Novel approaches for generalized planning

Damir Lotinac

TESI DOCTORAL UPF / ANY 2017

DIRECTOR DE LA TESI

Prof. Dr. Anders Jonsson, Department of Information and Communication Technologies

To my family

Acknowledgements

This thesis would not have been written without the support of my advisor Anders Jonsson. I am sincerely grateful for his guidance, patience and encouragement throughout my work on the thesis.

My office mates created a friendly and supportive environment. I express my gratitude to Javier Segovia, Oussam Larkem and Guillem Frances for making my stay at UPF more pleasurable.

During my stay in Barcelona I was lucky to share a flat with Marina Corbera, Tardor and Iru, who have been the best flatmates I could ever ask for.

All the stress would probably be unbearable, without the optimism of Filippos Kominis, Jonathan Ferrer and Bruno Paun who have been a tremendous support all these years. I am also grateful to Dunja Veselinovic and Denis Salkanovic who never let the distance affect our friendship.

I thank Elvira Lakovic, without whom this journey would not have had started at all.

Finally I owe my gratitude to my parents and sister, who have always supported and encouraged me.

Abstract

Classical planning is the problem of finding a sequence of actions, from a given initial state to some goal state. While, in generalized planning, a plan is a solution to a set of planning problems, which belong to the same class. In this thesis we explore novel ways of computing generalized plans, inductively from a set of examples, and deductively from a model of actions.

First we present an extension of planning programs, as a representation of a generalized plan, which is induced from a set of examples. The extension, allows for modeling of classification tasks.

This work also introduces a novel domain-independent algorithm for generating hierarchical task networks directly from the action model and one representative instance of the planning problem. We also present the optimizations used by the translation and show that the algorithm is competitive with the state-of-the-art algorithms.

Resum

La planificació clàssica és un problema que consisteix en trobar una seqüència d'accions, que va des d'un estat inicial fins un objectiu. Per una altra banda en planificació general, un plà és una solució a un conjunt de problemes de planificació els quals pertanyen a una mateixa classe. En aquesta tesi explorem camins nous per obtenir plans generals de forma inductiva sobre un conjunt d'exemples, i de forma deductiva sobre un model d'accions.

Primer presentem una extensió per programes de planificació, com una representació d'un plà general, el qual es induït d'un conjunt d'exemples. L'extensió permet modelar una tasca de classificació.

Aquest treball també introdueix un nou algorisme, que és independent de domini, que genera una xarxa de tasques jeràrquiques directament d'un model d'accions i d'una instància d'un problema de planificació. També presentem les optimitzacions utilitzades en la traducció i mostrem que aquest algorisme és competitiu amb l'estat de l'art.

Preface

Classical planning is the problem of finding a sequence of actions, which leads from an initial state to some goal state. The effects of the actions are deterministic and the states are fully observable. It is equivalent to a search problem in an implicitly defined directed graph, where states represent the nodes, and actions represent the edges. To tackle the complexity arising from the number of states, modern planners usually rely on heuristics. The heuristics are usually computed from a relaxed version of the original planning problem. However, such planners find solutions for a single instance of the problem. Each time a classical planner solves a planning instance, the search begins without any prior knowledge.

A generalized plan is a solution to a family of planning problems. Two types of generalized plans can be distinguished. One can just be executed without search, while the other reduces the search space by encoding the knowledge derived from the commonalities among the planning problems. There are many different representations used to encode the generalized plans. In the literature these representations range from decision lists (Martín and Geffner, 2004), finite-state controllers (Bonet et al., 2010), to programs (Jiménez and Jonsson, 2015).

In this thesis we explore novel ways to compute generalized plans. We investigate how to generate relevant features, in tight coupling with the computation of the generalized plan. There are two major approaches to synthesizing a generalized plan. One is to either induce a plan from a set of examples. Another is to generate the generalized plan by analyzing the action model, which captures the commonalities of the family of planning problems. The first approach is often referred to as *inductive*, while the latter is called *deductive*.

In Part II, we introduce an extension to planning programs, which enhances the expressive power of basic planning programs. Planning pro-

grams are a representation of a generalized plan. These are programs that consist of sequential, goto and termination instructions. The goto instructions are used for branching, however in the basic version, the condition for branching can only be a fluent. In this work we explore the use of conjunctive queries to encode high-level state features, which in turn allows for higher expressiveness. This extension is applied to the branching conditions. In this part we show how the enhanced planning programs enable modeling of supervised classification tasks. We also show a novel approach for generating high-level state features, in tight coupling with the induction of the planning program.

In Part III of this thesis, we introduce automatically generated hierarchical task networks (HTNs). HTNs can be seen as another representation of the generalized plan, since such hierarchies capture knowledge about the whole planning domain. The HTNs presented in this work are generated from planning domain in PDDL format and one representative planning instance. We show that a lifted representation of invariant graphs can be used to analyze the planning domain, and generate HTNs which encode knowledge about the domain as a whole. While other approaches induce the hierarchical task networks from examples, we present a compilation directly from a PDDL domain to HTN. This compilation is a novel way to deductively generate a generalized plan. In this part, we compare the HTN planner using the generated HTNs with Fast Downward blind search algorithm, since both the HTN planner uses blind search as well. We show that the HTN planner with the generated HTNs, constructed from lifted invariant graphs, are competitive with Fast Downward blind search algorithm.

Further, in Part III, we explore optimizations which can be imposed over the generated HTN. The optimizations we present are domain-independent and are used to either constrain the search space, or guide the task-subtask decompositions. We present a goal ordering optimization, which which specifies how to order the goal fluents of the original planning instance. We also introduce an ordering of the invariant graphs, which enables more efficient selection of an invariant graph in which the search is to be per-

formed. Finally, we introduce an optimization which guides the search by sorting the bindings of the free variables of the HTN decomposition methods. We show that fully optimized generated HTNs are competitive with HTNs which are induced from a set of examples, by a state-of-the-art HTN generator. Compared to existing approaches, the input of our algorithm is significantly simpler and solves instances that are much larger.

Some of the work presented in this thesis, has previously been published in the following articles:

- *Jonsson, A., & Lotinac, D. (2015). Automatic Generation of HTNs From PDDL. ICAPS Workshop, Planning and Learning (PAL-15), 15.* [Chapter 4]
- *Lotinac, D. & Jonsson, A. (2015). Automatic Generation of HTNs From PDDL. Proceedings of the 2nd Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM'15)* [Chapter 4]
- *Lotinac, D., Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2016). Automatic generation of high-level state features for generalized planning. In Proceedings of the 25th International Joint Conference on Artificial Intelligence; 2016 July 9-15; New York, United States. Palo Alto: AAAI Press; 2016. p. 3199-3205.. Association for the Advancement of Artificial Intelligence (AAAI).* [Chapter 3]
- *Lotinac, D., & Jonsson, A. (2016). Constructing Hierarchical Task Models Using Invariance Analysis. In ECAI (pp. 1274-1282).* [Chapter 4, Chapter 5]
- *Lotinac, D., & Jonsson, A. Generating Hierarchical Task Networks. Submitted to Journal of Artificial Intelligence (JAIR)* [Chapter 4, Chapter 5]

Contents

Abstract **vii**

Resum **viii**

Preface **ix**

List of Figures **xvii**

List of Tables **xix**

I Background **1**

1 CLASSICAL PLANNING **3**
1.1 Modeling languages 3
1.2 Planning domains and instances 5
1.3 Classical planning model 8
1.3.1 Classical planning problem 8
1.3.2 Classical planning with conditional effects . . . 10
1.3.3 Multivalued representations 11
1.4 Invariants 12
1.5 Abstractions in planning 13

2 GENERALIZED PLANNING **15**
2.1 Introduction 15

2.2 Planning Programs 19
2.2.1 Basic Planning Programs 19
2.3 Hierarchies in Planning 21
2.3.1 Hierarchical Task Networks 21
2.3.2 SHOP2 modeling language 24

II High-Level state features 29

3 PLANNING PROGRAMS WITH HIGH-LEVEL STATE FEATURES 31
3.1 Introduction 31
3.2 Generating High-Level State Features 34
3.2.1 High-Level State Features 34
3.2.2 Planning Programs with Conjunctive Queries . . 37
3.2.3 Computing Planning Programs with Conjunctive Queries 38
3.3 Properties of Planning Programs with conjunctive queries 42
3.4 Classification with Planning Programs 43
3.5 Evaluation 45
3.6 Discussion 47

