous orientera vers des approches demandant peu de données.

Présentons désormais, de façon succincte, l'approche développée dans le cadre de cette thèse, focalisées sur l'apprentissage de réseaux de tâches hiérarchiques (Hierarchical Task Networks, HTN).

L'Algorithme A.1 présente la procédure globale. À partir d'un HTN de base (donné en entrée ou généré automatiquement), nous générons un voisinage de structures de HTN similaires, avant de les évaluer pour obtenir la meilleure. Ce processus est répété jusqu'à ce que les structures obtenues cessent de s'améliorer. Une fois la meilleure structure obtenue, nous optimisons ces paramètres pour améliorer ses capacités de planification. Chacune des étapes est détaillée ci-après, de manière à faire ressortir les contributions principales de cette thèse.

Algorithm A.1 Processus global d'apprentissage de HTN

$\mathcal{H}^* \leftarrow$ GENERER HTN DE BASE
while QUALITÉ($\mathcal{H}^*$) s'améliore **do**
    $\Theta_c \leftarrow$ GEN CANDIDATS STRUCTURE HTN($\mathcal{H}^*, D$)
    $\mathcal{H}^* \leftarrow$ TROUVER MEILLEUR HTN($\Theta_c \cup \{\mathcal{H}^*\}$)
$\mathcal{H}^* \leftarrow$ EXTRAIRE PARAMÈTRES HTN($\mathcal{H}^*, D$)

Apprentissage de la structure d'un HTN La taille de l'espace des HTN structurellement valides étant très grande, la génération des candidats devrait être biaisée vers des structures

pertinentes, tout en conservant la possibilité d'échapper aux optima locaux.

Pour ce faire, nous présenterons une approche novatrice basée sur l'*extraction de motifs* [ABH14] afin d'abstraire les comportements apparaissant fréquemment ensemble de manière à amorcer la génération de candidats, utilisant ensuite une version adaptée de l'algorithme HTN-Maker [HMK08].

Optimisation des paramètres d'un HTN Pour obtenir une paramétrisation pertinente d'une structure de HTN, une intuition simple est de propager les paramètres dans la hiérarchie, à la fois depuis les primitives et les tâches démontrées.

Cependant, nous nous rendrons rapidement compte que cette propagation naïve ne transmettra en réalité que peu d'informations. Nous proposerons donc une approche basée MAX-SMT pour unifier les paramètres et résoudre ce problème.

Évaluation efficace d'un HTN Si l'approche la plus naturelle pour évaluer la qualité d'un HTN serait d'évaluer ses capacités de planification, cette approche est bien trop couteuse pour être utilisée durant la phase de recherche de la meilleure structure.

Nous présenterons donc une nouvelle métrique basée sur le principe de longueur de description minimale (Minimum Description Length, MDL), assez naturel à appliquer à notre cas étant donné les similitudes entre les HTN et les grammaires formelles. Cette métrique nous permettra d'orienter la recherche vers des modèles capables de reproduire de façon efficace les comportements démontrés, le tout pour un coût de calcul relativement faible.

Appendix B

Handling Recursive Task Definitions: Arbitrary Recursive Structures

Contents

B.1 A New Argument Propagation Procedure . . . 121
B.1.1 Direct Recursions . . . 121
B.1.2 Indirect and Independent Recursions . . . 123
B.2 Parameter Minimization . . . 126
B.2.1 Required Features in the MAX-SMT Solver . . . 131
B.2.2 Defining Datatypes and Functions . . . 131
B.2.3 Defining the Constraints . . . 132
B.2.3.1 Evidence from the Demonstrations . . . 132
B.2.3.2 Grouping Constraints . . . 133
B.2.4 Defining the Optimization Objectives . . . 137
B.3 Conclusion . . . 138

While Chapter 4 presented a solution to handle a specific recursive pattern, we wish for our parameterization algorithm to work efficiently on every valid HTN structure. For that, we need to consider that we can have task parameters that can refer to a next step in a recursive decomposition, similar to this previous chapter. However, instead of having a single parameter that we need to chose whether it should be bound to the current or the next step, we introduce two separate parameters, one for each case. While these parameters may be unified together, they do not have to be.

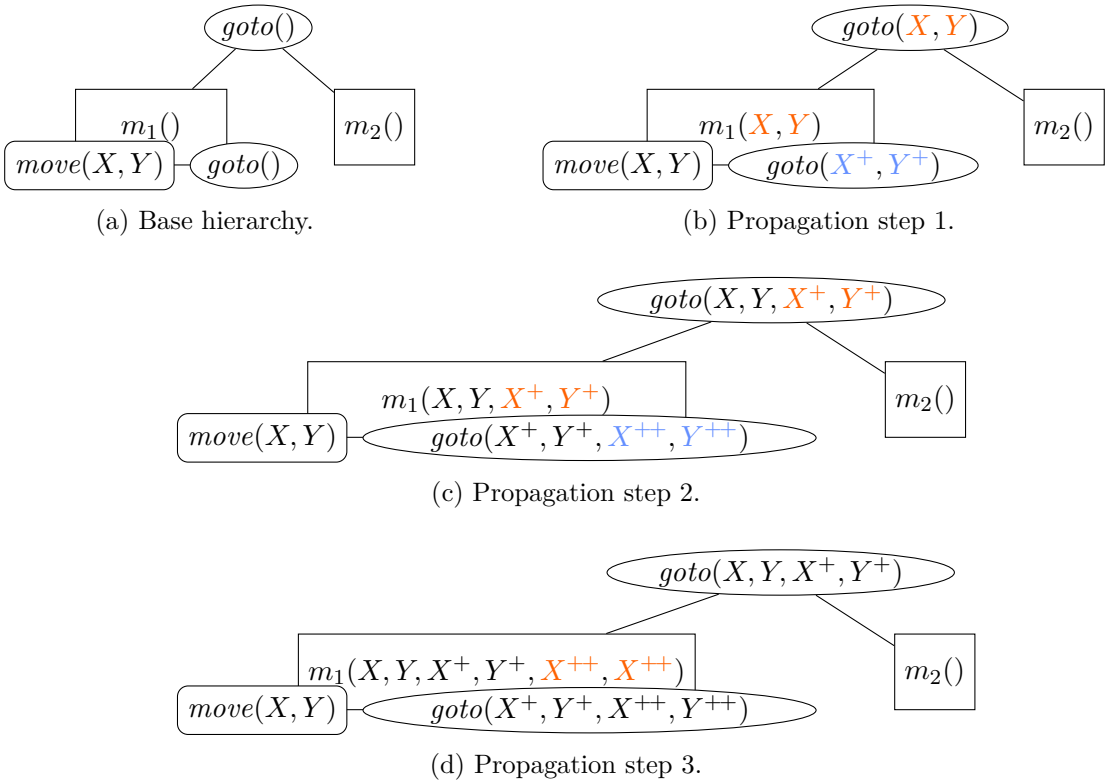
B.1 A New Argument Propagation Procedure

In order to obtain such a parameterization, we need to make a change to the way we propagate arguments upwards, as the current filter in the methods is not appropriate to generate the argument structure we wish to obtain. Instead of keeping track of the methods a parameter X has been propagated through during the propagation phase, we define a source subhierarchy for X and increment an instantiation counter each time a new parameter X', based on X, is added to a subtask in this source subhierarchy. To represent this counter easily in the examples, we will use a $\cdot^{+}$ (counter of 1) and $\cdot^{++}$ (counter of 2). Furthermore, it should be noted that three parameters X, X^{+} and X^{++} do represent three different parameters, their common name mainly highlighting their common origin.

B.1.1 Direct Recursions

Before detailing further this new procedure and the associated algorithm, let us provide an example on a simple `goto` task structure, identical to the one presented in Chapter 4 and

presented in Figure B.1. Algorithm B.1 presents the basic algorithm that was used to propagate the arguments in this example. A more general version will be presented later in this section.



(a) Base hierarchy.

(b) Propagation step 1.

(c) Propagation step 2.

(d) Propagation step 3.

Figure B.1: New argument propagation scheme algorithm example on a simple goto task. In each step, orange arguments are the ones that were propagated upwards and blue ones the ones that were added by updating the subtasks.

Algorithm B.1 Argument Propagation For Recursion

Input: H the set of subhierarchies

▷ *Arguments to propagate upwards.*

1: **repeat**
2: $A \leftarrow \emptyset$ ▷ *The arguments that will be used to update the subtasks, indexed by task symbol*
3: **for all** $h \in H$ **do**
4: $A \leftarrow A \cup$ Propagate Arguments Upwards(h)
5: **for all** $h \in H$ **do**
6: Update Subtasks Basic(h, A)
7: **until** fixed point reached

As can be seen in this example and in the algorithm, at each iteration, we propagate yet unprogapated arguments from the leaf tasks (here, starting with the move task) up to the top level task, similar to what we were doing in the original version of the argument propagation procedure. However, when updating the subtasks of a subhierarchy h, for each parameter $\tilde{P}$ added to a subtask, if its source argument P is an argument of a subtask of h, then we increment its iteration counter by one. This is denoted by writing the argument as P^+ in the example, and incrementing the counter of P, as seen line 3. However, if an argument's instantiation counter is equal to two, then we do not propagate it upwards from a method anymore, which

Algorithm B.2 PROPAGATE ARGUMENTS UPWARDS(h)

Input: h a subhierarchy
Output: A_t the set of newly added arguments to the top level task of h

1: $t_h \leftarrow \text{ttop}(h)$
2: $A_t \leftarrow \emptyset$ ▷ *Arguments to add to* t_h
3: **for all** $m \in \text{METHODS}(h)$ **do**
4: $\quad A_s \leftarrow$ the arguments of all the subtasks of m
5: $\quad A_m \leftarrow A_s \setminus \text{args}(m)$
6: $\quad A_t \leftarrow A_t \cup \{X \in A_m \mid \text{COUNTER}(X) < 2\}$
7: $\quad \text{args}(m) \leftarrow A_s$
8: $\text{args}(t) \leftarrow \text{args}(t) \cup A_t \cup$

Algorithm B.3 UPDATE SUBTASKS BASIC(h, A)

Input: h a subhierarchy
A the set of newly added parameters to the abstract tasks

1: **for all** $t_s \in \text{SUBTASKS}(h)$ **do**
2: $\quad$ **if** $A_s = \text{NEW ARGS}(A, t_s)$ is not empty **then**
3: $\quad\quad A_s^+ \leftarrow \{Y \mid \text{COUNTER}(Y) = \text{COUNTER}(X) + 1, \forall X \in A_s\}$
4: $\quad\quad \text{args}(t_s) \leftarrow \text{args}(t_s) \cup A_s^+$

corresponds to the filter line 6 in the algorithm. In the example, this is shown in Figure B.1d, where X^{++} and Y^{++} are not propagated upwards from m_1. While this filter is arbitrary, as we could stop at a counter greater than two, this limit presents two advantages: i) generating parameters that can refer both to the current instantiation of P (the source argument) and to a next instantiation of P in the top level task and ii) limiting the total number of arguments that are generated in our model. The ability to refer to the next instantiation of a parameter will be used to either link two steps of a recursion or to find bindings to a "last instantiation" of this parameter, as was done in the previous section.

B.1.2 Indirect and Independent Recursions

One may naturally wonder what happens in the case of indirect recursions, or if a recursion has another independent recursive task as subtask. Figure B.2 shows an example hierarchy with an indirect recursion.

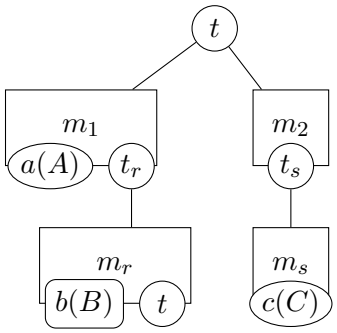
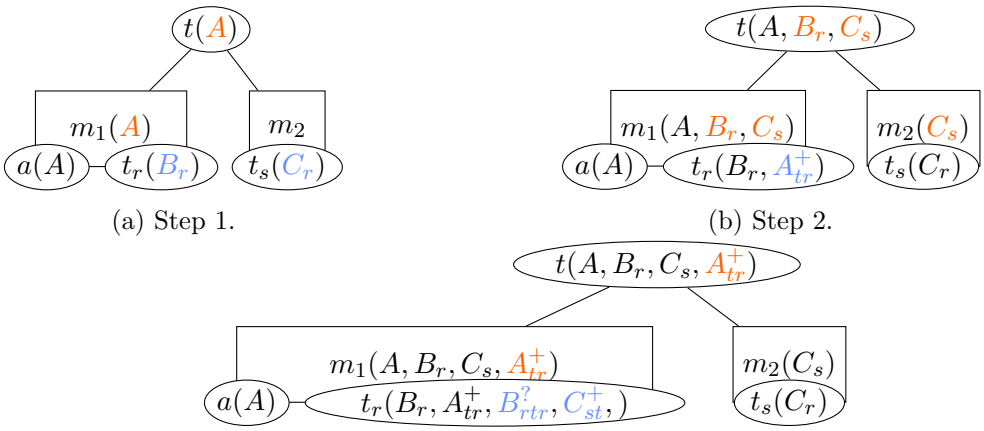


Figure B.2: An HTN with an indirect recursion. Here, t is recursive through t_r and t_s is a non-recursive abstract task.

Figures B.3 and B.4 present a naive application of the algorithm to this hierarchy (the task

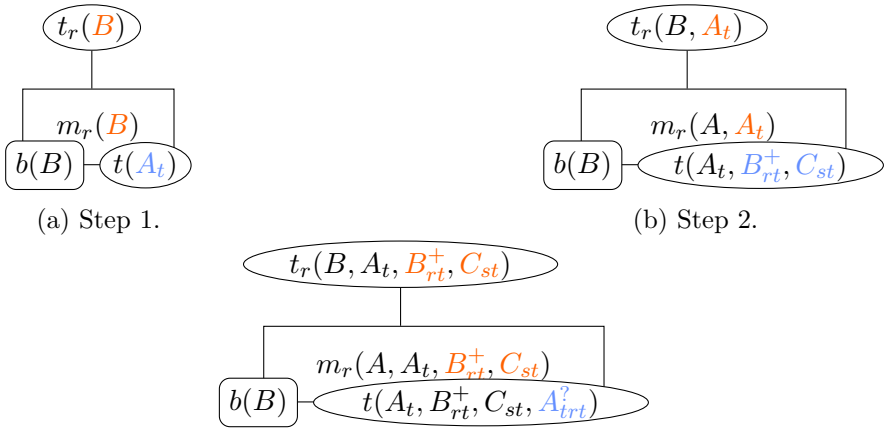
t_s is not detailed as it is trivial). To apply it to this structure, we add as a subscript to a given propagated parameter the subtasks to which it was added in the update step and increase a parameter counter each time we add it again to a task present in the subscript. However, this present an issue at the third iteration of the algorithm, where simply applying this technique will lead to having parameters B^{++}_{rtr} and A^{++}_{trt}, without a valid B^{+} or A^{+} to refer to the next instantiation.