III Generating HTNs 49

4 GENERATING HTNS 51
4.1 Introduction 51
4.2 Invariant graphs 54
4.3 Translation algorithm 56
4.3.1 Predicates 58
4.3.2 Primitive tasks 58
4.3.3 Compound tasks 59
4.3.4 Methods 60
4.3.5 Planning Instances 65
4.3.6 Example 66

4.4 Ordering Preconditions 69
4.5 Properties of the generated HTNs 70
4.6 Experimental results 74
4.7 Discussion 76

5 OPTIMIZATIONS OF GENERATED HTNS 79
5.1 Introduction 79
5.2 Goal Ordering 81
5.3 Ordering the invariant graphs 84
5.4 Sorting the bindings of the free variables 86
5.5 Experimental Results 87
5.6 Discussion 90

IV Related Work 93

6 RELATED WORK 95
6.1 High-Level State Features 95
6.2 Generating HTNs 96

V Conclusions and Future Work 99

7 CONCLUSIONS 101
7.1 Contributions 101
7.2 Future work 102
7.2.1 Planning Programs with High-Level State Features 103
7.2.2 Generating Hierarchical Task Networks 104

VI Appendix 107

Bibliography 109

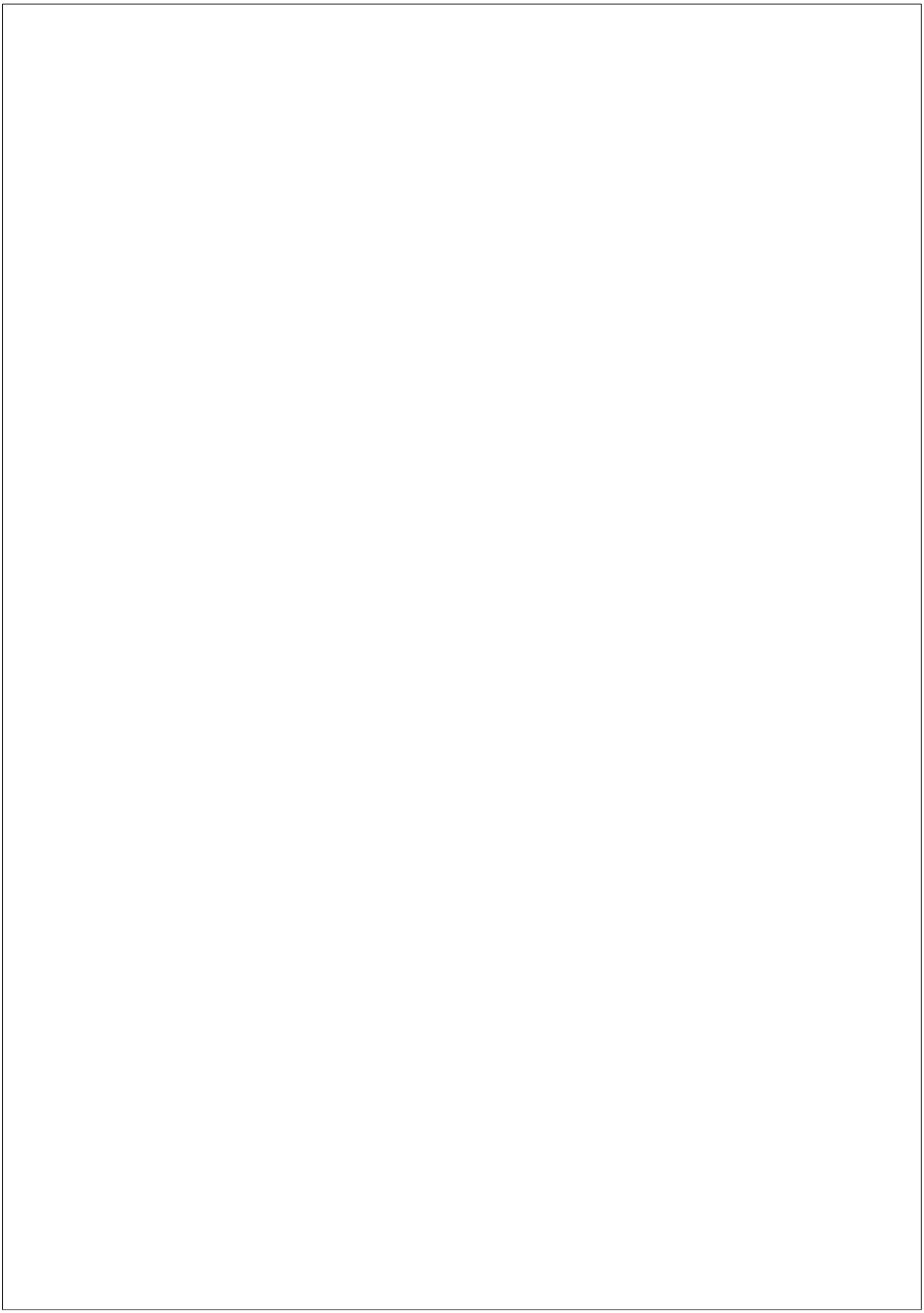
List of Figures

2.1 Example instances of TOWER domain 18
2.2 Planning program for placing all blocks of a single tower, on the table 21

3.1 Planning program for finding the minimum element in a list of integers of size n. 32
3.2 Derived predicates in the form of conjunctive queries for finding the minimum number in a list. 35
3.3 Planning program that encodes a noise-free classifier for the Michalski's train problem. 44

4.1 Invariant graphs (G_1, G_2 and G_3) in LOGISTICS. 55
4.2 Architecture overview 57
4.3 Example of search in the invariant graph G_1 in LOGISTICS domain. 68
4.4 Algorithm ordering preconditions of a except p. 70
4.5 Invariant graphs (G1, G2), where (uv), (xy), (yz) represent actions, while the nodes represent predicates. The preconditions of each action are: $prec(uv) = \{x\}$, $prec(xy) = \emptyset$, $prec(yz) = \{v\}$ 72

5.1 Invariant graphs (G1, G2 and G3) in the BLOCKS domain. 83



List of Tables

3.1 Program lines, slots and variables of the features, time (in seconds) elapsed while computing the solution, and plan length required to generate and verify the solution. . . . 46

4.1 The first five task expansions of the `solve` task generated from the running example in LOGISTICS. The colored tasks are those added by the decomposition of the underlined task in the previous step. 67

4.2 Results in the IPC-2000 and IPC-2002 domains, with the total number of instances of each domain shown in brackets. For each solver we report number of solved instances (#s), average time in seconds (t) and average number of backtracks in thousands (#b) respectively 75

5.1 Coverage of HTN-MAKER's experiment instances, scores marked with * are scores over IPC instances 88

5.2 Results in the IPC-2000 and IPC-2002 domains, with the total number of instances of each domain shown in brackets. For each solver we report number of solved instances (#s), average time in seconds (t) and average number of backtracks in thousands (#b) respectively 89

Part I

Background

Chapter 1

CLASSICAL PLANNING

In this chapter modeling languages for classical planning and the classical planning model are introduced. The domain-instance separation and different representations of planning problem are discussed, along with invariance and abstractions used in classical planning.

1.1 Modeling languages

One of the first modeling languages used for planning was STRIPS (Stanford Research Institute Problem Solver) (Fikes and Nilsson, 1971). Such a model is constructed using first-order logic formuale, where the literals are grounded and function-free. A state is represented by a set of true literals and all other literals are assumed to be false. The model consists of the initial and goal state. Transitions between states are encoded by a set of actions which are represented by a set of preconditions and post-conditions. The main limitation of STRIPS is the expressive power. After a number of languages attempted to address the shortcomings of STRIPS language, the Planning Domain Definition Language (PDDL) (McDermott et al., 1998), emerged as a standard modeling language used in plan-

ning. It allows for the domain to be modeled separately from any particular instance of the problem, and thus reuse the specified action model. The model specified in the domain describes all the actions in terms of preconditions and effects. On the other hand, each instance of a planning problem specifies the objects, the initial state and the goal condition.

Since the first version, PDDL imported older languages used in planning, like STRIPS and ADL. PDDL was first introduced at the International Planning Competition (IPC) in 1998, and has been evolving ever since. The PDDL versions, along with features they introduced, are listed here:

- PDDL 1.2, used in the first planning competition 1998.
- PDDL 2.1, introduced numeric fluents, plan metrics, durative continuous actions.
- PDDL 2.2, introduced derived predicates and timed literals.
- PDDL 3.0, introduces state-trajectory constraints and preferences.
- PDDL 3.1, introduced object fluents.