(a) Step 1. (b) Step 2.

(c) Step 3. Here, the problematic parameter is B_{rtr}.

Figure B.3: Naive application of the propagation algorithm, focusing on the subhierarchy of task t.



(a) Step 1. (b) Step 2.

(c) Step 3. Here, the problematic parameter is A_{trt}.

Figure B.4: Naive application of the propagation algorithm, focusing on the subhierarchy of task t_r.

We therefore added the concept of a *source* parameter to our set of subhierarchy. The source parameters in a given recursion are the ones that are from the leaf tasks not participating in the recursion. We therefore need to define which task are part of a given recursion or not. Intuitively, in the subhierarchy presented in Figure B.2, t and t_r are part of a recursion, while t_s should not be. However, the parameters of t should still be able to set the parameters of a potential next instantiation of t_s. To determine which tasks are part of which recursion, we can build a directed graph of the task dependency and then extract its Strongly Connected Components (SCCs) to obtain the condensed graph, as presented in Figure B.5. In these graphs,

an arrow from node n to n' means that n' depends on n, in the sense that the arguments of n' are a function of the arguments of n Furthermore, the primitives actions (a, b, and c) are represented for clarity but do not play a role in the graph structure because they do not have an associated subhierarchy.

(a) Raw task dependency graph. (b) Condensed task dependency graph.

Figure B.5: Task dependency graph for the hierarchy in Figure B.2.

The condensed graph shows that t and t_r are part of a recursion, and that t_s is not. However, the parameters of t and t_r depend on that of t_s. Therefore, we process the tasks in this graph (and their associated subhierarchies) in topological order, ensuring that all the dependencies of a given task t have had their parameters generated before t's own arguments are generated.

We can therefore present a new version of Algorithm B.1, adding the separation into different recursion groups processed in a specific order and the concept of source subhierarchies for the arguments. In this algorithm, presented in Algorithm B.4, the function GENERATE ARGS(t) generate new variables for each argument of t, with a counter of 0. It is initially only defined on primitive tasks, and is updated each time a set of subhierarchies is processed.

Algorithm B.4 Argument Propagation For Recursion - Improved

Input: H_g the graph of subhierarchies recursion groups

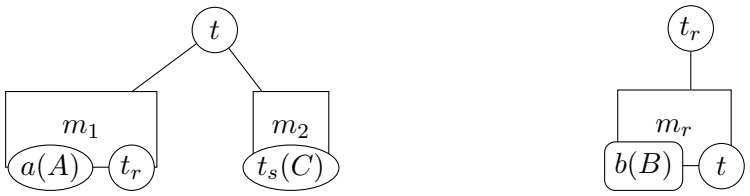
```
1:  for all H ∈ H_g sorted in topological order do
2:      for all h ∈ H do
3:          for all t_s ∈ SUBTASKS(H) do
4:              args(t_s) ← GENERATE ARGS(t_s)        ▷ Parameters from previous iteration
5:              for all x ∈ args(t_s) do
6:                  SOURCE(x) ← h
7:      repeat
8:          A ← ∅                                     ▷ The arguments that will be used to update the subtasks
9:          for all h ∈ H do
10:             A ← A ∪ PROPAGATE ARGUMENTS UPWARDS(h)
11:         for all h ∈ H do
12:             UPDATE SUBTASKS(h, A)
13:     until fixed point reached
14:     for all h ∈ H_g do
15:         Define GENERATE ARGS(ttop(h))
```

Let us now present the application of this improved algorithm on the same structure as presented previously (Figure B.2). According to the graph presented Figure B.5b, we will first process the subhierarchy of t_s, trivially obtaining the parameterization $t_s(C)$. Therefore, we can then process the recursion group $\{t, t_r\}$. After applying the first part of the process (Alg. B.4, line 2), we obtain the subhierarchies presented in Figure B.6.

Algorithm B.5 UPDATE SUBTASKS(h, A)

Input: h a subhierarchy
A the set of newly added parameters to the abstract tasks

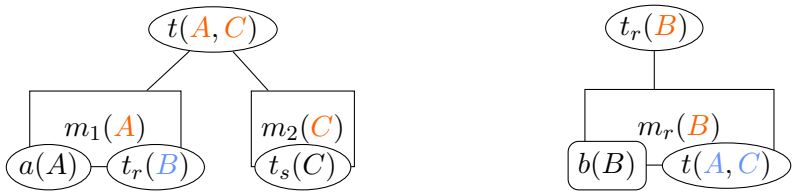
1: **for all** $t_s \in$ SUBTASKS(h) **do**
2: **if** $A_{new} =$ NEW ARGS(A, t_s) is not empty **then**
3: $A_s^+ \leftarrow \{y \mid \text{COUNTER}(y) = \text{COUNTER}(x) + 1, \forall x \in A_{new}, \text{SOURCE}(x) = h\}$
4: $A_s \leftarrow \{y \mid \text{COUNTER}(y) = \text{COUNTER}(x), \forall x \in \{A_{new} \setminus A_s^+\}\}$
5: $\text{args}(t_s) \leftarrow \text{args}(t_s) \cup A_s^+ \cup A_s$



(a) Base subhierarchy for t. (b) Base subhierarchy for t_r.

Figure B.6: Base subhierarchies for the recursion group $\{t, t_r\}$ after processing t_s and generating the subtask arguments.

We then enter the main argument propagation loop (Alg. B.4, line 2), in which we can propagate the arguments as presented in Figures B.7 to B.11. Here, the previous issue is naturally avoided, and we can see that t is able to refer to the direct instantiation of its parameters A, B and C, as well as a future one. When considering a recursion instantiation starting with t, even though B is an argument t_r, the first time we encounter the primitive action b, B is correctly the argument of the first instantiation of the argument that we encounter.



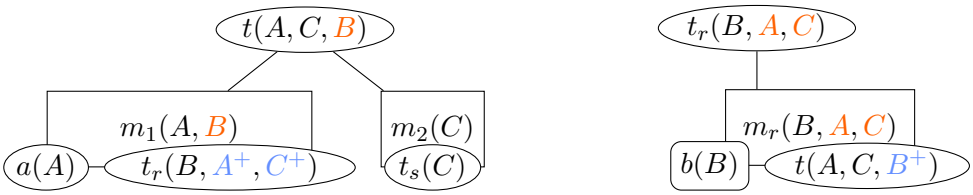
(a) Subhierarchy for t. (b) Subhierarchy for t_r.

Figure B.7: Application of Algorithm B.4 to the HTN structure in Figure B.2, step 1.

Furthermore, we this new propagation methods also naturally handles the presence of independent recursions in the hierarchy. Modifying the previous subhierarchy as presented in Figure B.12, the new process will not change the parameterization of the *goto* task presented earlier (Figure B.1). Therefore, assuming that the parameters $\{X', Y', X'', Y''\}$ have been generated through a call to GENERATE ARGS(*goto*), it is easy to see that the previously presented procedure can be applied to this set of subhierarchies and will give a result close to the one presented in Figure B.11.

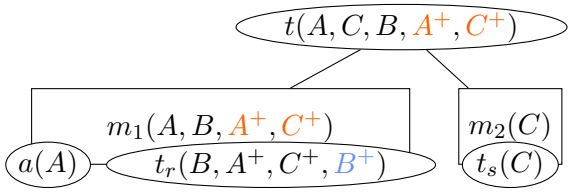
B.2 Parameter Minimization

Let us now focus on the processing applied to the propagated arguments in this general case through a formulation as a MAX-SMT problem. Before formulating the problem, some changes have to be made to the arguments of the hierarchy extracted using the previously presented

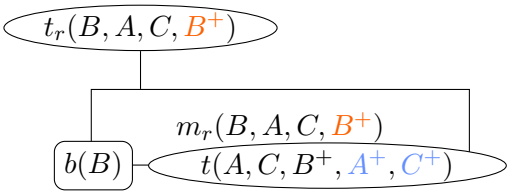


(a) Subhierarchy for t. (b) Subhierarchy for t_r.

Figure B.8: Application of Algorithm B.4 to the HTN structure in Figure B.2, step 2.

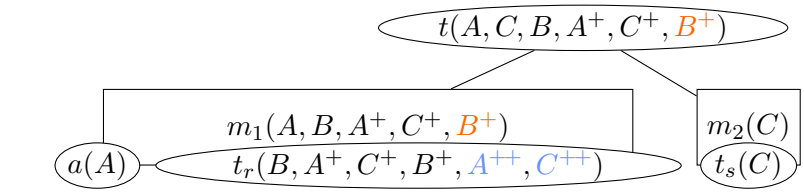


(a) Subhierarchy for t.

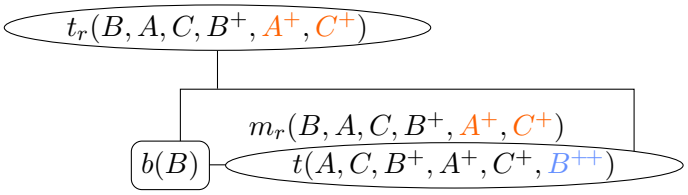


(b) Subhierarchy for t_r.

Figure B.9: Application of Algorithm B.4 to the HTN structure in Figure B.2, step 3.

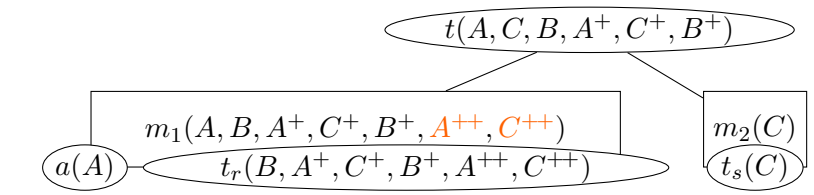


(a) Subhierarchy for t.

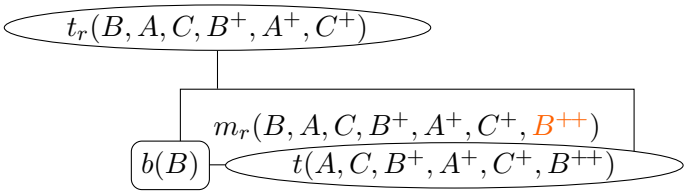


(b) Subhierarchy for t_r.

Figure B.10: Application of Algorithm B.4 to the HTN structure in Figure B.2, step 4.

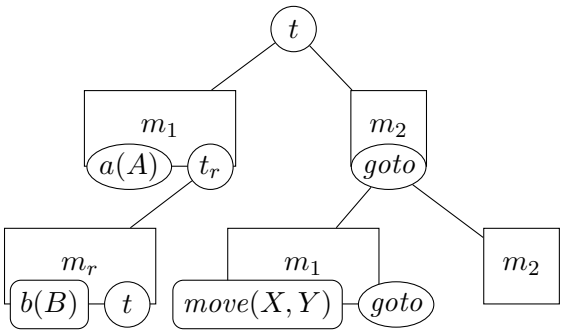


(a) Subhierarchy for t.

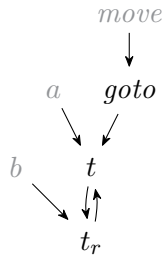


(b) Subhierarchy for t_r.

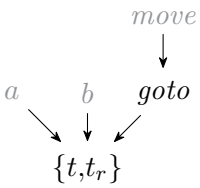
Figure B.11: Application of Algorithm B.4 to the HTN structure in Figure B.2, step 5.



(a) HTN structure.

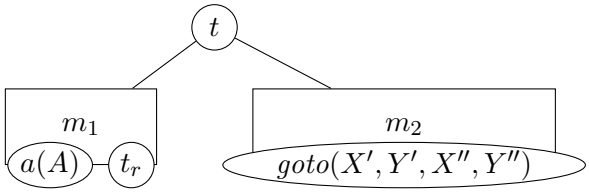


(b) Raw task dependency graph.

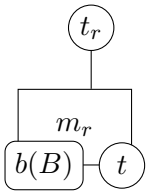


(c) Condensed task dependency graph.

Figure B.12: HTN structure with two independent recursion groups, one of which is indirect, and associated dependency graphs.



(a) Base subhierarchy for t.



(b) Base subhierarchy for t_r.

Figure B.13: Base subhierarchies for the recursion group $\{t, t_r\}$ after processing *goto* and generating the subtask arguments

algorithms, to convert between hierarchy arguments and constants used in the MAX-SMT problem.

Focusing once again on the goto task presented Figure B.1, we mainly generate different constants for parameters in a top level task from parameters in a method or subtask. To make the distinction, considering a parameter P, we will write $\bar{P}$ to denote the constant representing the parameter in the top task, and $\underline{P}$ to denote the constant representing the parameter in a method or subtask, as presented in Figure B.14. We will also introduce the notation P^0, to denote an argument that is specifically not a next instantiation, i.e. not P^+ nor P^{++}.

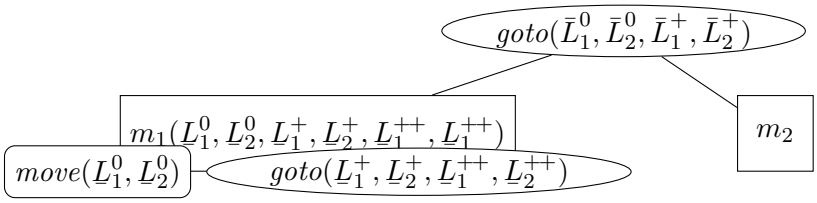


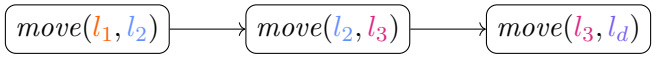
Figure B.14: The goto task, with its propagated parameters modified to be converted to MAX-SMT constants.

Using these new notations, we can now write an example of decomposition for a given demonstration, as shown in the example Figure B.15. In this example, the notation $\cdot \mid x_N$ is used to indicate the node identifier, unique across all decomposition, which will be used to define the constraints later in this section. Furthermore, some abstract nodes (such as 3_N, 6_N and 9_N) have two lines, because they can be seen both as a subtask or a top level task. In that case, the first one shows what happens when viewing this node as a subtask (for example, considering 3_N as a subtask in the subhierarchy with the top level in node 0_N), and the second one when viewing it as a top level task.

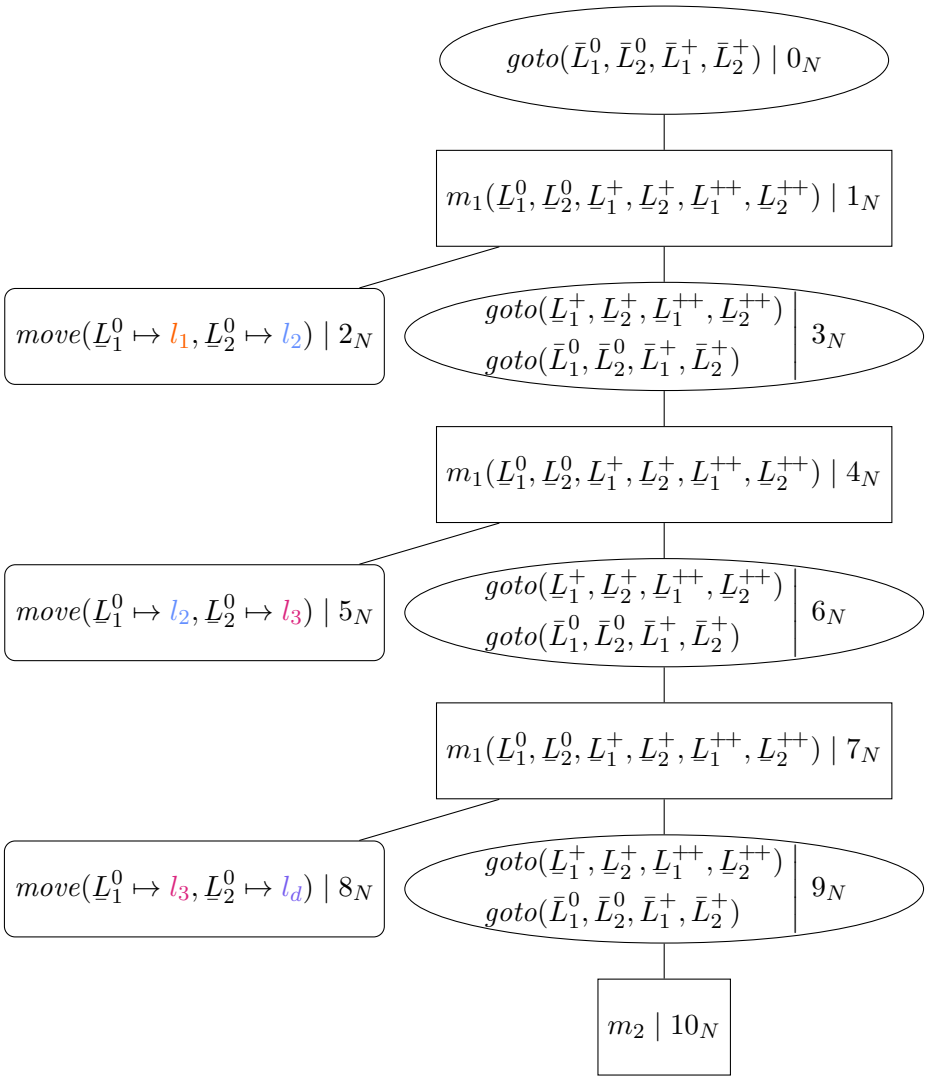
Here, the example is simple enough to have a single *recursion chain*. However, as more complex examples will potentially have several ones, such as in the case of a task which decomposition relies on multiple independent instantiations of the goto task. In that case, we will write $\mathcal{R}$ the set of all the possible chains, and assign them similar identifiers $x_R, x \in \mathbb{N}$ as done previously. In order to determine when we are in a given recursion chain and when we exit it, we reuse the tasks dependency graphs presented earlier (e.g. Figures B.13b), and consider that we change out of a recursion each time the decomposition requires us to move out of the current condensed node.

We will now propose a MAX-SMT formulation for simplifying the set of arguments in a given HTN, replacing the procedure presented in Section 4.3.2. At a high level, this procedure is similar, trying to unify arguments when supported by some evidence found in the demonstrations. However, we will consider that we can have recursion chains, which will be used to determine potential evidence pertaining to the last instantiation of a given argument, as well as the *next step* constraints between some arguments, drawing on the insights provided by the work presented in Section 4.4.

It should be noted that even though we use the MAX-SMT terminology, our formulation will depart from the traditional setting, in the sense that we do not limit our optimization objective to the maximal satisfaction of some soft constraints. We rather try and optimize arbitrary cardinality constraints.



(a) Demonstration trace.



(b) Decomposition tree.

Figure B.15: Possible example trace and corresponding decomposition tree example for the goto task. Colours are used to highlight identical constants.

B.2.1 Required Features in the MAX-SMT Solver

Before presenting the details of the constraints that we use, let us define the features that the used solver should support.

Uninterpreted functions and variables with equality This is the base on which we can build our unification constraints.

Optimization of arbitrary cardinality constraints This is used to propose natural optimization objectives.

Algebraic datatypes Because we will want to differentiate between arguments, ground constants and several kinds of unique identifiers, this feature is required to remove large inequality constraints.

B.2.2 Defining Datatypes and Functions

In order to express the constraints that will form the satisfaction problem that we will solve, we must define some datatypes and uninterpreted functions.

All the datatypes that we use are simple enumerations. All the types will be illustrated using the structure and decomposition examples presented in Figures B.14 and B.15.

The first datatype that we define is the *Parameter* type, whose domain is written $\mathcal{P}$, which comprises all our parameters. Furthermore, it is augmented with a set of parameters that are constant across a recursion chain, and noted $\cdot^C$, such as the destination location in a `goto` task. For simplicity, we will write $\mathcal{P}^* = \mathcal{P} \setminus \left\{P \in \mathcal{P} | \neg P^C\right\}$. One constant parameter is generated for each $\cdot^{++}$ parameter in the original propagated arguments. In our example, $\mathcal{P} = \left\{\bar{L}_1^0, \bar{L}_2^0, \bar{L}_1^+, \bar{L}_2^+, \underline{L}_1^0, \underline{L}_2^0, \underline{L}_1^+, \underline{L}_2^+, \underline{L}_1^{++}, \underline{L}_2^{++}, L_1^C, L_1^C\right\}$.

The second one is the *Ground Constant* type, noted $\mathcal{C}$, which contains all the constants. The constants are unique to a given example, that is two constants with the same name c in two distinct demonstrations d and d' will be mapped to two different objects c_d and $c_{d'}$. It is augmented by the symbol ξ, used to denote the *free* variable, which will be used as a placeholder in case we do not have evidence of a given binding. In our example, $\mathcal{C} = \{l_1, l_2, l_3, l_d, \xi\}$. For simplicity, we will use $\mathcal{C}^d, d \in \mathcal{D}$ to denote the set of constants restricted to a given demonstration d.

We also define the *Node Identifier* type, noted $\mathcal{N}$, which contains all the node identifiers in all the decompositions that are part of our demonstration set. In our example, $\mathcal{N} = \{x_N | x \in [0, 10]\}$.

We also define a *Parameter Group* type, noted $\mathcal{G}$, which will be mainly used to constrain the possible unifications of parameters and express cardinality constraints to be optimized. These groups will be used to generate the set of simplified parameters. Additionally, we consider a special group δ containing all the arguments that should be removed from the HTN.

Let us now define some uninterpreted functions that will be required, specifying their domains and codomains, as well as the reasoning behind their existence, to make the interpretation of the constraints easier.

Equations (B.1a) to (B.1c) shows the definitions of these functions. Here, PGroup maps a given parameter to each group. GndParam is used to assert that a given parameter is mapped to a specific constant in a specific node of the decomposition, allowing us to introduce the evidence given by the demonstrations in our set of constraints. The GndGroup is similar but applied to parameter *groups* instead. It will be used to express the fact that sometimes it may

be possible to define a grounding for a given parameter group even though some of the group's parameters are free according the initial evidence.

$$\textsc{PGroup} : \mathcal{P} \to \mathcal{G} \tag{B.1a}$$
$$\textsc{GndParam} : \mathcal{P} \times \mathcal{N} \to \mathcal{C} \tag{B.1b}$$
$$\textsc{GndGroup} : \mathcal{G} \times \mathcal{N} \to \mathcal{C} \tag{B.1c}$$

B.2.3 Defining the Constraints

We will now define the constraints to solve to obtain a valid parameterization, grouped according to the idea behind them for clarity. Because we are using optimization objectives, which will be defined later, all the constraints presented are hard constraints. For illustrating these constraints, unless explicitly specified, we will continue to refer to the example presented Figure B.15. Because of limitations in the SMT engine used, some preprocessing is done before generating the actual constraints, particularly in order to eliminate quantifiers. Therefore, whenever an equation is not numbered, it means that this part is actually handled as part of the preprocessing. For example, the equation below should be read as an instantiation of all the possible couples (P, N) for which we add the constraint in Equation B.2.

$$\begin{aligned} &\forall (P, N) \in \mathcal{P} \times \mathcal{N} \\ &\quad \textsc{GndParam}(P, N) = \xi \end{aligned} \tag{B.2}$$

B.2.3.1 Evidence from the Demonstrations

Let us describe the most basic set of constraints, the ones that are a direct translation of the basic evidence in our example, presented in Equations (B.3a) to (B.3c). In order to conserve a clear definition of the constraints, let use write $\textsc{Type}(X)$ the function that returns the type (as defined in the planning domain) of an argument or ground constant. Let us also write $\textsc{IsFixed}(P)$ the functions that returns whether a parameter is fixed in the given demonstration of a task, and cannot be removed.

$$\begin{aligned} &\forall (P, N) \in \mathcal{P}^* \times \mathcal{N} \\ &\quad \text{if } \exists C \in \mathcal{C}, P \underset{N}{\mapsto} C \\ &\qquad \textsc{GndParam}(P, N) = C \qquad \text{(B.3a)} \\ &\quad \text{else} \\ &\qquad \text{if } \textsc{IsFixed}(P) \\ &\qquad\quad \mathcal{C}^P = \left\{ C \in \mathcal{C}^d \mid d \in \mathcal{D} \wedge \textsc{Type}(C) = \textsc{Type}(P) \right\} \cup \{\xi\} \\ &\qquad\qquad \bigvee_{C \in \mathcal{C}^P} \textsc{GndParam}(P, N) = C \qquad \text{(B.3b)} \\ &\qquad \text{else} \\ &\qquad\quad \textsc{GndParam}(P, N) = \xi \qquad \text{(B.3c)} \end{aligned}$$

Applying these constraints to our example, Table B.1 presents a subset of the constraints that can be obtained, highlighting which part of the previous equation is their source.

Constraint	Source
$\textsc{GndParam}(\underline{L}_1^0, 2_N) = l_1$	(B.3a)
$\textsc{GndParam}(\underline{L}_1^0, 5_N) = l_2$	(B.3a)
$\textsc{GndParam}(\underline{L}_1^0, 1_N) = \xi$	(B.3c)
$\textsc{GndParam}(\bar{L}_1^+, 3_N) = \xi$	(B.3c)

Table B.1: Example of direct evidence constraints that can be generated

Using the demonstrations, we can add another set of constraints, setting the possible values that a constant value parameter P^C can take, in order to avoid irrelevant parameterization. To this end, let us define $\textsc{Source}(P^C)$ the function that returns the source argument P of P^C. Let us also write $\mathcal{P}^C = \mathcal{P} \setminus \left\{P \in \mathcal{P} | P^C\right\}$. Finally, let us write N_R^* as the node identifier of the start of the recursion chain.

$$
\begin{aligned}
&\forall (P^C, R) \in \mathcal{P}^C \times \mathcal{R} \\
&\quad \text{if } P^C \text{ is part of } R \\
&\qquad \mathcal{C}^P = \left\{ C \in \mathcal{C}^R \mid \exists N \in \mathcal{N}^R, \textsc{Source}(P^C) \underset{N}{\mapsto} C \right\} \\
&\qquad\quad \text{if } \mathcal{C}^P = \emptyset \\
&\qquad\qquad \textsc{GndParam}(P^C, N_R^*) = \xi \qquad \text{(B.4a)} \\
&\qquad\quad \text{else} \\
&\qquad\qquad \begin{cases} \bigvee_{C \in \mathcal{C}^P} \textsc{GndParam}(P^C, N_R^*) = C & \text{if } \textsc{PGroup}(P^C) \neq \delta \\ \textsc{GndParam}(P^C, N_R^*) = \xi & \text{otherwise} \end{cases} \qquad \text{(B.4b)} \\
&\quad \text{else} \\
&\qquad \textsc{GndParam}(P^C, N_R^*) = \xi \qquad \text{(B.4c)} \\
&\bigwedge_{N \in \mathcal{N}^R} \textsc{GndParam}(P^C, N) = \textsc{GndParam}(P^C, N_R^*) \qquad \text{(B.4d)}
\end{aligned}
$$

Here, Equations (B.4a) to (B.4c) are used to limit the possible groundings of a given constant argument P^C. Equation B.4b enforces that if P^C is not dropped, then it must be grounded with one of the possible constants in the recursion chain R, otherwise it must be free. Equation B.4a allows to handle the case where we have no instantiation of the source argument in the currently considered recursion chain R. Similarly, Equation B.4c handles the case where P^C is not part of R. Finally, Equation B.4d is used to enforce that in a given recursion chain R, the constant associated with P^C is unique.

Table B.2 presents a subset of the constraints that can be obtained with these new constraints, as shown previously.

B.2.3.2 Grouping Constraints

We will now describe the constraints that are used to govern the grouping of the parameters. This will cover both structural constraints, as not any parameter can be unified with another one, and evidence-based constraints, where the previously presented grounding constraints will be used to build more complex ones.

Constraint	Source
$\begin{cases} \textsc{GndParam}(\underline{L}_1^C, 0_N) = l_1 \\ \vee\ \textsc{GndParam}(\underline{L}_1^C, 0_N) = l_2 \quad \text{if } \textsc{PGroup}(P^C) \neq \delta \\ \vee\ \textsc{GndParam}(\underline{L}_1^C, 0_N) = l_3 \\ \\ \textsc{GndParam}(\underline{L}_1^C, 0_N) = \xi \qquad \text{otherwise} \end{cases}$	(B.4b)
$\bigwedge_{x_N, \forall x \in [1,10]} \textsc{GndParam}(P^C, N) = \textsc{GndParam}(P^C, 0_N)$	(B.4d)

Table B.2: Example of direct evidence constraints that can be generated

The first, most basic group of constraints is the one that governs which groups a given argument may be associated with. Because the grouping of arguments is local to a given subhierarchy $h \in H$, we will consider a specific h in the following paragraphs.

To identify the possible groups that a parameter may be part of, for every subhierarchy $h \in H$, we generate a set of unique identifiers $\mathcal{G}^h \subset \mathcal{G}$, one for each parameter in $\mathcal{P}^h$, so that we can define a bijective function $\textsc{PGroup}_{base} : \mathcal{P}^h \to \mathcal{G}^h$. The group G associated with a parameter P will be called the *base group* of P, and will be written G^P as a shorthand.

$$\forall G \in \mathcal{G}^h$$
$$\textsc{GContainsTopArg}(G) = \bigvee_{P \in \bar{\mathcal{P}}^h} \textsc{PGroup}(P) = G \tag{B.5}$$

Constraint B.5 is a constraint placed on the uninterpreted function GContainsTopArg, which will be used in later constraints.