The work presented in this thesis is based on fragments of PDDL version 2.2 (Edelkamp and Hoffmann, 2004). Apart form PDDL versions listed here, there are a number of extensions and variants of the PDDL modeling language, for example: PDDL+ (Fox and Long, 2002) which introduces processes and events, Probabilistic PDDL (PPDDL) (Younes and Littman, 2004) which introduces probabilistic effects, Relational Dynamic influence Diagram Language (RDDL) (Sanner, 2010) which introduces partial observability, etc. These extensions are mainly addressing the issues related with modeling of the real-world problems, by adding new features. The enhanced expressiveness of such extensions enables modeling of the different kinds of problems and not necessarily in the scope of classical planning.

1.2 Planning domains and instances

In this section we introduce the concepts of planning domains and show how instances are induced, by binding the objects specified in the instances. To represent planning domains, we adapt a definition based on function symbols (Bäckström et al., 2014). We consider a fragment of PDDL modeling typed STRIPS planning domains, which include negative preconditions and goals. Planning domains are used to specify action schemata in PDDL style. The instantiated actions can only be deterministic, and the states on which they operate must be fully observable. Planning instances introduce particularities of a specific planning problem, by defining the initial state and the goal condition, along with specific objects. Given a planning domain and a planning instance, a planning problem is instantiated.

Given a set X, let X^n denote the set of vectors of length n whose elements are symbols in X. Given such a vector $x \in X^n$, let $x_k \in X$, $1 \leq k \leq n$, denote the k-th element of x.

We distinguish between typed and untyped function symbols. An untyped function symbol f has an associated arity $\alpha(f)$. In addition, a typed function symbol f has an associated type list $\beta(f) \in \mathcal{T}^{\alpha(f)}$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a set of types. Let F be a set of function symbols, and let $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$ be a set of objects, where Σ_i, $1 \leq i \leq n$, is a set of objects of type τ_i. We define $F_\Sigma = \{f[x] : f \in F, x \in \Sigma^{\alpha(f)}\}$ as the set of new objects obtained by applying each function symbol in F to each vector of objects in Σ of the appropriate arity. If f is typed, $f[x]$ has to satisfy the additional constraint $x_k \in \Sigma_{\beta_k(f)}$ for each k, $1 \leq k \leq \alpha(f)$, i.e. the type of each element in x is determined by the type list $\beta(f)$ of f.

Let f and g be two function symbols in F. An *argument map from f to g* is a function $\varphi : \Sigma^{\alpha(f)} \to \Sigma^{\alpha(g)}$ mapping arguments of f to arguments of g. An argument map φ allows us to map each object $f[x] \in F_\Sigma$ to an object $g[\varphi(x)] \in F_\Sigma$. In PDDL, argument maps have a restricted form:

each element in $\varphi(x)$ is either an element from x or a constant object in Σ independent of x. WLOG we assume that argument maps are well-defined for typed function symbols.

Definition 1. *A planning domain is a tuple* $\boldsymbol{d} = \langle \mathcal{T}, <, \mathcal{P}, \mathcal{A} \rangle$*, where:*

- $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ *is a set of types*
- $<$ *is an inheritance relation on types*
- $\mathcal{P}$ *is a set of typed function symbols called* predicates
- $\mathcal{A}$ *is a set of typed function symbols called* action schemata

Given a set of types $\mathcal{T}$ and the inheritance relation $<$, we denote a transitive closure over the inheritance relation as $<^*$. We say that τ_i inherits from τ iff $\tau_i <^* \tau$. Each action schema $a \in \mathcal{A}$ has a set of preconditions $\mathrm{pre}(a)$ and a set of positive and negative effects $\mathsf{eff}(a)$. Each element in these two sets is a pair (p, φ) consisting of a predicate $p \in \mathcal{P}$ and an argument map φ from a to p.

For example, consider a PDDL planning domain which has only one action, and the sole purpose of the domain is to be able to move blocks stacked in a tower to the table:

```
(define (domain tower-strips)
        (:requirements :strips :typing)
        (:types block)
        (:predicates (on ?x - block ?y - block)
                     (ontable ?x - block)
                     (clear ?x - block)
        )

        (:action mvToTable
                :parameters (?x - block ?y - block)
                :precondition (and (clear ?x) (on ?x ?y))
                :effect (and (ontable ?x) (clear ?y)
                             (not (on ?x ?y)))
        )
)
```

Types impose restrictions on the objects which can be mapped to the arguments of an action schema. In the example above, parameters $?x$ and $?y$ can only be objects of type `block`.

A planning instance specifies the initial state and the goal condition, along with the set of objects which are needed for defining those.

Definition 2. *Given a planning domain* $\boldsymbol{d} = \langle \mathcal{T}, <, \mathcal{P}, \mathcal{A} \rangle$*, a planning instance is a tuple* $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$*, where:*

- $\Omega = \Omega_1 \cup \ldots \cup \Omega_n$ *is a set of objects of each type in* $\mathcal{T}$
- *Object* $\omega \in \Omega$ *has type* $\tau \in \mathcal{T}$ *iff* $\omega \in \Omega_i$ *for some* $1 \leq i \leq n$ *and* τ_i *equals* τ *or inherits from* τ
- init *is an initial state*
- goal *is a goal condition*

An example of a PDDL instance which defines objects, initial state and goal condition is given here:

```
(define (problem tower-strips-3)
        (:domain tower-strips)
        (:objects A B C - block)
        (:init  (ontable A) (on B A)
                (on C B) (clear C)
        )
)
```

To induce a planning instance, predicates and action schemata defined in the domain $\boldsymbol{d}$ are combined with a set of objects defined in the planning instance $\boldsymbol{p}$, along with the initial state and the goal condition, specified in the planning instance $\boldsymbol{p}$. Action schemata are used to implicitly define actions. However, it is only by assignment of objects of the correct type in the action schema parameter list, that the grounded actions are generated. In the same manner predicates implicitly define fluents.

The induced planning instance $\boldsymbol{p}$, implicitly defines a set of fluents $\mathcal{P}_\Omega$ and a set of grounded actions $\mathcal{A}_\Omega$. A grounded action $a[x] \in \mathcal{A}_\Omega$, has a set of preconditions $\mathrm{pre}(a[x])$ and a set of positive and negative effects $\mathsf{eff}(a[x])$. Each element in these sets is a fluent $p[\varphi(x)] \in \mathcal{P}_\Omega$, where (p, φ) is the associated precondition or effect of the action a. The initial state $\mathrm{init} \in \mathcal{P}_\Omega$ and the goal condition $\mathrm{goal} \in \mathcal{P}_\Omega$, are both subsets of fluents. We often abuse notation by dropping the argument x of elements in $\mathcal{P}_\Omega$ and $\mathcal{A}_\Omega$.

In the TOWER example, the predicate $\texttt{on}[?\texttt{a}, ?\texttt{b}]$ for objects $\texttt{A}, \texttt{B}, \texttt{C}$ induces fluents such as $\texttt{on}[\texttt{B}, \texttt{A}], \texttt{on}[\texttt{C}, \texttt{B}]$. In the same example, actions such as $\texttt{mvToTable}[\texttt{B}, \texttt{A}]$ and $\texttt{mvToTable}[\texttt{C}, \texttt{B}]$ are induced.

1.3 Classical planning model

A classical planning frame consists of a set of fluents F and a set of actions A.

Definition 3. *A classical planning frame is a tuple $\Phi = \langle F, A \rangle$, where:*

- *F is a set of fluents*
- *A is a set of actions*

The planning frame is induced by a planning domain $\boldsymbol{d} = \langle \mathcal{T}, <, \mathcal{P}, \mathcal{A} \rangle$ and the set of objects Ω of a planning instance, therefore $F = \mathcal{P}_\Omega$ and $A = \mathcal{A}_\Omega$.

1.3.1 Classical planning problem

The problem of classical planning is one of finding a valid sequence of actions, which leads from an initial state to the goal condition.

Definition 4. *A planning problem is a tuple $P = \langle F, A, I, G \rangle$, where:*

- *$\langle F, A \rangle$ is a planning frame*
- *I is an initial state*
- *G is a goal condition*

A literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). Given L, let $\neg L = \{\neg l : l \in L\}$ be the complement of L.

As in the case of induced planning instances a *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then bounded and given by $2^{|F|}$. Given a subset $F' \subseteq F$ of fluents, let $s|_{F'}$ be the *projection* of s onto F', defined as $s|_{F'} = (s \cap F') \cup (s \cap \neg F')$, i.e. $s|_{F'}$ contains all literals on F' that are present in s. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s.