We can now define the possible groups for the top arguments of h. In order to break symmetries, we need to order the parameters, for which we use their appearance order in the top level task, from left to right, indexed from 0 to n.

$$\forall P_i \in \bar{\mathcal{P}}^h$$
$$\textsc{PGroup}(P_i) = G^{P_i} \vee \bigvee_{j=0}^{i-1} \left(\textsc{PGroup}(P_j) = G^{P_j} \wedge \textsc{PGroup}(P_i) = G^{P_j}\right) \tag{B.6}$$

The constraint defined in Equation B.6 shows that a given parameter is either associated with its base group, or with one of the preceding groups, provided this preceding group is not empty and a unification is possible.

Similarly, we define the possible groups for the bottom arguments of h. We keep a similar ordering of the parameters as previously for the same reason. To clarify the following constraints, let us write $\bar{\mathcal{G}}^h$ the set of the base groups for all the top parameters of h. Let us also write $\textsc{ArgsFixed}(h), h \in H$ the set of arguments that are fixed, and cannot be removed in a

subhierarchy, that is the argument of the primitive tasks or of the demonstrated tasks.

$$
\begin{aligned}
&\forall P_i \in \mathcal{P}^h \\
&\quad \textsc{PGroup}(P_i) = G^{P_i} \\
&\qquad \vee \bigvee_{G^{\bar{P}} \in \bar{\mathcal{G}}^h} \left(\textsc{PGroup}(\bar{P}) = G^{\bar{P}} \wedge \textsc{PGroup}(P_i) = G^{\bar{P}} \right) \\
&\qquad \vee \begin{cases} \textsc{PGroup}(P_i) = \delta & \text{if } P_i \notin \textsc{ArgsFixed} \\ \bot & \text{otherwise} \end{cases} \\
&\qquad \vee \bigvee_{j=0}^{i-1} \begin{cases} \begin{pmatrix} \textsc{PGroup}(P_j) = G^{P_j} \\ \quad \wedge \textsc{PGroup}(P_i) = G^{P_j} \end{pmatrix} & \text{if } \textsc{Method}(P_j) = \textsc{Method}(P_i) \\ \begin{pmatrix} \textsc{PGroup}(P_j) = G^{P_j} \\ \quad \wedge \textsc{PGroup}(P_i) = G^{P_j} \\ \quad \wedge \textsc{GContainsTopArg}(G^{P_j}) \end{pmatrix} & \text{otherwise} \end{cases}
\end{aligned}
\tag{B.7}
$$

Here, Equation B.7 shows that we have a parameter either associated with its base group or one of the preceding ones (as in Equation B.6), but also maybe any of the (non-empty)top argument group, to handle the case where a bottom argument is unified with one (or several) top arguments. In case a bottom argument is not coming from a fixed argument task, we add the δ group as a possibility, to be able to remove it. Lastly, while we can unify it with any other parameter in the same method, this constraint enforces that parameters can be unified across method if and only if there is a top argument acting as a bridge.

Focusing now on the constant-valued arguments $\mathcal{P}^{h,C}$, and writing P_b^C the source of P^C we have the following constraint:

$$
\begin{aligned}
&\forall P^C \in \mathcal{P}^{h,C} \\
&\quad \mathcal{G}^{P^C} = \begin{cases} \left\{ \begin{array}{l} \textsc{PGroup}_{base}(P) \mid P \in P^*, \\ \textsc{ArgType}(P) \text{ is a subtype of } \textsc{ArgType}(P_b^C) \end{array} \right\} & \text{if } P_b^C \in \textsc{ArgsFixed} \\ \left\{ \begin{array}{l} \textsc{PGroup}_{base}(P) \mid P \in P^*, \\ \left\{ \begin{array}{l} \textsc{ArgType}(P) \\ \textsc{ArgType}(P_b^C) \end{array} \right\} \text{ have a common ancestor} \end{array} \right\} & \text{otherwise} \end{cases} \\
&\quad \bigvee_{G \in \mathcal{G}^{P^C}} \textsc{PGroup}(P^C) = G
\end{aligned}
\tag{B.8}
$$

Now that we described the possible groups associated with a given parameter, let us describe the structural constraints that will govern the unification of several parameters in a single group. Because groups are local to a given subhierarchy, we will still keep focusing on h. These constraints only apply to subhierarchies where the top tasks are not fixed, as they are mainly a translation of the origin of the argument after the propagation process.

$$\forall P^0 \in \mathcal{P}^{0,h}$$
$$\quad \textsc{PGroup}(\bar{P}^0) \neq \delta \Rightarrow \textsc{PGroup}(\bar{P}^0) = \textsc{PGroup}(\underline{P}^0) \tag{B.9a}$$
$$\forall P^C \in \mathcal{P}^C, \text{and corresponding } \mathcal{P}_b^h = \left\{(\bar{P}_0^+, \underline{P}_0^+, \underline{P}_0^{++}), \ldots, (\bar{P}_n^+, \underline{P}_n^+, \underline{P}_n^{++})\right\}$$
$$\quad \forall (\bar{P}^+, \underline{P}^+, \underline{P}^{++}) \in \mathcal{P}_b^h$$
$$\quad\quad \begin{pmatrix} \textsc{PGroup}(\bar{P}^+) \neq \textsc{PGroup}(P^C) \Rightarrow \textsc{PGroup}(P^C) = \delta \\ \wedge\, \textsc{PGroup}(\bar{P}^+) = \textsc{PGroup}(P^C) \Leftrightarrow \textsc{PGroup}(\underline{P}^{++}) = \textsc{PGroup}(P^C) \\ \wedge\, \textsc{PGroup}(\bar{P}^+) \neq \textsc{PGroup}(P^C) \Leftrightarrow \textsc{PGroup}(\bar{P}^+) = \textsc{PGroup}(\underline{P}^+) \end{pmatrix} \tag{B.9b}$$
$$\quad \forall P^+ \in \mathcal{P}^{+,h} \setminus \mathcal{P}_b^{+,h}$$
$$\quad\quad \textsc{PGroup}(\bar{P}^+) = \textsc{PGroup}(\underline{P}^+) \tag{B.9c}$$

The first constraint (Equation B.9a) simply is used to enforce that both the top and bottom versions of a non-next argument are always in the same group if the top version is not removed. The second one (Equation B.9b) applies only to P^C where there is at least one corresponding tuple $(\bar{P}^+, \underline{P}^+, \underline{P}^{++})$. It enforces that constant-valued parameters and next versions of arguments are unified in a way consistent with their intended meaning, that is that a P^+ refers to P^0 in the next step of a recursion chain, and that similarly P^{++} refers to P^+ and P^C is the same throughout the chain. Finally, Equation B.9c has a similar role, in the simple case where the current subhierarchy is not the source of the P^+ parameter, and therefore does not have to enforce the consistency with the corresponding P^C.

Up to this point, we were only focusing on constraints limited to a single subhierarchy. However, we obviously need to keep a consistent parameterization at the boundaries between them, such as when an abstract task is used as a subtask of another one. Indeed, we do not want to have, for example, a parameter removed in the reference definition of a task, and still use it in the parameterization when it is instantiated as a subtask. To define these consistency constraints for a given task t, let us define t_{ref} the version of t as a top level task, and $\mathcal{T}_{sub}^t$ the set of all the instantiations of t as a subtask. Let us also write $\mathcal{P}_z = \text{zip}(\text{args}(t_{ref}), \text{args}(t_{sub}))$.

$$\forall t \in \mathcal{T}, \forall t_{sub} \in \mathcal{T}_{sub}^t$$
$$\quad \forall (P_{ref}, P_{sub}) \in \mathcal{P}_z$$
$$\quad\quad \textsc{PGroup}(P_{ref}) = \delta \Leftrightarrow \textsc{PGroup}(P_{sub}) = \delta \tag{B.10a}$$
$$\quad \forall \left((P_{ref}^0, P_{sub}^0), (P_{ref}^1, P_{sub}^1)\right) \in \binom{\mathcal{P}_z}{2}$$
$$\quad\quad \left(\textsc{PGroup}(P_{ref}^0) = \textsc{PGroup}(P_{ref}^1)\right) \Rightarrow \left(\textsc{PGroup}(P_{sub}^0) = \textsc{PGroup}(P_{sub}^1)\right) \tag{B.10b}$$

Let us now move to the use of the grounding for constraining the unification of parameters into groups. This grounding information will rely on the information from the demonstrations applied to each parameter previously.

First, we need to add constraints to avoid unifying parameters together without having a positive example, as was done in the initial formulation. Let us then define a logic formula that is true if and only if a given parameter can be used to provide a positive example for unification in a group. Let us define:

- $\mathcal{P}_A = \mathcal{P}^*$, restricted to the parameters in the abstract tasks.
- For a given recursion chain block, with node identifier N, the node identifier of the parent task is written $N^\uparrow$ while the similar set of all child identifiers for the subtasks is written $\mathcal{N}_N^\downarrow$.
- For a given top level parameter $\bar{P} \in \mathcal{P}_A$ in a node with identifier N, the corresponding bottom parameter in the subtask with node identifier $N^\uparrow$ is written $\underline{P}_N^\uparrow$.
- Similarly, for a given bottom level parameter $\underline{P} \in \mathcal{P}_A$ in a node with identifier N,the corresponding top parameter is written $\bar{P}_N^\downarrow$.

$$\begin{aligned} &E_{dir} : \mathcal{P}_A \times \mathcal{N} \to \mathcal{G} \\ &\qquad (P, N) \mapsto \textsc{GndParam}(P, N) \end{aligned} \tag{B.11a}$$

$$\begin{aligned} &E_{ind} : \mathcal{P}_A \times \mathcal{N} \to \mathcal{G} \\ &\qquad (P, N) \mapsto \begin{cases} \textsc{GndGroup}\left(\textsc{PGroup}(\underline{P}_{N^\uparrow}), N^\uparrow\right) & \text{if } \exists (N^\uparrow, \underline{P}_{N^\uparrow}) \\ \textsc{GndGroup}\left(\textsc{PGroup}(\bar{P}_{N^\downarrow}), N^\downarrow\right) & \text{if } \exists (N^\downarrow, \bar{P}_{N^\downarrow}) \end{cases} \end{aligned} \tag{B.11b}$$

Here, the function E returns the grounding evidence associated with a parameter P in a node with identifier N. It should be noted that even though we are talking about grounding evidence, any member of this tuple may be the free (ξ) grounding.

More precisely, E_{dir} (Equation B.11a) returns the direct evidence from the demonstrations, and is therefore always defined for every couple (P, N). E_{ind} (Equation B.11b) returns indirect evidence, based on the evidence from parameter groupings in the parent or children decomposition from a given one with identifier N. The first case should be interpreted as "if P is a top argument and there is a parent task to the decomposition N, then this parent decomposition provides as a potential positive example the grounding of the group associated with the corresponding bottom argument to P in the parent decomposition". The second case is similar, but with P being a bottom argument and relying on the example provided by the child decomposition.

Using this function, we can define constraints that enforce consistent grounding between parameters and their groups:

$$\begin{aligned} &\forall (P, N) \in \mathcal{P}_A \times \mathcal{N} \\ &\quad G_*(P, N) \neq \xi \Rightarrow \textsc{GndGroup}(\textsc{PGroup}(P), N) = G_*(P, N) \end{aligned} \tag{B.12}$$

B.2.4 Defining the Optimization Objectives

Similar to the original formulation of the constraint satisfaction problem, we wish to obtain a "good" parameterization, and therefore need to define some metric to optimize and go beyond simply satisfying the previously presented constraints.

We therefore define four objectives, to be optimized in lexicographic order.

Minimizing Unused Top Level Arguments For each subhierarchy, we wish to minimize the number of top level arguments that are not unified with at least one bottom parameter. The idea is that such arguments will only cause the planner to consider a larger parameterization space when instantiating a method containing such task, while this argument will never provide

additional information. This objective can be expressed as a pseudo-Boolean function to be minimized, as presented in Equation B.13b.

$$\forall h \in H, \forall \bar{P} \in \bar{\mathcal{P}}^h$$

$$B_h^{\bar{P}} = \left(\bigwedge_{\underline{P} \in \underline{\mathcal{P}}^h} \text{PGROUP}(\bar{P}) \neq \text{PGROUP}(\underline{P}) \right) \vee \text{PGROUP}(\bar{P}) \neq \delta \tag{B.13a}$$

$$O_1 = \min \sum_{\forall B_h^{\bar{P}}} B_h^{\bar{P}} \tag{B.13b}$$

Minimizing the Number of Groups Containing Bottom Parameters While this objective may appear counter-intuitive at first, because the bottom parameters *must* be part of a group, this will push the resulting parameterization towards a state where multiple bottom parameters are part of a single group, thus improving their unification.

$$\forall G \in \mathcal{G}, B_G = \bigvee_{\forall \underline{P} \in \underline{\mathcal{P}}} \text{PGROUP } \underline{P} = G \tag{B.14a}$$

$$O_2 = \min \sum_{\forall B_G} B_G \tag{B.14b}$$

Maximizing Horizontal Unification Across Methods This objective is rather straightforward, unifying as many arguments across methods if possible (remember that we have a constraint to limit these possible unifications).

$$\begin{aligned} &\forall h \in H, \forall G \in \mathcal{G} \\ &\quad B_m = \exists (m_1, m_2) \in \text{METHODS}(h)^2, m_1 \neq m_2 \\ &\qquad \exists (P_1, P_2) \in (\text{args}(m_1), \text{args}(m_2)), \\ &\qquad\quad \text{PGROUP}(P_1) = G \wedge \text{PGROUP}(P_2) = G \end{aligned} \tag{B.15a}$$

$$O_3 = \min \sum_{B_m} B_m \tag{B.15b}$$

Maximizing Horizontal Unification Across Subtasks This objective is basically the same as previous one, but operating at the method level instead of the subhierarchy level.

$$\begin{aligned} &\forall h \in H, \forall m \in \text{METHODS}(h), \forall G \in \mathcal{G} \\ &\quad B_{st} = \exists (st_1, st_2) \in \text{SUBTASKS}(m)^2, st_1 \neq st_2 \\ &\qquad \exists (P_1, P_2) \in (\text{args}(st_1), \text{args}(st_2)), \\ &\qquad\quad \text{PGROUP}(P_1) = G \wedge \text{PGROUP}(P_2) = G \end{aligned} \tag{B.16a}$$

$$O_4 = \min \sum_{B_{st}} B_{st} \tag{B.16b}$$

B.3 Conclusion

In this appendix, we have presented some ideas on how our parameter learning algorithm could be adapted to better handle arbitrary recursive structures. While these ideas stem from our

work on this topic, we were not able to bring it to a complete state at the time of writing the manuscript.

The proposed constraint system was implemented as a prototype, and was able to give the expected results on the LOGISTICS domain, but only if given a small set of demonstration. In every other case (more demonstrations or other domains), the approach does not scale, and the solver used (Z3 [dMB08]) runs out of memory before being able to output a solution.

Therefore, this remains an open subject, and we hope that the ideas presented in this section may be of use for future research in this avenue.

Appendix C

The Minimum Description Length Principle

As we have hinted at in the introductory chapter of this manuscript, learning grammars from positive examples often leverages a simplicity bias. A common one is the MDL principle, which we will detail here as it is used prominently in our work, summarizing the definitions given by Grünwald [Grü07].

The MDL principle, stemming from information theory, is a model selection tool, dating back to the work of Rissanen [Ris78]. This principle states that learning can be viewed as a form of data compression, as both intend to find some regularity in some source material. It can be summarized as follows: *The best model of some set of data is the one that minimizes the sum of:*

- *The length of the model.*
- *The length of the data reconstructed using the model.*

This technique has already been used in several works, ranging from learning Context Free Grammars (CFGs) [SBS12], of which HTNs are close [EHN94], to finding common graph patterns [BCF20].

To better understand it, let us use the common example of encoding different strings. Consider the sequences below:

`abbabb...abbabb` (C.1)

`mskldj...vapwgf` (C.2)

Sequence C.1 is a repetition of the pattern `abb` 5 000 times, while sequence C.2 is a string of 15 000 random characters. Intuitively, the first sequence appears 'simpler' than the second, in that their seems to be a regularity to it which can easily be described. On the other hand, the second sequence appears a lot more complex, without a clear intuition of how it could be described short of reproducing the whole sequence.

Going back to compression, we need to agree on description method, as this will obviously impact the size of our description. An intuitive choice would be a general purpose computer language. Choosing C as a language, we can write the programs that produce our two sequences, as presented in Figure C.1.

Clearly, the first program, in terms of number of characters, is a lot smaller than the 15 000 characters of the first sequence, which is therefore efficiently compressed using this description. However, the second program is roughly the same size as the original sequence.

More formally, when considering a computer language as the description system for a set of data, we lean towards the *Kolmogorov complexity* [LV90] of the data. Without going into many details about this complexity, it is the size of the shortest program in a given language that is able to produce the data as output. It should be noted that given sufficient data length,

```c
int main() {
    for (int i = 0; i < 5000; ++i){
        printf("abb");
    }
}
```

(a) Sequence C.1.

```c
int main() {
    printf("mskldj...vapwgf");
}
```

(b) Sequence C.2.

Figure C.1: C programs to generate the sequences presented earlier.

the *invariance theorem* states that the language used to compute the Kolmogorov complexity is irrelevant. Indeed, given two languages L_1 and L_2 and a dataset D, assuming that the Kolmogorov complexity of D described using language L_1 is noted K_1 (respectively L_2 and K_2), we have the relation in Equation C.3, where the constant c is independent on the length of the data.

$$|K_1 - K_2| \leq c \quad c \in \mathbb{R}^+ \tag{C.3}$$

As stated in the paper written by Li and Vitányi [LV90], the Kolmogorov complexity can be used to find the MDL of some data, by considering that the best model of this data is the shortest program that can be used to describe it.

However, the Kolmogorov complexity is actually uncomputable if we consider every possible dataset. Furthermore, in practical cases, the data from which we intend to extract a model is too small for the invariance theorem to apply, and thus the programming language considered would have an important impact on the determination of the best model for the data.

We will therefore focus on practical MDL versions, which fall into two main families: *crude* and *refined* MDL. The crude MDL approach use two-part codes, where the total description length of some data is the sum of the model M length and the data encoded using the model. However, it is necessary to choose an arbitrary representation for the model, which may bias the resulting description lengths. The refined part is using one-part codes, which do not exhibit such bias, but are harder to determine and sometimes downright uncomputable. In practice, most approaches use crude MDL. Furthermore, some approaches use only MDL-inspired approaches, departing from its formal information theory formulation to apply practical alternatives based on this principle, such as the work of Lam et al. [Lam+14].

Appendix D

Notable Domains Used in Experiments

We present here graphical representations of the reference domains used for evaluating our approach, as well as some notable learned domains.

These graphical representations are mainly intended to be viewed in the electronic version of this manuscript, due to the small size of certain elements.

Domains from Woodworking are not presented as they would be illegible, even with high zoom levels on a screen.

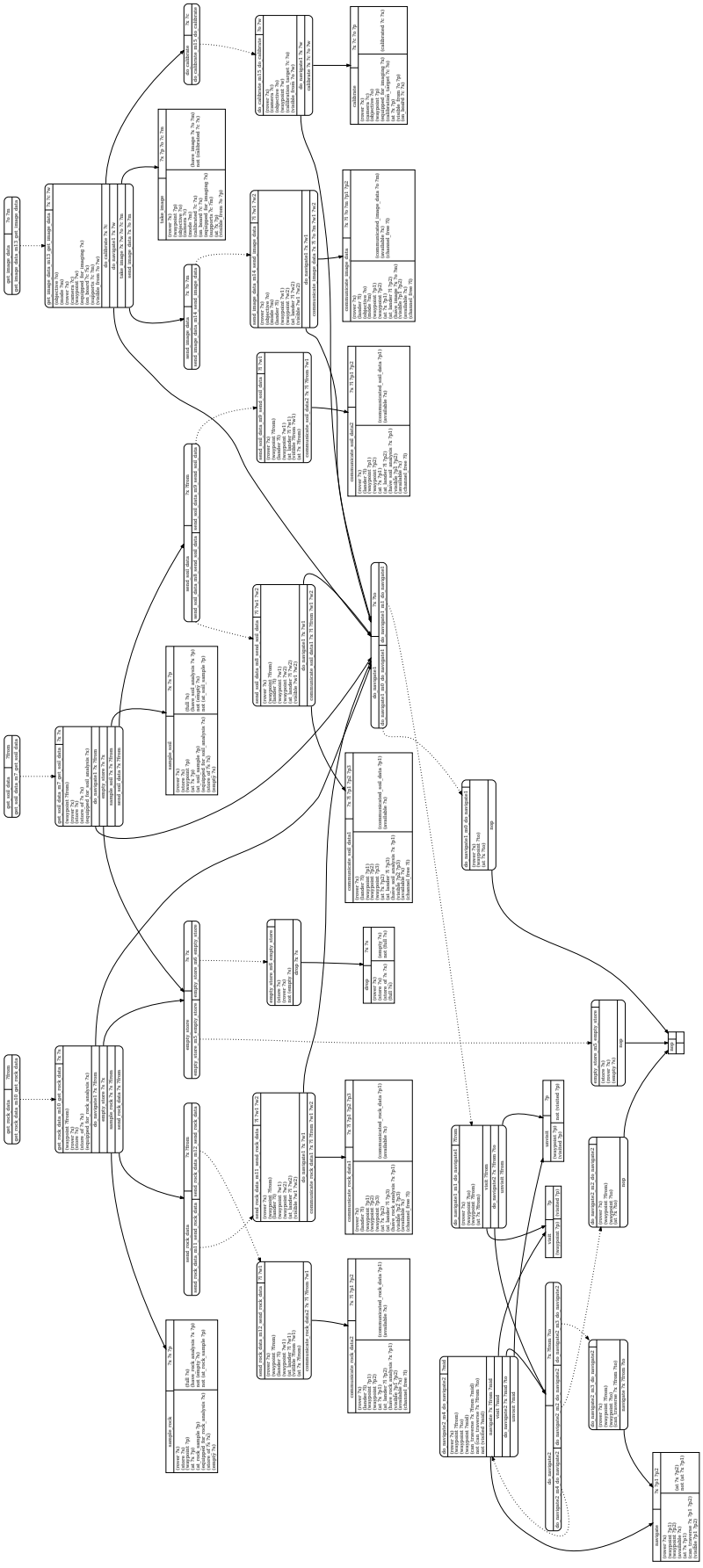


Figure D.1: Graphical representation of the IPC domain in Rovers.

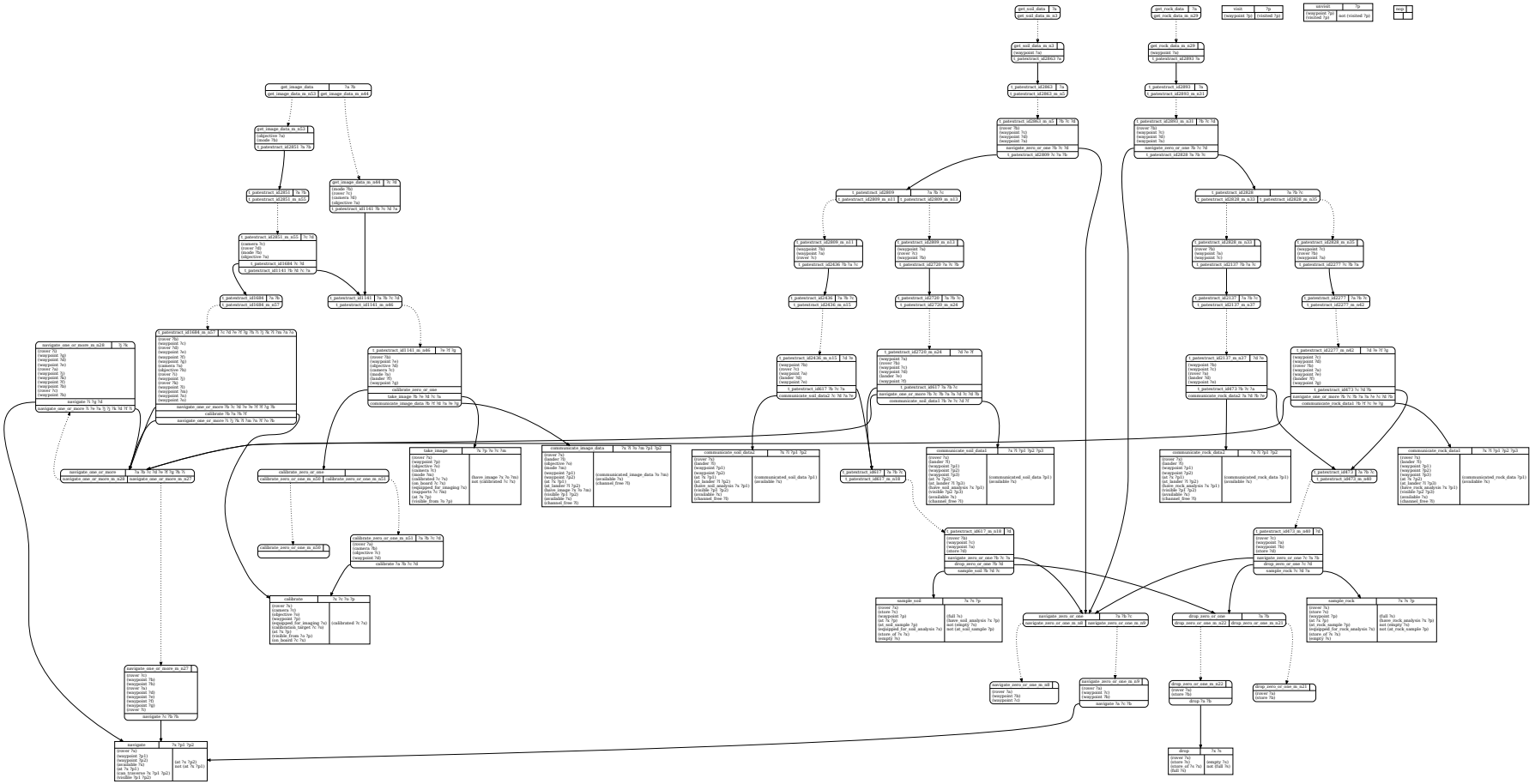


Figure D.2: Graphical representation of the best learned domain in Rovers.

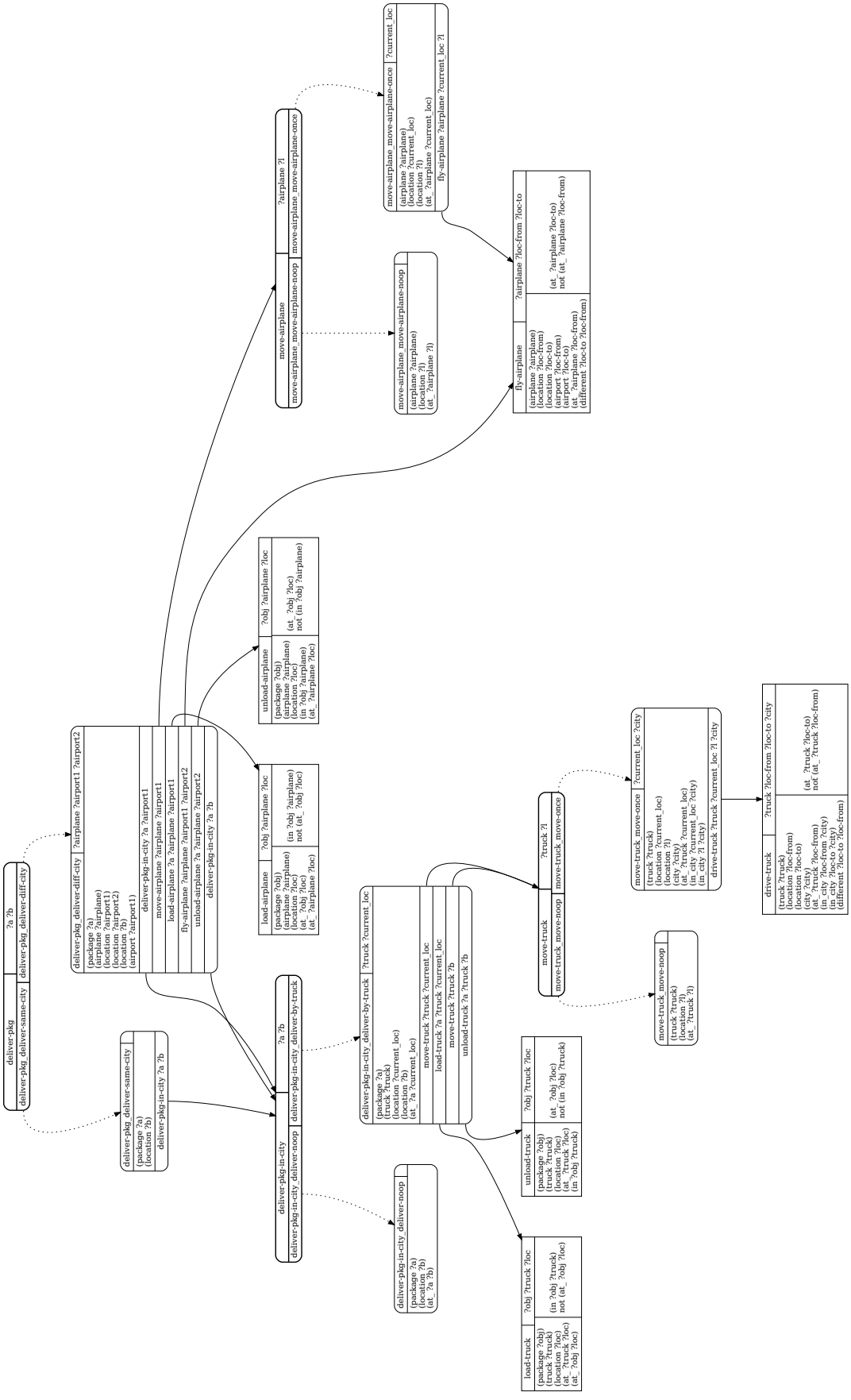


Figure D.3: Graphical representation of the reference domain in LOGISTICS.