An action $a \in A$ is *applicable* in s if and only if $\mathrm{pre}(a) \subseteq s$, and the result of applying a in s is a new state $s \ltimes a = (s \setminus \neg\mathsf{eff}(a)) \cup \mathsf{eff}(a)$. A plan is then a sequence of applicable actions which leads from an initial state to the goal state.

Definition 5. *A plan for P is a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ such that a_i, $1 \leq i \leq n$, is applicable in $I \ltimes a_1 \ltimes \cdots \ltimes a_{i-1}$, and π solves P if it reaches the goal state, i.e. if $G \subseteq I \ltimes a_1 \ltimes \cdots \ltimes a_n$.*

Given a domain $\boldsymbol{d}$ and an induced set of planning instances $\boldsymbol{p}_s = \{\boldsymbol{p}_1 \ldots \boldsymbol{p}_T\}$, which share the same set of objects Ω, the equivalent set of planning problems is Ψ. The planning frame Φ is shared by a set of planning problems Ψ if and only if:

- The same planning domain $\boldsymbol{d}$ induced each of the planning instances

in $\boldsymbol{p}_s$

- The set of objects Ω is shared by the planning instances in $\boldsymbol{p}_s$

1.3.2 Classical planning with conditional effects

So far we have considered actions with preconditions, such that all the effects of an action are applied depending only on the preconditions of an action. Here we introduce conditional effects which allow for a specified subset of action effects to be applied if a certain condition holds, along with the action precondition.

As discussed before, we consider the fragment of classical planning with conditional effects that includes negative conditions and goals. Under this formalism, a *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions with conditional effects.

Each action $a \in A$ has a set of literals $\mathrm{pre}(a)$ called the *precondition* and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals C (the condition) and E (the effect).

An action $a \in A$ is applicable in state s if and only if $\mathrm{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is a new state $s \ltimes a = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

The only difference in the definition of a planning problem and a solution plan, with conditional effects, is that the effects of actions are conditional on the state. In fact, the effects of actions without the conditional effects are a special case of conditional effects. We can create a conditional effect: $\emptyset \rightarrow \mathtt{eff}(a)$, so that the effect of action a will always occur.

For example, using the conditional effects, `mvToTable` can be written, without parameters, as:

```
(:action mvToTable
        :parameters ()
        :precondition (and)
        :effect (forall (?x ?y - block)
           (and (when (and (on ?x ?y) (clear ?x))
           (and (not (on ?x ?y))
                     (ontable ?x) (clear ?y)))
                )
)
```

1.3.3 Multivalued representations

A set of fluents of a planning problem defined above can be seen as a set of first-order variables. However, often a more compact encoding of a planning problem is beneficial for the solvers. These representations can represent the same planning problem using a set of variables which are usually defined on a certain finite domain. We introduce a multivalued representation which we later use to define causal and domain transition graphs, in the following sections.

Our definition of a multivalued representation of a planning problem is based on the definition by Helmert (2004).

Definition 6. *A multivalued representation of a planning problem is a tuple $\langle \mathcal{V}, A, I, G \rangle$, where:*

- *$\mathcal{V}$ is a list of state variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, with an associated finite domain $\mathcal{D}_v$. A partial variable assignment over $\mathcal{V}$ is a function s on some subset of $\mathcal{V}$ such that $s(v) \in \mathcal{D}_v$ wherever $s(v)$ is defined. If $s(v)$ is defined for all $v \in \mathcal{V}$, s is called a state.*

- *A is a set of actions, where each action is a pair $\langle \texttt{pre}, \texttt{eff} \rangle$, of partial variable assignments, preconditions and effects respectively*
- *I is an initial state*
- *G is a goal state defined as a partial variable assignment*

An action $a \in A$ is applicable in a state if the value of the variables in its preconditions are defined and equal to the value of the variables in the state. Applying the action changes the value of some state variable to $\mathsf{eff}(v)$ if $\mathsf{eff}(v)$ is defined.

Previous definition of the planning problem, can be seen as a multivalued representation, where variables can only take two values, i.e. for all variables, $D_v = \{\texttt{true}, \texttt{false}\}$.

1.4 Invariants

An invariant is a property, which is true in all reachable states of a given planning problem. Many invariants are not interesting from a planning perspective, as they cannot be exploited in finding a solution. Other invariants, while useful, can also be hard to compute (some can be PSPACE-hard). However, *mutual exclusion* (mutex) invariants are often exploited in planning. Planners such as *Fast Downward* use them to construct a multivalued planning problem representation from a propositional planning problem described before.

A mutex invariant for a planning problem is defined by a set of fluents which can not be true at the same time i.e. in the same state. Invariance is usually proven inductively, starting with some initial guess, called *invariant candidate*. First the initial state is checked, and then if the condition holds in that state, by analyzing the actions, it can be shown to hold in an arbitrary reachable state, and all successor states. The condition is then proven for all reachable states and thus is an invariant property.

Fast Downward includes an algorithm for synthesizing the invariants (Helmert, 2009) that works directly on a PDDL domain part described in the section before. The resulting mutexes cannot be validated, without a planning instance. While other algorithms for synthesizing invariants exist, most of them depend on STRIPS representation of a planning problem, or do not support PDDL 2.2 domains, therefore in this work we rely on Fast Downward's algorithm for finding mutex invariants.

The synthesis of the invariants consists of an initial guided guess, validation, and refining of the failed candidates. Given that this is a search problem, breath first search with a closed list is used, so that the search is guaranteed to terminate. The initial invariant candidates are constructed from fluents that appear as effects of actions, as opposed to static fluents whose value remains unchanged, which are then checked. If an invariant candidate is confirmed it is not considered further by the search algorithm. If the invariant candidate is not confirmed in the check phase, the algorithm attempts to classify it and refine it before it attempts a new check on this invariant candidate. The refinement is done by adding new atoms to the failed invariant candidate.

1.5 Abstractions in planning

In order to gain insight into the original planning problem, planners usually construct abstractions over it. The properties of such derived structures can then be exploited to limit the search space, extract heuristics or infer if the problem is solvable, etc. Many of these structures are graphs, constructed from the original planning problem. In this work, we will use causal and domain transition graphs. Our definitions of causal and domain transition graph are based on the definitions introduced by Helmert (2004).

In a multivalued representation of a planning problem, the causal graph shows dependencies of the variables, based on preconditions and effects

of the actions. The nodes of such graph are variables, while the edges are induced by the actions. Both the causal and the domain transition graph, are not only more compact under multivalued representation, but can also have different properties. It can happen that a causal graph is cyclic in STRIPS problem representation, but not in a multivalued representation. It is widely believed that problems with acyclic causal graphs are easier to solve, however this is not true in the worst case (Jonsson et al., 2013).

Definition 7. *Given a multivalued representation of a planning problem* $\langle \mathcal{V}, A, I, G \rangle$*, a causal graph is a digraph* $(\mathcal{V}, E)$ *containing an arc* $(\mathtt{u}, \mathtt{v})$ *iff* $(\mathtt{u} \neq \mathtt{v})$ *and there exists an action* $\langle \mathtt{pre}, \mathtt{eff} \rangle \in A$ *, such that* $\mathtt{eff}(\mathtt{v})$ *is defined and either* $\mathtt{pre}(\mathtt{u})$ *or* $\mathtt{eff}(\mathtt{u})$ *are defined.*

A domain transition graph (DTG) is constructed to represent the dependencies between the values of the variables in a multivalued representation. A DTG can be thought of as a graphical representation of a variable in a multivalued representation.

Definition 8. *Given a multivalued representation of a planning problem* $\langle \mathcal{V}, A, I, G \rangle$*, where* $v \in \mathcal{V}$*, a domain transition graph is a labeled digraph* G_v*, with vertex set* $\mathcal{D}_v$ *which contains an arc* (d, d') *iff there is an action* $\langle \mathtt{pre}, \mathtt{eff} \rangle$ *where* $\mathtt{pre}(\mathtt{v}) = d$ *or* $\mathtt{pre}(\mathtt{v})$ *is undefined, and* $\mathtt{eff}(\mathtt{v}) = d'$*. The arc is labeled by* $\mathtt{pre}| \ (\mathcal{V} \setminus \{v\})$*. For each arch* (d, d') *with label* L *we say that there is a transition of* v *from* d *to* d' *under the condition* L*.*

Chapter 2

GENERALIZED PLANNING

In this chapter the problem of generalized planning is introduced. Planning programs and hierarchical task networks are presented as representations of generalized plans.