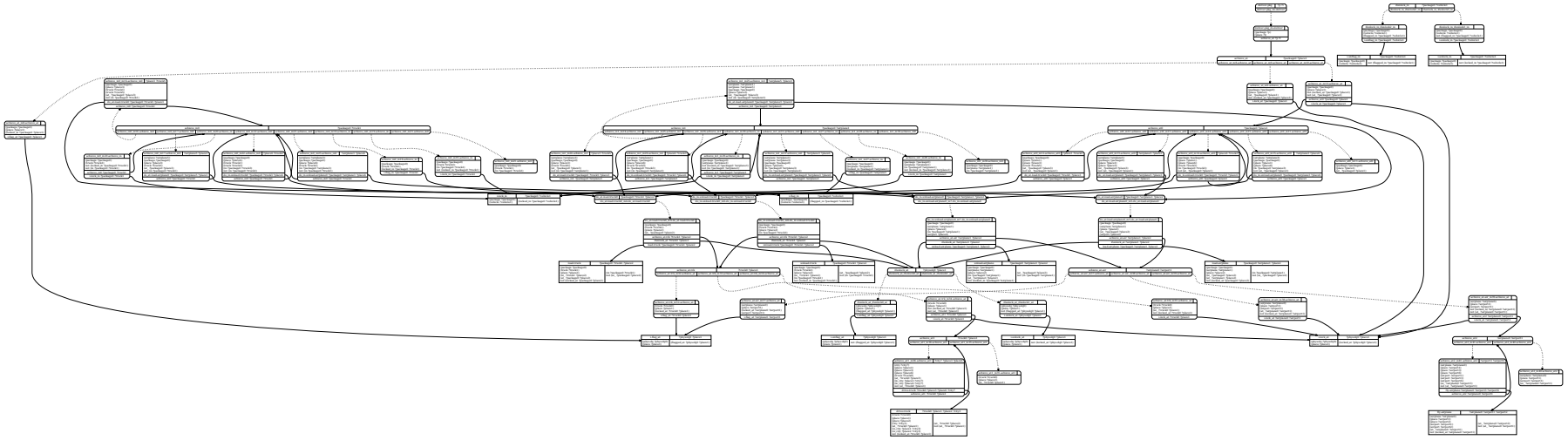


Figure D.4: Graphical representation of the adapted IPC domain in Logistics.

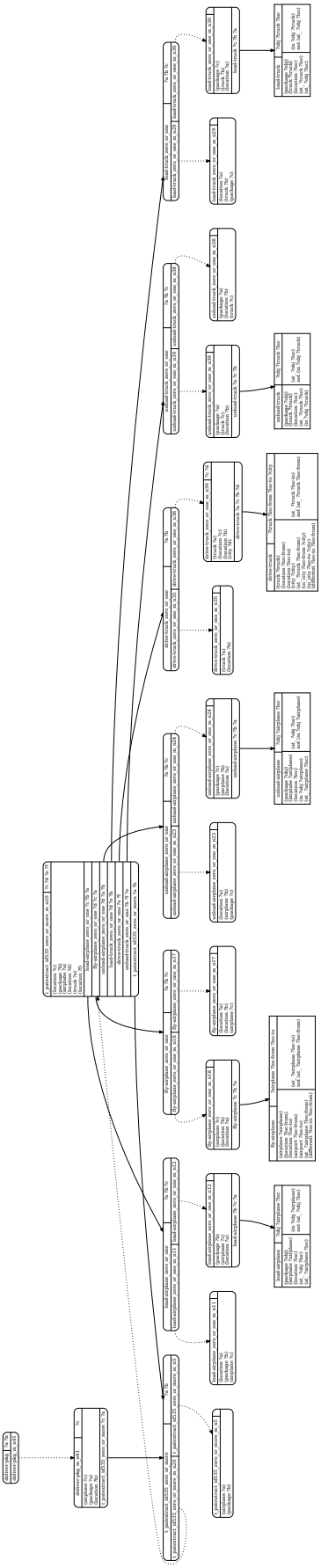


Figure D.5: Graphical representation of the best learned domain in LOGISTICS.

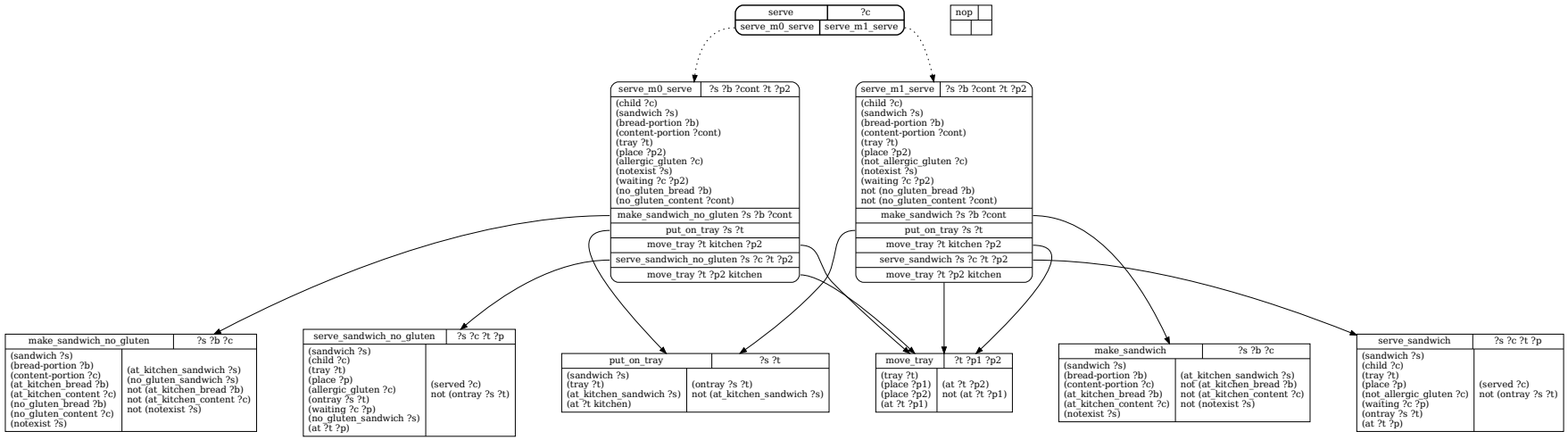


Figure D.6: Graphical representation of the IPC domain in CHILDSNACK.

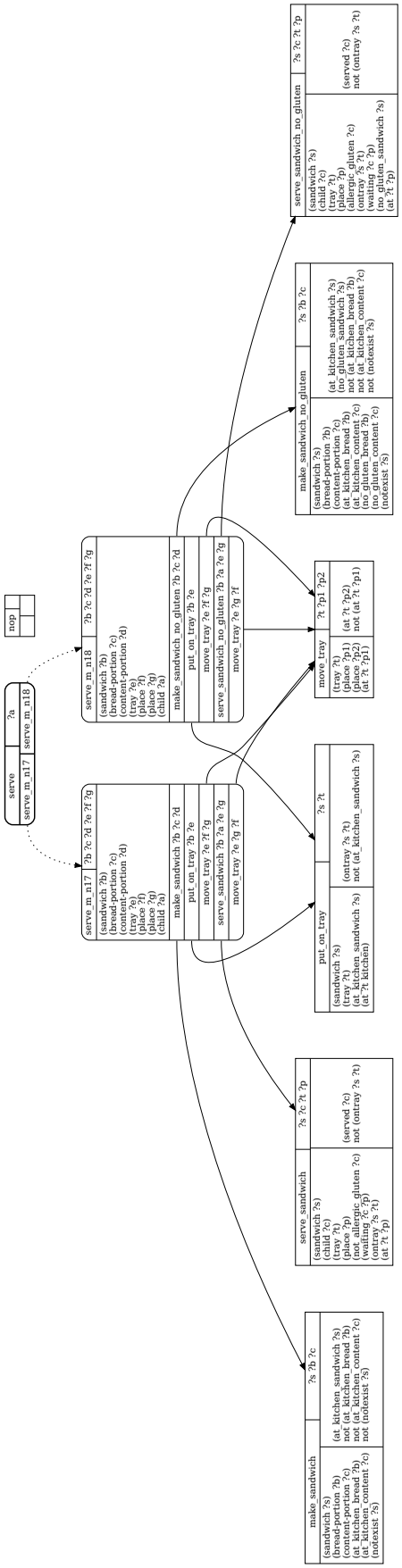


Figure D.7: Graphical representation of the best learned domain in Childsnack.

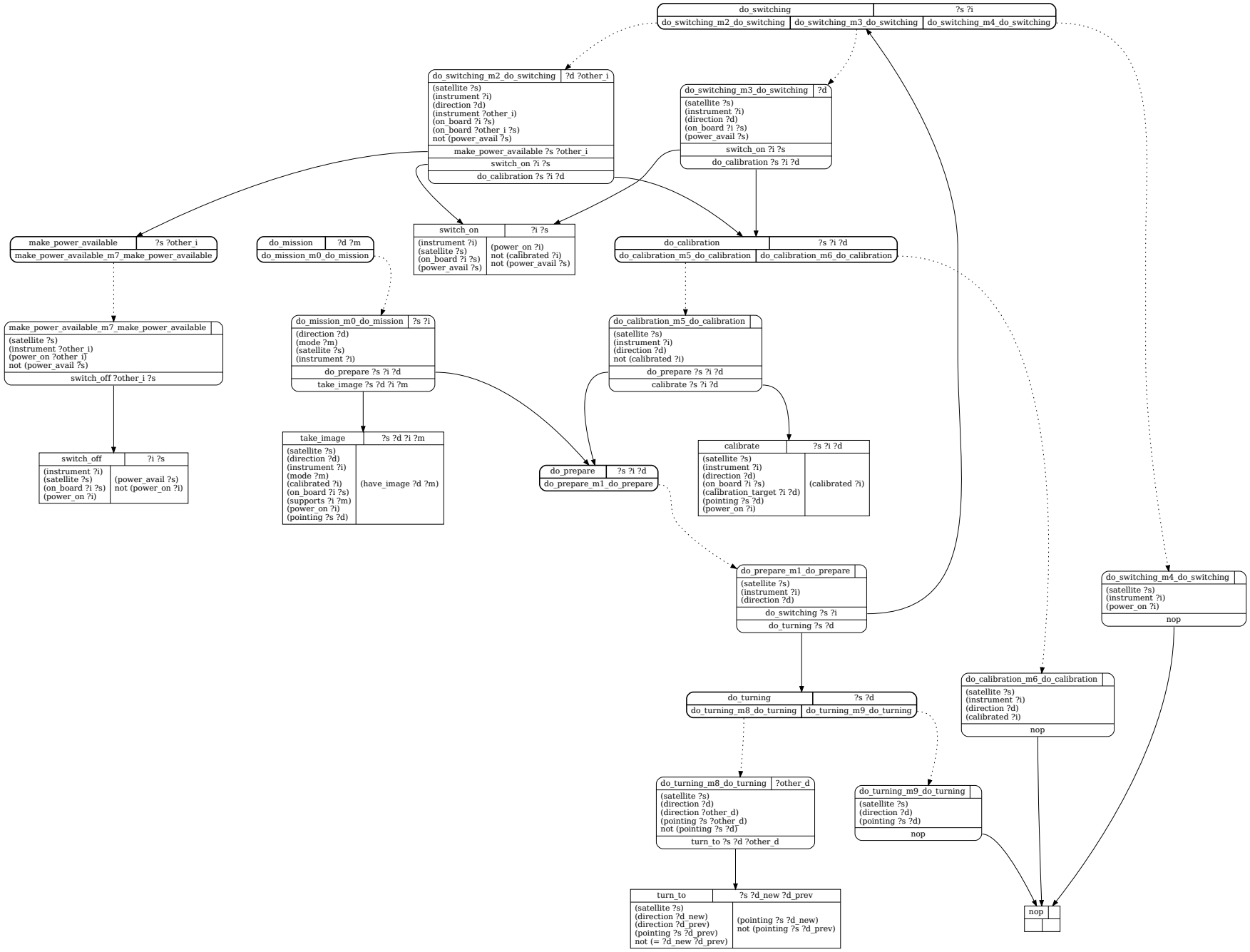


Figure D.8: Graphical representation of the IPC domain in SATELLITE.

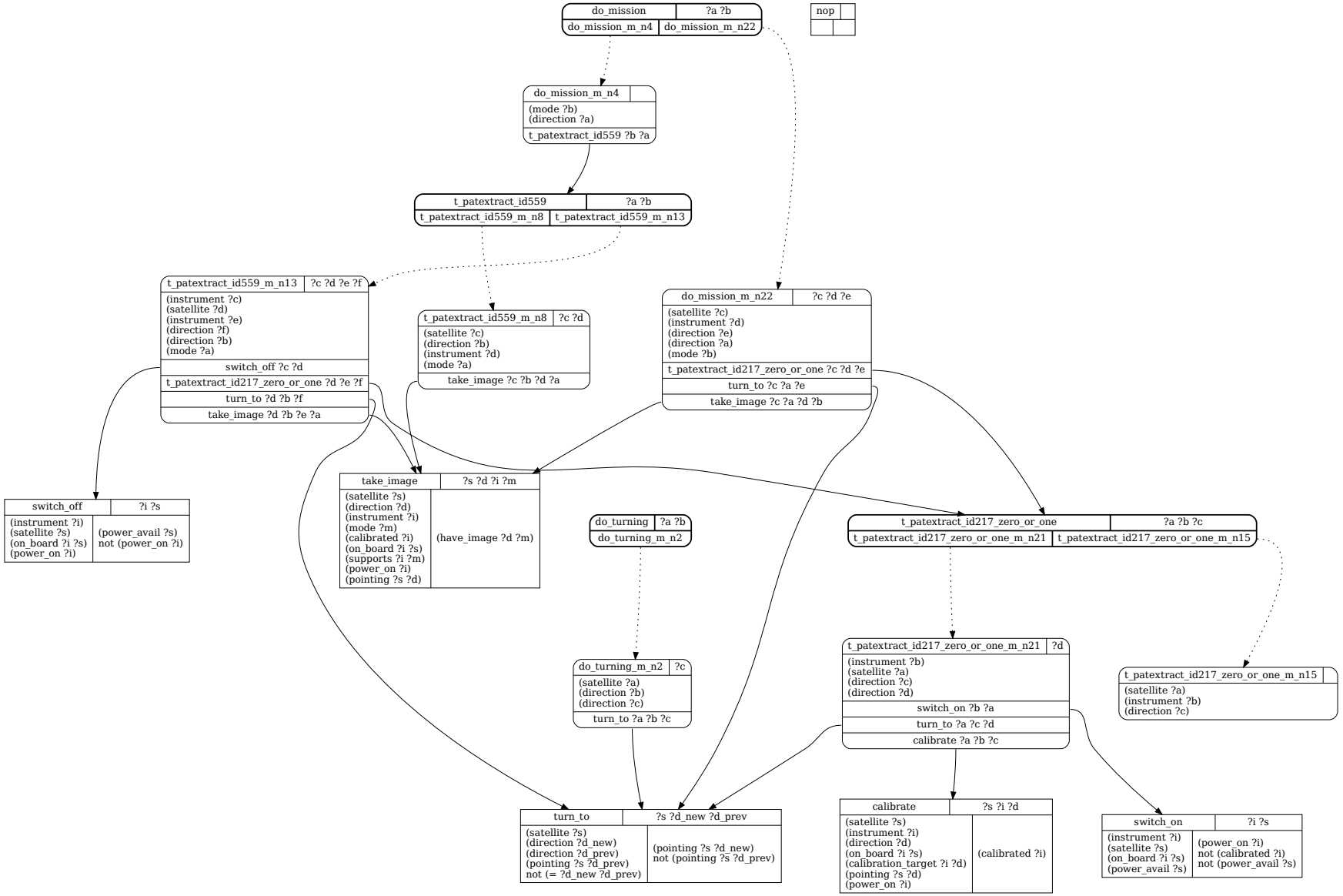


Figure D.9: Graphical representation of the best learned domain in SATELLITE.