2.1 Introduction

In classical planning a plan is a valid sequence of actions which reaches the goal from a given initial state, for a particular planning problem. As discussed in section 1.3, such plans are not expected to be valid for any other planning problem. In a broad sense, generalized planning is a problem of finding plans which are valid for more than one planning problem. Those planning problems are expected to share a certain structure, and definitions of generalized planning can vary depending on what commonalities are assumed. Classical planners can be seen as singleton generalized planners, since they are able to generate a plan for every individual planning instance.

Our definition of the generalized planning task is based on that of Hu and De Giacomo (Hu and De Giacomo, 2011), who define a generalized planning problem as a finite set of multiple individual planning problems $\Psi = \{P_1, \ldots, P_T\}$ that share the same observations and actions. Although actions are shared, in their formalism each action can have a different interpretation, depending on a state, due to conditional effects.

In this work we restrict the above definition for generalized planning in two ways:

1. States are fully observable, so observations are equivalent to the state variables.
2. Each action has the same (conditional) effects, in each individual problem.

Definition 9. *A generalized planning problem Ψ is a set of individual classical planning problems $P_1 = \langle F, A, I_1, G_1\rangle, \ldots, P_T = \langle F, A, I_T, G_T\rangle$, that share the same planning frame $\Phi = \langle F, A\rangle$, and differ only in the particular initial state and goals.*

Consequently a generalized planning problem can also be defined as a set of planning instances $\Psi = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_T\}$, which share the same domain $\boldsymbol{d}$ and set of objects Ω, inducing a planning frame $\langle F, A\rangle$.

Definition 10. *A solution π to a generalized planning problem Ψ is a generalized plan that solves each individual problem P_t, $1 \leq t \leq T$ in a set of planning problems Ψ.*

This definition of the generalized planning task is related to previous works on planning and learning that extract and reuse general knowledge from different tasks of the same domain (Fern et al., 2011; Jiménez et al., 2012). The constraint imposed in this work is stronger, as by sharing the same domain the set of instances also share the same fluents, so all individual planning tasks have the same state space.

Representations of a generalized plan can take many different forms. In

the literature the form of a generalized plan ranges from *DS-planners* (Winner and Veloso, 2003) and *generalized polices* (Martín and Geffner, 2004) to finite state machines (FSMs) (Bonet et al., 2010). Each representation has a different syntax and semantics, but they all allow non-sequential execution flow to solve planning instances with different initial states and goals. In this work we use planning programs as expressions of generalized plans and hierarchical task networks (HTNs) as a different form of generalized plan that does not explicitly represent a unique solution.

Once derived a generalized plan allows for solving multiple planning instances. However, the complexities of instantiation and means for computing such plans can vary depending on the representation. According to Srivastava (2011) several criteria are relevant for evaluating a generalized plan:

- Complexity of checking applicability
- Complexity of plan instantiation
- Domain coverage
- Complexity of computing the generalized plan
- Quality of the instantiated plan

As an extreme example, a classical planner can be considered as a generalized plan, if the complexity of the instantiation of such a plan is acceptable. A classical planner forms an independent search problem, regardless of whether it has already solved the same, or a similar instance of the planning problem. Given a generalized plan, depending on the representation, checking quality, applicability, and determining the exact domain coverage of such a plan, can still be hard. A generalized plan can be incomplete in the sense that some search still has to be performed. However, the idea of generalized planning is to reduce the search space by encoding knowledge, about the whole planning domain.

Approaches to construction of generalized plans can be divided into *in-*

ductive and *deductive* (or *generative*). The inductive approaches attempt to generalize from a given set of examples. In case of classical planning, those examples are solutions to a training set of planning problems. The deductive approach to generalized planning derives the plan directly from the model, which is represented as the planning domain $\boldsymbol{d}$, introduced before. The classical planning domain models the actions for a certain world and by deriving general rules one can assemble a strategy that solves all the instances in a given planning domain.

According to our definition of the generalized planning task the particular case where $|\Psi| = 1$ corresponds to classical planning with conditional effects. The case for which all the individual problems in Ψ share the same goals, $G_1 = G_2 = \cdots = G_{T-1} = G_T$, corresponds to conformant planning (Palacios and Geffner, 2009). Nevertheless the form of the solutions is different. While solutions in classical planning or in conformant planning are defined as sequences of actions, generalized plans relax this assumption and exploit a more expressive solution representation with non-sequential execution that can achieve more compact solutions.

Consider the example TOWER domain from the previous chapter. We can define an instance which describes a single tower of any size, or an instance with multiple towers as shown in Figure 2.1. A generalized plan would be able to solve any such instance.

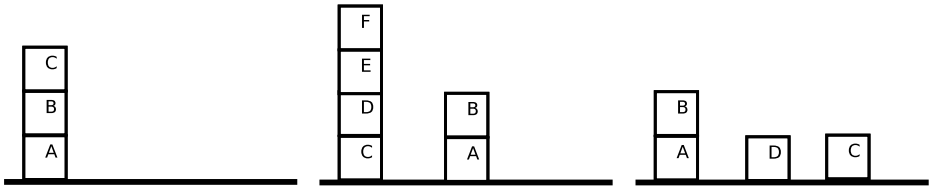


Figure 2.1: Example instances of TOWER domain

Such a generalized plan can be expressed as a policy:

1. If there is a clear block on top of another block, move that block to the table.

2. While not all blocks are on table, repeat 1.

In the following sections we will explore how such a policy can be expressed.

2.2 Planning Programs

This section introduces the basic version of the *planning program* formalism (Jiménez and Jonsson, 2015). In its simplest form, a planning program is a sequence of planning actions enhanced with *goto instructions*, i.e. conditional constructs for jumping to arbitrary locations of the program, allowing for non-sequential plan execution with branching and loops. In this section we define basic planning programs.

2.2.1 Basic Planning Programs

Given a STRIPS frame $\Phi = \langle F, A \rangle$, a basic planning program is a sequence of instructions $\Pi = \langle w_0, \ldots, w_n \rangle$. Each instruction w_i, $0 \leq i \leq n$, is associated with a *program line* i and is drawn from a set of instructions $\mathcal{I}$ defined as

$$\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\texttt{end}\}, \quad \mathcal{I}_{go} = \{\texttt{goto(i',!f)} : 0 \leq i' \leq n, f \in F\}.$$

In other words, each instruction is either a planning action $a \in A$, a goto instruction `goto(i',!f)` or a termination instruction `end`. A termination instruction acts as an explicit marker that program execution should end, similar to a `return` statement in programming. We explicitly require that the last instruction w_n should equal `end`, and since this instruction is fixed, we say that Π has $|\Pi| = n$ program lines, even though Π in fact contains $n + 1$ instructions.

The execution model for a planning program Π consists of a *program state* (s, i), i.e. a pair of a planning state $s \subseteq F$ and a program counter whose

value is the current program line i, $0 \leq i \leq n$. Given a program state (s, i), the execution of instruction w_i on line i is defined as follows:

- If $w_i \in A$, the new program state is $(s', i+1)$, where $s' = s \ltimes w_i$ is the result of applying action w_i in planning state s, and the program counter is simply incremented.
- If $w_i = \texttt{goto(i', !f)}$, the new program state is $(s, i{+}1)$ if $f \in s$, and (s, i') otherwise. We adopt the convention of jumping to line i' whenever f is *false* in s. Note that the planning state s remains unchanged.
- If $w_i = \texttt{end}$, execution terminates.

To execute a planning program Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I, 0)$, i.e. the initial state of P and program line 0. We say that Π *solves* P if and only if execution terminates and the goal condition holds in the resulting program state (s, i), i.e. $G \subseteq s \wedge w_i = \texttt{end}$. Executing Π on P can fail for three reasons:

1. Execution terminates in program state (s, i) but the goal condition does not hold, i.e. $G \not\subseteq s \wedge w_i = \texttt{end}$.
2. When executing an action $w_i \in A$ in program state (s, i), the precondition of w_i does not hold, i.e. $\mathrm{pre}(w_i) \not\subseteq s$.
3. Execution enters an infinite loop that never reaches an `end` instruction.

This execution model is *deterministic* and hence a basic planning program can be viewed as a form of compact reactive plan for the family of planning problems defined by the STRIPS frame $\Phi = \langle F, A \rangle$.

If we consider the TOWER example, to construct a planning program we would need a condition in the `goto` instruction that depends on a single fluent. This means that in the basic version of planning programs we cannot check if the goal condition is satisfied, without explicitly defining the bottom of the tower. We can define the bottom of the tower to be block `z`, in this case the stop condition would be when block `z` becomes clear.

Of course this solution would work for single tower instances only. The planning program for any instance with a single tower, is a program in Figure 2.2.

```
0. mvToTable
1. goto(0,!(clear(z)))
2. end
```

Figure 2.2: Planning program for placing all blocks of a single tower, on the table

In order to represent a plan which can put the blocks, of any number of towers, on the table, we would need to enhance the expressiveness of the basic planning programs. One possible extension, which is further explored in Chapter 3, is the use of the derived predicates. The basic version of planning programs is sound and complete. The proofs have been submitted to a journal and currently are under revision.

2.3 Hierarchies in Planning

In this section we define the domain and instance of hierarchical task networks (HTN). HTNs are models, which enable capturing knowledge about a certain planning domain, through hierarchical definitions of tasks. Hierarchies in a domain can help reduce the search space, in any single instance of the problem. In this section we also discuss modeling languages for HTNs and the notation used in this work.

2.3.1 Hierarchical Task Networks

Hierarchical Task Networks models enable defining compound parameterized tasks which can reduce the search complexity if the adequate constraints can be identified during the modeling. The hierarchical structure

enables the modeler to encode domain-specific knowledge. The expressiveness of HTNs can help to impose the constraints, which can in turn lead to a reduction in the search complexity. The hierarchy ideally imposes some constraints on how tasks can be decomposed. The more constrained the task network, the less search has to be performed in order to achieve a certain task.

HTNs are more expressive than STRIPS (Erol et al., 1994), which along with the ability to construct parametric compound tasks allows for capturing domain-specific knowledge. This knowledge in turn allows even state-of-the-art HTN planners like SHOP2 (Nau et al., 2003) to rely on the hierarchies provided in the model by applying nothing more than blind search. In contrast most STRIPS planners use heuristic search to find plans. However in practice HTN planners are the ones that have been more extensively and applied in a variety of problems: military planning (Munoz-Avila et al., 1999), Web service composition (Wu et al., 2003), unmanned air vehicle control (Miller et al., 2004), strategic game playing (van der Sterren, 2009; Menif et al., 2013), personalized patient care (Sánchez-Garzón et al., 2013) and business process management (González-Ferrer et al., 2013), to name a few.

Apart from STRIPS being less expressive, another reason for the popularity of HTNs is that classical planners rely on domain-independent heuristics, which have to be calculated for each instance of the domain, which leads to scalability issues. While in STRIPS planning it is enough to specify the basic actions, to create an HTN as an efficient representation, the modeler has to identify useful hierarchies over those basic actions and in doing so ideally impose some constraints.

Our HTN definition is inspired by Geier and Bercher (Geier and Bercher, 2011). However, just as for classical planning, we separate the definition into a domain part and an instance part. In this work, we also impose additional restrictions: a task network can contain at most one copy of each task, and task decomposition is limited to *progression*, always decomposing tasks with no predecessor.

Definition 11. *An HTN domain is a tuple* $\boldsymbol{h} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M} \rangle$ *consisting of four sets of untyped function symbols. Specifically:*

- $\mathcal{P}$ *is the set of* predicates
- $\mathcal{A}$ *is the set of* action schemata *(i.e. primitive tasks)*
- $\mathcal{C}$ *is the set of* compound tasks
- $\mathcal{M}$ *is the set of* decomposition methods.

Predicates and actions are defined as for the planning domains in previous chapter but, unlike PDDL domains, HTN domains are untyped. Each method $m \in \mathcal{M}$ has an associated tuple $\langle c, tn_m, \mathrm{pre}(m) \rangle$ where $c \in \mathcal{C}$ is a compound task with the same arity as m, tn_m is a *task network* and $\mathrm{pre}(m)$ is a set of preconditions, defined as for actions. The task network $tn_m = (T, \prec)$ consists of a set T of pairs (t, φ), where $t \in \mathcal{A} \cup \mathcal{C}$ is a task and φ is an argument map from m to t, and a partial order $\prec$ on the tasks in T.

Definition 12. *Given an HTN domain* $\boldsymbol{h}$*, an HTN instance is a tuple* $\boldsymbol{s} = \langle \Omega, \mathrm{init}, tn_I \rangle$*, where:*

- Ω *is a set of objects*
- init *is an initial state.*
- tn_I *is an initial task network*

The instance $\boldsymbol{s}$ induces sets $\mathcal{P}_\Omega$ and $\mathcal{A}_\Omega$ of fluents and grounded actions, and sets $\mathcal{C}_\Omega$ and $\mathcal{M}_\Omega$ of grounded compound tasks and grounded methods, respectively. A grounded method $m[x] \in \mathcal{M}_\Omega$ has associated tuple $\langle c[x], tn_m[x], \mathrm{pre}(m[x]) \rangle$, where $c[x]$ is a grounded compound task and the precondition $\mathrm{pre}(m[x])$ is derived as for grounded actions. The grounded task network $tn_m[x] = (T_x, \prec)$ is defined by $T_x = \{t[\varphi(x)] : (t, \varphi) \in T\}$. In this work, we assume that the initial grounded task network $tn_I = (\{t_I\}, \emptyset)$ contains a single grounded compound task $t_I \in \mathcal{C}_\Omega$.

An HTN state (s, tn) consists of a state $s \subseteq \mathcal{P}_\Omega$ on fluents and a grounded task network tn. We use $(s, tn) \rightarrow_D (s', tn')$ to denote that an HTN state decomposes into another HTN state, where $tn = \langle T_x, \prec \rangle$ and $tn' = \langle T_y, \prec' \rangle$. A valid *progression decomposition* consists in choosing a grounded task $t \in T_x$ such that $t' \not\prec t$ for each $t' \in T_x$, and applying one of the following rules:

1. If t is primitive, the decomposition is applicable if $\mathrm{pre}(t) \subseteq s$, and the resulting HTN state is given by $s' = s \ltimes t$, $T_y = T_x \setminus \{t\}$ and $\prec' = \{(t_1, t_2) \in \prec \mid (t_1, t_2) \in T_y\}$.
2. If t is compound, a grounded method $m = \langle t, tn, \mathrm{pre}(m) \rangle$ with $tn = (T_m, \prec_m)$ is applicable if $\mathrm{pre}(m) \subseteq s$, and the resulting HTN state is given by $s' = s$, $T_y = T_x \setminus \{t\} \cup T_m$ and

$$\begin{aligned} \prec' = & \{(t_1, t_2) \in \prec \mid (t_1, t_2) \in T_y\} \\ & \cup \{(t', t_1) \in T_m \times T_y \mid (t, t_1) \in \prec\} \cup \prec_m . \end{aligned}$$

The first rule removes a grounded primitive task t from tn and applies the effects of t to the current state, while the second rule uses a grounded method m to replace a grounded compound task t with tn_m while leaving the state unchanged. If there is a finite sequence of decompositions from (s_1, tn_1) to (s_n, tn_n) we write $(s_1, tn_1) \rightarrow_D^* (s_n, tn_n)$. An HTN instance $\boldsymbol{s}$ is solvable if and only if $(\mathrm{init}, tn_I) \rightarrow_D^* (s_n, \langle \emptyset, \emptyset \rangle)$ for some state s_n, i.e. the initial HTN state (init, tn_I) is decomposed into an empty task network. Let π be the sequence of grounded actions extracted during such a decomposition; π corresponds to a *plan* that results from solving $\boldsymbol{s}$.

2.3.2 SHOP2 modeling language

There are many modeling languages used to encode HTNs. These languages often differ depending on the HTN solver. The HTN solvers provide different capabilities and are thus reliant on the expressiveness of the

modeling language. In this work we use the JSHOP2 HTN planner, which uses SHOP2 syntax with minor modifications. The syntax of the SHOP2 modeling language allows encoding of a highly expressive domain representations, while incorporating many features from PDDL. In this section we introduce only the subset of features of the SHOP language which are relevant for this work.

The basis for SHOP2 language are the notions of *operator* and *method*. The methods can decompose a single compound task into a list of tasks. This list can be composed of compound and primitive tasks. The same task is allowed to be added to the decomposition list, providing means for recursion. The methods can only decompose a certain task if the precondition list of a particular method holds in the current state. The operators are similar to action schemata in classical planning introduced in the previous chapter, with the difference that the positive and negative effects are given as an add and delete lists, respectively. Therefore operators are used to achieve the primitive tasks, while the methods decompose compound tasks.