References

[ABA15] Ron Alford, Pascal Bercher and David Aha. 'Tight Bounds for HTN Planning'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 25 (8th Apr. 2015), pp. 7–15 (cit. on pp. 6, 10, 14).

[ABH14] Charu C. Aggarwal, Mansurul A. Bhuiyan and Mohammad Al Hasan. 'Frequent Pattern Mining Algorithms: A Survey'. In: *Frequent Pattern Mining*. Ed. by Charu C. Aggarwal and Jiawei Han. Cham: Springer International Publishing, 2014, pp. 19–64 (cit. on pp. 36, 120).

[Agg14] Charu C. Aggarwal. 'An Introduction to Frequent Pattern Mining'. In: *Frequent Pattern Mining*. Ed. by Charu C. Aggarwal and Jiawei Han. Cham: Springer International Publishing, 2014, pp. 1–17 (cit. on p. 47).

[AJO19] Diego Aineto, Sergio Jiménez Celorrio and Eva Onaindia. 'Learning Action Models with Minimal Observability'. In: *Artificial Intelligence* 275 (1st Oct. 2019), pp. 104–137 (cit. on pp. 23, 24).

[AK95] D. Angluin and M. Kharitonov. 'When Won t Membership Queries Help?' In: *Journal of Computer and System Sciences* 50.2 (1st Apr. 1995), pp. 336–355 (cit. on p. 31).

[Aug+19] Adriano Augusto et al. 'Automated Discovery of Process Models from Event Logs: Review and Benchmark'. In: *IEEE Transactions on Knowledge and Data Engineering* 31.4 (Apr. 2019), pp. 686–705 (cit. on p. 35).

[BCF20] Francesco Bariatti, Peggy Cellier and Sébastien Ferré. 'GraphMDL: Graph Pattern Selection Based on Minimum Description Length'. In: *IDA 2020 - Symposium on Intelligent Data Analysis*. Konstanz, Germany, Apr. 2020 (cit. on pp. 47, 141).

[Beh+19] G Behnke et al. 'Hierarchical Planning in the IPC'. In: *Workshop on HTN Planning (ICAPS)*. Berkeley, United States, July 2019 (cit. on p. 12).

[Ben95] Scott Benson. 'Inductive Learning of Reactive Action Models'. In: *Machine Learning Proceedings 1995*. Ed. by Armand Prieditis and Stuart Russell. San Francisco (CA): Morgan Kaufmann, 1st Jan. 1995, pp. 47–54 (cit. on pp. 23, 24).

[BFG19] Blai Bonet, Guillem Francès and Hector Geffner. 'Learning Features and Abstract Actions for Computing Generalized Plans'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (17th July 2019), pp. 2703–2710 (cit. on p. 33).

[BG20] Blai Bonet and Hector Geffner. 'Learning First-Order Symbolic Representations for Planning from the Structure of the State Space.' In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. ECAI. 2020, pp. 2322–2329 (cit. on pp. 23, 25).

[BH12] Yonatan Bisk and Julia Hockenmaier. 'Simple Robust Grammar Induction with Combinatory Categorial Grammars'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 26.1 (1 2012), pp. 1643–1649 (cit. on p. 31).

[BHB19] Gregor Behnke, Daniel Höller and Susanne Biundo. 'Finding Optimal Solutions in HTN Planning - A SAT-based Approach'. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Twenty-Eighth International Joint Conference on Artificial Intelligence {IJCAI-19}. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 5500–5508 (cit. on p. 60).

[BHB21] Gregor Behnke, Daniel Höller and Pascal Bercher, eds. *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*. 2021 (cit. on p. 29).

[Bit23] Arthur Bit-Monnot. 'Experimenting with Lifted Plan-Space Planning as Scheduling: Aries in the 2023 IPC'. In: *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling*. Prague, Czech Republic, July 2023 (cit. on p. 60).

[BMC19] Roman Barták, Adrien Maillard and Rafael C. Cardoso. 'Parsing-Based Approaches for Verification and Recognition of Hierarchical Plans'. In: ICAPS Hierarchical Planning Workshop (HPlan). 20th June 2019 (cit. on p. 31).

[CAÖ16] Michele Colledanchise, Diogo Almeida and Petter Ögren. 'Towards Blended Reactive Planning and Acting Using Behavior Trees'. In: *2019 International Conference on Robotics and Automation (ICRA)* (2016), pp. 8839–8845 (cit. on p. 32).

[Che+21] Kevin Chen et al. 'Learning Hierarchical Task Networks with Preferences from Unannotated Demonstrations'. In: *Proceedings of the 2020 Conference on Robot Learning*. Conference on Robot Learning. PMLR, 4th Oct. 2021, pp. 1572–1581 (cit. on pp. 27, 28, 30, 34, 36, 65, 67).

[CK23] David M. Cerna and Temur Kutsia. 'Anti-Unification and Generalization: A Survey'. In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*. Thirty-Second International Joint Conference on Artificial Intelligence {IJCAI-23}. Macau, SAR China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2023, pp. 6563–6573 (cit. on p. 68).

[CMÖ17] M. Colledanchise, R. M. Murray and P. Ögren. 'Synthesis of Correct-by-Construction Behavior Trees'. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Sept. 2017, pp. 6039–6046 (cit. on p. 32).

[CÖ18] Michele Colledanchise and Petter Ögren. 'Behavior Trees in Robotics and AI: An Introduction'. 20th July 2018 (cit. on p. 32).

[CSL21] Elliot Chane-Sane, Cordelia Schmid and Ivan Laptev. 'Goal-Conditioned Reinforcement Learning with Imagined Subgoals'. In: ICML. arXiv, 2021 (cit. on pp. 2, 118).

[CT12] Michele Chinosi and Alberto Trombetta. 'BPMN: An Introduction to the Standard'. In: *Computer Standards & Interfaces* 34.1 (1st Jan. 2012), pp. 124–134 (cit. on p. 35).

[CVM14] Lukáš Chrpa, Mauro Vallati and Thomas McCluskey. 'MUM: A Technique for Maximising the Utility of Macro-operators by Constrained Generation and Use'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 24 (10th May 2014), pp. 65–73 (cit. on p. 26).

[CVM15] Lukáš Chrpa, Mauro Vallati and Thomas Leo McCluskey. 'On the Online Generation of Effective Macro-Operators'. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 25th July 2015, pp. 1544–1550 (cit. on p. 26).

[DFG11] Arianna D'Ulizia, Fernando Ferri and Patrizia Grifoni. 'A Survey of Grammatical Inference Methods for Natural Language Learning'. In: *Artificial Intelligence Review* 36.1 (1st June 2011), pp. 1–27 (cit. on p. 31).

[DI00] Olivier Despouys and François Félix Ingrand. 'Propice-Plan: Toward a Unified Framework for Planning and Execution'. In: *Recent Advances in AI Planning*. Ed. by Susanne Biundo and Maria Fox. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 278–293 (cit. on pp. 2, 118).

[dMB08] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340 (cit. on p. 139).

[EHN94] K. Erol, J. Hendler and D. Nau. 'HTN Planning: Complexity and Expressivity'. In: AAAI Conference on Artificial Intelligence. 1st Aug. 1994 (cit. on pp. 9, 30, 141).

[FHN72] Richard E Fikes, Peter E Hart and Nils J Nilsson. 'Learning and Executing Generalized Robot Plans'. In: *Artificial Intelligence* 3 (1st Jan. 1972), pp. 251–288 (cit. on p. 26).

[Fin+22] Morgan Fine-Morris et al. 'Learning Decomposition Methods with Numeric Subtasks'. In: ACS. 2022 (cit. on pp. 28, 29).

[Fre+19] Kevin French et al. 'Learning Behavior Trees From Demonstration'. In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019 International Conference on Robotics and Automation (ICRA). May 2019, pp. 7791–7797 (cit. on p. 32).

[GG11] Christopher W. Geib and Robert P. Goldman. 'Recognizing Plans with Loops Represented in a Lexicalized Grammar'. In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, 2011 (cit. on pp. 30, 31).

[Gha+98] Malik Ghallab et al. 'PDDL - The Planning Domain Definition Language'. In: (31st Aug. 1998) (cit. on p. 9).

[Gil94] Yolanda Gil. 'Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains'. In: ICML. Vol. 11. 10th–13th July 1994, pp. 87–95 (cit. on pp. 22, 23).

[GJ20] Antonio Garrido and Sergio Jimenez. 'Learning Temporal Action Models via Constraint Programming'. In: ECAI. 2020, p. 8 (cit. on pp. 23, 25).

[GK18] Christopher W. Geib and Pavan Kantharaju. 'Learning Combinatory Categorial Grammars for Plan Recognition'. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*. AAAI. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 3007–3014 (cit. on p. 31).

[GMK18] Sriram Gopalakrishnan, Héctor Muñoz-Avila and Ugur Kuter. 'Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning'. In: *AI Communications* 31.2 (2nd Mar. 2018). Ed. by Mark Roberts et al., pp. 167–180 (cit. on pp. 27, 29, 65, 68).

[GNT14] Malik Ghallab, Dana Nau and Paolo Traverso. *Automated Planning and Acting*. Cambridge: Cambridge University Press, 2014 (cit. on pp. 1, 2, 6, 117, 118).

[Goe+10] Moritz Goebelbecker et al. 'Coming Up with Good Excuses: What to Do When No Plan Can Be Found'. In: ICAPS. 2010, p. 8 (cit. on p. 25).

[Gol67] E Mark Gold. 'Language Identification in the Limit'. In: *Information and Control* 10.5 (1st May 1967), pp. 447–474 (cit. on p. 31).

[GPF20] Maxence Grand, Damien Pellier and Humbert Fiorino. 'AMLSI: A Novel Accurate Action Model Learning Algorithm'. In: *International Workshop on Knowledge Engineering for Planning and Scheduling (KEPS) during the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*. Nancy, France, Oct. 2020 (cit. on p. 29).

[GPF22] Maxence Grand, Damien Pellier and Humbert Fiorino. 'An Accurate HDDL Domain Learning Algorithm from Partial and Noisy Observations'. In: *Proceedings of the 5th ICAPS Workshop on Hierarchical Planning (HPlan 2022)*. 2022, pp. 1–9 (cit. on pp. 27, 28).

[Grü07] Peter D. Grünwald. *The Minimum Description Length Principle*. Adaptive Computation and Machine Learning. Cambridge, Mass: MIT Press, 2007. 703 pp. (cit. on pp. 47, 141).

[Grü96] Peter Grünwald. 'A Minimum Description Length Approach to Grammar Inference'. In: *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*. Ed. by Stefan Wermter, Ellen Riloff and Gabriele Scheler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 203–216 (cit. on p. 31).

[HB23] Philippe Hérail and Arthur Bit-Monnot. 'Leveraging Demonstrations for Learning the Structure and Parameters of Hierarchical Task Networks'. In: The 36th International FLAIRS Conference. Vol. 36. 14th May 2023 (cit. on pp. 49, 61).

[HD11] Yuxiao Hu and Giuseppe De Giacomo. 'Generalized Planning: Synthesizing Plans That Work for Multiple Environments'. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 16th July 2011, pp. 918–923 (cit. on pp. 32, 33).

[HKM09] Chad Hogg, Ugur Kuter and Héctor Muñoz-Avila. 'Learning Hierarchical Task Networks for Nondeterministic Planning Domains'. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 1708–1714 (cit. on pp. 27, 28).

[HKM10] Chad Hogg, Ugur Kuter and Hector Muñoz-Avila. 'Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning'. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. Ed. by Maria Fox and David Poole. AAAI Press, 2010 (cit. on p. 28).

[HMK08] 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 - Volume 2*. AAAI'08. Chicago, Illinois: AAAI Press, 13th July 2008, pp. 950–956 (cit. on pp. 27, 28, 34, 36, 40, 52, 62, 67, 120).

[Hog11] Chad Hogg. 'Learning Hierarchical Task Networks from Traces and Semantically Annotated Tasks'. Lehigh, 2011 (cit. on p. 67).

[Höl+18] D. Höller et al. 'Plan and Goal Recognition as HTN Planning'. In: *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. ICTAI. Nov. 2018, pp. 466–473 (cit. on p. 31).

[Höl+20] Daniel Höller et al. 'HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.06 (06 3rd Apr. 2020), pp. 9883–9891 (cit. on p. 11).

[Höl+21] Daniel Höller et al. 'Compiling HTN Plan Verification Problems into HTN Planning Problems'. In: *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)*. 2021, pp. 8–15 (cit. on p. 58).

[HS16] B. Hayes and B. Scassellati. 'Autonomously Constructing Hierarchical Task Networks for Planning and Human-Robot Collaboration'. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 5469–5476 (cit. on pp. 27, 30, 36, 65, 67).

[Hsi+22] Eric Hsiung et al. 'Generalizing to New Domains by Mapping Natural Language to Lifted LTL'. In: *2022 International Conference on Robotics and Automation (ICRA)*. Philadelphia, PA, USA: IEEE Press, 23rd May 2022, pp. 3624–3630 (cit. on p. 68).