The operators are given as: (: `operator` `h` `P` `D` `A`) Where:

- h - the operator name and an untyped parameter list
- P - precondition list
- D - delete list
- A - add list

The methods are given as: (: `method` `h` `P` `T`) Where:

- h - method name and an untyped parameter list
- P - precondition list
- T - task list, to which the method decomposes

The operators directly correspond to the notion of action schemata defined in a PDDL domain. An operator in the example of TOWER domain introduced in the previous chapter, is equivalent to the action schema, from the

original planning domain: (`mvToTable` ?`x` ?`y`). Just as the original action schema, this operator removes a clear block from a tower of blocks and places it on the table.

In the SHOP2 syntax this operator can be written as:

```
( :operator (mvToTable ?x ?y)
        ( (block ?x) (block ?y)
          (clear ?x) (on ?x ?y)
        )
        ( (on ?x ?y) )
        ( (clear ?y) (on-table ?x) )
)
```

The domains modeled in SHOP2 are not typed, so predicates that denote types should be used throughout the HTN to impose restrictions on input parameter types. In the case of (`mvToTable` ?`x` ?`y`) operator, note the type predicates for blocks, which need to be explicitly encoded in the preconditions.

In SHOP2 a decomposition method consists of a list of preconditions and a task list to which it decomposes. The decomposition list may contain both compound and primitive tasks. To distinguish the primitive tasks we ad an exclamation mark in front of the name.

The example of a method that decomposes an `achieveOnTable` compound task, is given as:

```
( :method (achieveOnTable ?x)
        ( (block ?x) (block ?y)
          (not (on-table ?x))
        )
        ( (!mvToTable ?x ?y)
          (achieveOnTable ?y)
        )
)
```

This method will decompose the `achieveOnTable` task recursively. The HTN planner will non-deterministically select an object to bind the free parameter ?y, the only condition is that bound object is a block. In case that ?y is not a block, placed below ?x, the decomposition will fail, due to preconditions of (!`mvToTable` ?x ?y).

In SHOP2 the order in which the methods are defined in the HTN domain is used as a tie-breaker, among all the methods that decompose the same task (i.e. defined with the same name). We assume that the tasks are totally ordered unless specified explicitly. In SHOP2 syntax the task list is only considered unordered if preceded by `:unordered` keyword. In case the task list is unordered, all possible orderings of the tasks will be explored, until the task list is fully decomposed.

Recursion assumes that there is a well-defined base case that is eventually reached during progression. The example of a base case for the recursion is a method:

```
( :method (achieveOnTable ?x)
         ( (block ?x)
           (clear ?x) (on-table ?x)
         )
         ( )
)
```

This method will stop the recursion if the parameter block, is already on the table and clear. The first version of the method will fail to decompose due to the precondition (`not` (`on-table` ?x)), and the second version of the method decomposes to an empty list. The base case method, could have also been defined to decompose to an operator, which would check if the block in already on the table.

Therefore, the decomposition methods of the `achieveOnTable` task, encode a strategy for placing a single tower of blocks on the table. The initial task network, would then consist of a single grounded compound task `achieveOnTable`, with the only clear block of the tower as a param-

eter. To model a problem with more than one tower of blocks, we could create a different compound task, with assigned methods which would decompose it to the `achieveOnTable` task, for each tower.

Part II

High-Level state features

Chapter 3

PLANNING PROGRAMS WITH HIGH-LEVEL STATE FEATURES

In this chapter we introduce an extension of planning programs with high-level state features in form of conjunctive queries. In addition we present a novel approach to generating high-level state features in a tight coupling with computation of a planning program. We show that the extension of planning programs remains sound and complete. Furthermore, we show that planning programs with high-level state features can model a noise-free classifier.

3.1 Introduction

A generalized plan is a single solution valid for a set of planning problems. Generalized plans are usually built with branching and repetition constructs which allow them to solve arbitrarily large problems, and problems with partial observability and non-deterministic actions (Bonet et al.,

2010; Hu and Levesque, 2011; Srivastava et al., 2011; Hu and De Giacomo, 2013). For many problems, generalized plans can only be efficiently computed if branching (and/or repetition) is done according to key features that allow high-level reasoning and help to accurately distinguish between states.

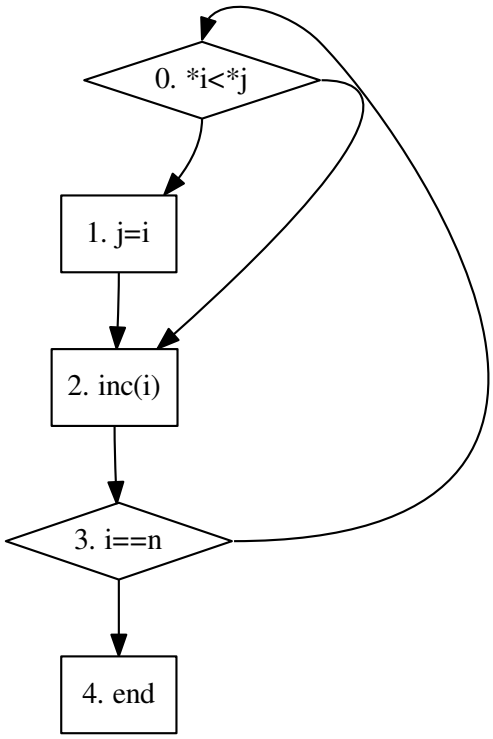


Figure 3.1: Planning program for finding the minimum element in a list of integers of size n.

To illustrate this, consider the problem of finding the minimum element in the following list of five integers: $(2, 5, 3, 1, 4)$. A classical plan for this problem is the 4-action sequence $\langle inc(i), inc(i), inc(i), j = i\rangle$ where i and j are list iterators that initially point to the first position in the list, $inc(i)$ increments iterator i, and $j = i$ assigns iterator i to j. The goal is for iterator j to point to the minimum element in the list. The plan fails to

generalize since it is no longer valid if the order of the integers in the list is changed or if integers are added to (or deleted from) the list.

A generalized plan can solve this problem for different list orders and sizes. In this work we focus on generalized plans in the form of *planning programs* (Jiménez and Jonsson, 2015; Segovia-Aguas et al., 2016). Figure 3.1 shows the planning program for finding the minimum element in a list of integers of any size. Instructions on lines 0 and 3, represented with diamonds, are conditional goto instructions that, respectively, jump to line 2 when $*i \not< *j$ and to line 0 when $i \neq n$. The outgoing bottom branch of the diamond indicates that the condition holds and the right branch that it does not. Instructions on lines 1 and 2 are sequential instructions and are represented with boxes. Finally, `end` marks the program termination.

A natural representation of this problem is to use predicates that represent the values of pointers. In this program, conditions $*i < *j$ and $i == n$ are high-level state features necessary to compactly represent a solution that generalizes. These features abstract different list contents and sizes as well as different values for iterators i and j. Specifically, condition $*i < *j$ abstracts the set of states where the element pointed to by i is smaller than that pointed to by j. Likewise, $i == n$ abstracts the set of states where iterator i reaches the end of the list, no matter the list size.

In generalized planning problems, high-level state features are traditionally hand-coded which requires significant human expertise. The contribution of this work is to automatically generate the high-level features required to solve a given generalized planning problem. We do so updating the notion of *planning programs* and extending a previous compilation of generalized planning into classical planning (Jiménez and Jonsson, 2015; Segovia-Aguas et al., 2016) to tightly integrate the computation of generalized plans with the computation of features in the form of conjunctive queries.

Another contribution of this work is bringing a new landscape of challeng-

ing benchmarks to classical planning given that our extended compilation naturally models classification tasks as classical planning problems. This allows us to solve problems that integrate planning and classification using an off-the-shelf classical planner.

3.2 Generating High-Level State Features

This section defines high-level state features as conjunctive queries and extends planning programs (Jiménez and Jonsson, 2015; Segovia-Aguas et al., 2016) such that *conditional goto instructions* jump according to the value of a conjunctive query. The section also extends the compilation from generalized planning into classical planning to generate planning programs with conjunctive queries using a classical planner.