[Ing+96] F.F. Ingrand et al. 'PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots'. In: *Proceedings of IEEE International Conference on Robotics and Automation*. Proceedings of IEEE International Conference on Robotics and Automation. Vol. 1. Apr. 1996, 43–49 vol.1 (cit. on pp. 2, 118).

[Jha+10] Susmit Jha et al. 'Oracle-Guided Component-Based Program Synthesis'. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. New York, NY, USA: Association for Computing Machinery, 1st May 2010, pp. 215–224 (cit. on p. 34).

[JJ15] Sergio Jiménez and Anders Jonsson. 'Computing Plans with Control Flow and Procedures Using a Classical Planner'. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 6. 1. 2015, pp. 62–69 (cit. on p. 34).

[JLS21] Brendan Juba, Hai S. Le and Roni Stern. 'Safe Learning of Lifted Action Models'. In: *Proceedings of the Eighteenth International Conference on Principles of Knowledge Representation and Reasoning*. 18th International Conference on Principles of Knowledge Representation and Reasoning {KR-2021}. Hanoii, Vietnam: International Joint Conferences on Artificial Intelligence Organization, Sept. 2021, pp. 379–389 (cit. on p. 25).

[JS22] Brendan Juba and Roni Stern. 'Learning Probably Approximately Complete and Safe Action Models for Stochastic Worlds'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.9 (9 28th June 2022), pp. 9795–9804 (cit. on pp. 23, 25).

[JSJ19] Sergio Jiménez, Javier Segovia-Aguas and Anders Jonsson. 'A Review of Generalized Planning'. In: *The Knowledge Engineering Review* 34 (2019), e5 (cit. on pp. 32, 33).

[KBK07] John-Paul Kelly, Adi Botea and Sven Koenig. 'Planning with Hierarchical Task Networks in Video Games'. In: ICAPS Workshop on Planning in Games. 2007 (cit. on p. 29).

[Kle+20] Kilian Kleeberger et al. 'A Survey on Learning-Based Robotic Grasping'. In: *Current Robotics Reports* 1.4 (1st Dec. 2020), pp. 239–249 (cit. on pp. 2, 118).

[KOG19] Pavan Kantharaju, Santiago Ontañón and Christopher W. Geib. 'Extracting CCGs for Plan Recognition in RTS Games'. In: *Proceedings of the 2nd Workshop on Knowledge Extraction from Games Co-Located with 33rd AAAI Conference on Artificial Intelligence, KEG@AAAI 2019, Honolulu, Hawaii, January 27th, 2019*. Ed. by Matthew Guzdial, Joseph C. Osborn and Sam Snodgrass. Vol. 2313. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 9–16 (cit. on p. 31).

[KS20] Beomjoon Kim and Luke Shimanuki. 'Learning Value Functions with Relational State Representations for Guiding Task-and-Motion Planning'. In: *Conference on Robot Learning*. Conference on Robot Learning. PMLR, 12th May 2020, pp. 955–968 (cit. on p. 34).

[Lam+14] Hoang Thanh Lam et al. 'Mining Compressing Sequential Patterns'. In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7.1 (2014), pp. 34–52 (cit. on pp. 47, 50, 142).

[Lem+17] Séverin Lemaignan et al. 'Artificial Cognition for Social Human–Robot Interaction: An Implementation'. In: *Artificial Intelligence*. Special Issue on AI and Robotics 247 (1st June 2017), pp. 45–69 (cit. on pp. 2, 118).

[LFA14] Sander Leemans, Dirk Fahland and Wil Aalst. 'Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour'. In: vol. 171. 10th May 2014, pp. 66–78 (cit. on p. 35).

[LFvdA13a] Sander J. J. Leemans, Dirk Fahland and Wil M. P. van der Aalst. 'Discovering Block-Structured Process Models from Event Logs - A Constructive Approach'. In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 311–329 (cit. on p. 29).

[LFvdA13b] Sander J. J. Leemans, Dirk Fahland and Wil M. P. van der Aalst. 'Discovering Block-Structured Process Models from Event Logs - A Constructive Approach'. In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 311–329 (cit. on p. 35).

[Li+14] Nan Li et al. 'Learning Probabilistic Hierarchical Task Networks as Probabilistic Context-Free Grammars to Capture User Preferences'. In: *ACM Transactions on Intelligent Systems and Technology* 5.2 (2014), p. 32 (cit. on pp. 27–31, 34, 36, 61, 65, 67).

[LJ16] 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. NLD: IOS Press, 29th Aug. 2016, pp. 1274–1282 (cit. on pp. 27, 29, 34, 65, 68).

[Lot17] Damir Lotinac. 'Novel Approaches for Generalized Planning'. PhD thesis. Universitat Pompeu Fabra, 14th Dec. 2017 (cit. on pp. 32, 34).

[LS00] Pat Langley and Sean Stromsten. 'Learning Context-Free Grammars with a Simplicity Bias'. In: *Machine Learning: ECML 2000*. Ed. by Ramon López de Mántaras and Enric Plaza. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 220–228 (cit. on p. 31).

[LTK12] Tobias Lang, Marc Toussaint and Kristian Kersting. 'Exploration in Relational Domains for Model-Based Reinforcement Learning'. In: *The Journal of Machine Learning Research* 13.1 (1st Dec. 2012), pp. 3725–3768 (cit. on p. 25).

[LV90] Ming Li and Paul M.B. Vitányi. 'Kolmogorov Complexity and Its Applications'. In: *Algorithms and Complexity*. Elsevier, 1990, pp. 187–254 (cit. on pp. 141, 142).

[MAT15] David Martínez, Guillem Alenyà and Carme Torras. 'Relational Reinforcement Learning with Guided Demonstrations'. In: *Artificial Intelligence*. Special Issue on AI and Robotics 247 (2015), pp. 295–312 (cit. on pp. 23, 25).

[MCA22] Amandine Mayima, Aurélie Clodic and Rachid Alami. 'JAHRVIS, a Supervision System for Human-Robot Collaboration'. In: *2022 31st IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. 2022 31st IEEE International Conference on Robot and Human Interactive Communication (RO-MAN). Aug. 2022, pp. 777–784 (cit. on pp. 2, 118).

[McD00] Drew M. McDermott. 'The 1998 AI Planning Systems Competition'. In: *AI Magazine* 21.2 (2 15th June 2000), pp. 35–35 (cit. on pp. 12, 39).

[MG04] Mario Martín and Hector Geffner. 'Learning Generalized Policies from Planning Examples Using Concept Languages'. In: *Applied Intelligence* 20.1 (1st Jan. 2004), pp. 9–19 (cit. on p. 33).

[Mik+13] Tomas Mikolov et al. 'Distributed Representations of Words and Phrases and Their Compositionality'. In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013 (cit. on p. 29).

[Min85] Steven Minton. 'Selectively Generalizing Plans for Problem-Solving'. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*. Ed. by Aravind K. Joshi. Morgan Kaufmann, 1985, pp. 596–599 (cit. on p. 26).

[MMdS21] Maurício Magnaguagno, Felipe Meneguzzi and Lavindra de Silva. 'HyperTensioN: A Three-Stage Compiler for Planning'. In: IPC 2020. 1st Jan. 2021 (cit. on p. 113).

[Moo88] Raymond J. Mooney. 'Generalizing the Order of Operators in Macro-Operators'. In: 1988 (cit. on p. 26).

[Mou+12] Kira Mourao et al. 'Learning STRIPS Operators from Noisy and Incomplete Observations'. 16th Oct. 2012 (cit. on pp. 23, 24).

[New+07] M. A. Hakim Newton et al. 'Learning Macro-Actions for Arbitrary Planners and Domains'. In: International Conference on Automated Planning and Scheduling. 22nd Sept. 2007 (cit. on p. 26).

[Ngu+17] Chanh Nguyen et al. 'Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents'. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017 IEEE Conference on Computational Intelligence and Games (CIG). Aug. 2017, pp. 226–231 (cit. on p. 29).

[NLK06] Negin Nejati, Pat Langley and Tolga Konik. 'Learning Hierarchical Task Networks by Observation'. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. New York, NY, USA: Association for Computing Machinery, 25th June 2006, pp. 665–672 (cit. on pp. 26, 27).

[NMB18] Xenija Neufeld, Sanaz Mostaghim and Sandy Brand. 'A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees'. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14.1 (1 25th Sept. 2018) (cit. on p. 32).

[OC96] T. Oates and P. Cohen. 'Searching for Planning Operators with Context-Dependent and Probabilistic Effects'. In: AAAI/IAAI, Vol. 1. 4th Aug. 1996 (cit. on pp. 23, 24).

[PB23] Kristýna Pantůčková and Roman Barták. 'Using Earley Parser for Recognizing Totally Ordered Hierarchical Plans'. In: *ECAI 2023*. ECAI. Kraków, Poland: IOS Press, 2023, pp. 1819–1826 (cit. on p. 61).

[Per+11] Diego Perez et al. 'Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution'. In: *Applications of Evolutionary Computation*. Ed. by Cecilia Di Chio et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 123–132 (cit. on p. 32).

[Pet+04] G. Petasis et al. 'E-GRIDS: Computationally Efficient Gramatical Inference from Positive Examples'. In: *Grammars* (2004) (cit. on p. 31).

[Plo70] Gordon D Plotkin. 'A Note on Inductive Generalization'. In: *Machine intelligence* 5.1 (1970), pp. 153–163 (cit. on p. 65).

[PZK07] H. M. Pasula, L. S. Zettlemoyer and L. P. Kaelbling. 'Learning Symbolic Models of Stochastic Domains'. In: *Journal of Artificial Intelligence Research* 29 (21st July 2007), pp. 309–352 (cit. on pp. 23, 25).

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993 (cit. on p. 32).

[RGK17] F. Rovida, B. Grossmann and V. Krüger. 'Extended Behavior Trees for Quick Definition of Flexible Robotic Tasks'. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Sept. 2017, pp. 6793–6800 (cit. on p. 32).

[Ris78] J. Rissanen. 'Modeling by Shortest Data Description'. In: *Automatica* 14.5 (1st Sept. 1978), pp. 465–471 (cit. on p. 141).

[Rod+21] Ivan D. Rodriguez et al. 'Learning First-Order Representations for Planning from Black Box States: New Results'. In: *Proceedings of the Eighteenth International Conference on Principles of Knowledge Representation and Reasoning*. 18th International Conference on Principles of Knowledge Representation and Reasoning {KR-2021}. Hanoii, Vietnam: International Joint Conferences on Artificial Intelligence Organization, Sept. 2021, pp. 539–548 (cit. on pp. 23, 25).

[RT97] Chandra Reddy and Prasad Tadepalli. 'Learning Goal-Decomposition Rules Using Exercises'. In: *Proceedings of the Fourteenth International Conference on Machine Learning*. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 8th July 1997, pp. 278–286 (cit. on p. 26).

[RW15] Glen Robertson and Ian Watson. 'Building Behavior Trees from Observations in Real-Time Strategy Games'. In: *2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA)*. 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA). Sept. 2015, pp. 1–7 (cit. on p. 32).

[SBG22] Simon Ståhlberg, Blai Bonet and Hector Geffner. 'Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits'. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 32. 13th June 2022, pp. 629–637 (cit. on p. 34).

[SBS12] Upendra Sapkota, Barrett R. Bryant and Alan Sprague. 'Unsupervised Grammar Inference Using the Minimum Description Length Principle'. In: *Machine Learning and Data Mining in Pattern Recognition*. Ed. by Petra Perner. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 141–153 (cit. on pp. 31, 47, 141).

[Sch21] Dominik Schreiber. 'Lilotane: A Lifted SAT-based Approach to Hierarchical Planning'. In: *Journal of Artificial Intelligence Research* 70 (17th Mar. 2021), pp. 1117–1181 (cit. on pp. 61, 87, 113).

[SdSP06] Sebastian Sardina, Lavindra de Silva and Lin Padgham. 'Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach'. In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '06. New York, NY, USA: Association for Computing Machinery, 8th May 2006, pp. 1001–1008 (cit. on pp. 2, 118).

[Shi+12] Vikas Shivashankar et al. '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*. 2012, pp. 981–988 (cit. on pp. 26, 29, 30).

[SIZ11] Siddharth Srivastava, Neil Immerman and Shlomo Zilberstein. 'A New Representation and Associated Algorithms for Generalized Planning'. In: *Artificial Intelligence* 175.2 (1st Feb. 2011), pp. 615–647 (cit. on p. 32).

[Sol09] Armando Solar-Lezama. 'The Sketching Approach to Program Synthesis'. In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Red. by David Hutchison et al. Vol. 5904. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 4–13 (cit. on p. 34).

[SPF17] Jose 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. Pittsburgh, United States, 19th June 2017 (cit. on pp. 27, 29, 36, 65, 67, 69).

[Sri10] Siddharth Srivastava. 'Foundations and Applications of Generalized Planning'. 2010 (cit. on p. 33).

[SRU16] Shirin Sohrabi, Anton V. Riabov and Octavian Udrea. 'Plan Recognition as Planning Revisited'. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 9th July 2016, pp. 3258–3264 (cit. on p. 31).

[TB22] Jérémy Turi and Arthur Bit-Monnot. 'Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner'. In: ICAPS Workshop on Integrated Planning, Acting, and Execution (IntEx). 17th June 2022 (cit. on pp. 2, 118).

[Van12] Wil Van Der Aalst. 'Process Mining: Overview and Opportunities'. In: *ACM Transactions on Management Information Systems* 3.2 (July 2012), pp. 1–17 (cit. on pp. 29, 35).

[vZel+21] Sebastiaan J. van Zelst et al. 'Event Abstraction in Process Mining: Literature Review and Taxonomy'. In: *Granular Computing* 6.3 (1st July 2021), pp. 719–736 (cit. on p. 35).

[Wan95] Xuemei Wang. 'Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition'. In: *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 549–557 (cit. on pp. 22, 23).

[YWJ07] Qiang Yang, Kangheng Wu and Yunfei Jiang. 'Learning Action Models from Plan Examples Using Weighted MAX-SAT'. In: *Artificial Intelligence* 171.2 (1st Feb. 2007), pp. 107–143 (cit. on pp. 23, 24, 28).

[ZC05] Luke S. Zettlemoyer and Michael Collins. 'Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars'. In: *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. UAI'05. Arlington, Virginia, USA: AUAI Press, 26th July 2005, pp. 658–666 (cit. on p. 31).

[Zha+18] Qi Zhang et al. 'Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution'. In: *Applied Sciences* 8.7 (7 July 2018), p. 1077 (cit. on p. 32).

[Zhu+09] Hankz Hankui Zhuo et al. 'Learning HTN Method Preconditions and Action Models from Partial Observations'. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 1804–1810 (cit. on p. 28).

[ZMY14] Hankz Hankui Zhuo, Héctor Muñoz-Avila and Qiang Yang. 'Learning Hierarchical Task Network Domains from Partially Observed Plan Traces'. In: *Artificial Intelligence* 212 (1st July 2014), pp. 134–157 (cit. on pp. 27, 28, 34, 67, 68).