3.2.1 High-Level State Features

The notion of a high-level state feature is very general and has been used in different areas of AI and for different purposes. If we restrict ourselves to planning, a high-level state feature can broadly be viewed as a state abstraction to compactly represent planning problems and/or solutions to planning problems. Diverse formalisms have been used to represent high-level state features in planning ranging from first order clauses (Veloso et al., 1995) to description logic formulae (Martín and Geffner, 2004), LTL formulae (Cresswell and Coddington, 2004), PDDL derived predicates (Hoffmann and Edelkamp, 2005) and, more recently, observation formulae (Bonet et al., 2010).

Given a planning domain $\boldsymbol{d} = \langle \mathcal{T}, <, \mathcal{P}, \mathcal{A} \rangle$, and a set of objects Ω defined in a planning instance, states are represented in terms of fluents F which are instantiated from a set of predicates $\mathcal{P}$ We consider high-level state features that are arbitrary formulae over the predicates in $\mathcal{P}$. A high-level state feature is also known as a *derived predicate* if it produces a new

predicate whose truth value is determined by the corresponding formula. Derived predicates have proven useful for concisely representing planning problems with complex conditions and effects (Thiébaux et al., 2005) and for more efficiently solving optimal planning problems (Ivankovic and Haslum, 2015).

$$\begin{aligned} \mathsf{equal}(a, b) \leftarrow\ & \exists x_1.\mathsf{assign}(a, x_1), \\ & \mathsf{assign}(b, x_1). \end{aligned}$$

$$\begin{aligned} \mathsf{lessthan\text{-}pointers}(a, b) \leftarrow\ & \exists x_1, x_2.\mathsf{points\text{-}to}(a, x_1), \\ & \mathsf{points\text{-}to}(b, x_2), \\ & \mathsf{lessthan}(x_1, x_2). \end{aligned}$$

Figure 3.2: Derived predicates in the form of conjunctive queries for finding the minimum number in a list.

In this work we restrict ourselves to formulae in the form of *conjunctive queries* from database theory (Chandra and Merlin, 1977). Conjunctive queries are a fragment of first-order logic in which formulae are constructed from atoms using conjunction and existential quantification (disallowing all other logical symbols). A conjunctive query can be written as

$$\varphi = (x_1, \ldots, x_k).\exists x_{k+1}, \ldots, x_m.\phi_1 \wedge \cdots \wedge \phi_q,$$

where $x_1, \ldots, x_k$ are *free variables*, $x_{k+1}, \ldots, x_m$ are *bound variables*, and $\phi_1, \ldots, \phi_q$ are *atoms*.

Figure 3.2 shows two derived predicates, in the form of conjunctive queries, that correspond to features $a == b$ and $*a < *b$. In both predicates, a and b act as free variables while x_1 and x_2 are bound variables. The first derived predicate models whether two given iterators point to the same memory address, while the second models whether the value pointed to by an iterator is less than the value pointed to by another iterator. The

example assumes that predicate assign models the assignment of a memory address to a pointer variable, points-to models the content of the associated memory location, and lessthan models that a value is less than another.

Given a planning domain ***d*** and a set of object Ω, we introduce a set X of m query variables, $X = \{x_1, \ldots, x_m\}$, each with domain Ω. Each atom $p(v)$ in the derived predicate consists of a predicate $p \in \mathcal{P}$ and a tuple $v \in X^{ar(p)}$ that assigns variables in X to arguments of p. In addition, we make the following assumptions regarding predicates and objects:

1. We partition Ω into a set of *variable objects* Ω_v (not to be confused with the query variables) and a set of remaining objects Ω_o.

2. For each predicate $p \in \mathcal{P}$ we designate at most one argument as a *variable argument* to be filled by a variable object (WLOG the variable argument always goes first). In the example, the first argument of predicates assign and points-to is a variable argument. For predicates without variable arguments, such as lessthan, we simply remove the variable argument and associated variable object.

3. In a conjunctive query, free variables can only be assigned to variable arguments of predicates and have domain Ω_v, while bound variables can only be assigned to non-variable arguments and have domain Ω_o.

Note that we could get rid of variable objects and variable arguments by redefining predicates (e.g. assign could become assign-i, assign-j and assign-n). As a consequence, all variables of a conjunctive query would be bound. However, variable objects offer flexibility, by allowing the variable objects to vary, and by generating derived predicates that are valid for a range of variable objects. For instance, the conditions $i == n$ and $*i < *j$ in the program of Figure 3.1 are generated by assigning objects to a and b accordingly (this requires i, j and n to be *variable objects* of the planning problem).

3.2.2 Planning Programs with Conjunctive Queries

To incorporate conjunctive queries into planning programs we simply replace the fluent f of conditional goto instructions $goto(i', !f)$ with a conjunctive query φ. The execution of a planning program with conjunctive queries proceeds as explained in Chapter 2, except when the instruction associated with the current program counter is a conditional goto instruction $goto(i', !\varphi)$. In that case, the program counter pc is set to i' if φ does not unify with the current state s, else pc is incremented.

Let $\varphi = (x_1, \ldots, x_k).\exists x_{k+1}, \ldots, x_m.\phi_1 \wedge \cdots \wedge \phi_q$ be a conjunctive query, and let $u \in \Omega_v^k$ be an assignment of variable objects to the free variables $x_1, \ldots, x_k$. We describe a strategy for unifying φ with the current state s. The idea is to maintain a subset $\Phi \subseteq \Omega_o^{m-k}$ of possible joint assignments of objects to the bound variables $x_{k+1}, \ldots, x_m$. We then unify the atoms of φ with s one at a time, starting with ϕ_1, and update the set Φ as we go along. After processing all atoms, φ unifies with s if and only if Φ is non-empty, i.e. if there remains at least one possible joint assignment to the bound variables.

To illustrate this idea, consider again the example problem introduced in Section 3.1 for finding the minimum element in the integer list $(2, 5, 3, 1, 4)$. Consider the derived predicate $\mathsf{lessthan\text{-}pointers}(a, b)$ from Figure 3.2, and let (i, j) be the assignment of variable objects to the free variables a, b. Assume that in the current state s, iterator i points to the third position of the list while iterator j points to the first position. Unification proceeds one atom at a time, and initially $\Phi = \Omega_o^2$, i.e. all joint assignments to the bound variables x_1, x_2 are possible.

The first atom $\mathsf{points\text{-}to}(i, x_1)$ unifies with $x_1 = 3$, the element in the third position of the list. Consequently, joint assignments in Φ that do not assign the value 3 to x_1 are no longer possible and thus removed. Likewise, the second atom $\mathsf{points\text{-}to}(j, x_2)$ unifies with $x_2 = 2$, and joint assignments that do not assign the value 2 to x_2 are removed from Φ. As a result, Φ contains a single possible joint assignment $(3, 2)$ to x_1, x_2.

Since 3 is not less than 2, no joint assignment in Φ unifies with the third atom $\mathsf{lessthan}(x_1, x_2)$. As a result, Φ becomes empty, and φ is considered *non-unifiable* with s.

3.2.3 Computing Planning Programs with Conjunctive Queries

In this section we extend the compilation of (Jiménez and Jonsson, 2015) to compute planning programs with conjunctive queries. The extended compilation takes as input a generalized planning problem $\Psi = \{P_1, \ldots, P_T\}$ and constants n, q and m (that bound the number of program lines, atoms and bound variables, respectively) and outputs a single planning problem $P^m_{n,q}$. A solution to $P^m_{n,q}$ corresponds to a planning program Π with conjunctive queries such that Π solves Ψ.

Since programming and executing sequential and termination instructions is identical to the original compilation, we only describe here the part of $P^m_{n,q}$ that corresponds to programming and evaluating conjunctive queries. We define a set of bound variables $X = \{x_1, \ldots, x_m\}$ and a set of *slots* $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$. Each slot is a placeholder for an atom of a conjunctive query, and we also define a dummy slot σ_0. The compilation is extended with the following novel fluents:

- For each pair of program lines i, i' such that $i' \neq i + 1$, a fluent $\mathsf{ins}_{i,\mathsf{goto}(i')}$ indicating that the instruction on line i is a goto instruction $goto(i', !\varphi)$.
- For each slot $\sigma_k \in \Sigma \cup \{\sigma_0\}$, a fluent slot^k indicating that σ_k is the current slot.
- For each line i and slot $\sigma_k \in \Sigma$, a fluent eslot^k_i indicating that slot σ_k on line i is empty.
- For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \mathcal{P}$, variable object $v \in \Omega_v$ and variable tuple $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, a fluent

$\mathsf{atom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})$ indicating that $p(v, y_2, \ldots, y_{ar(p)})$ is the atom in slot σ_k of line i.

- For each slot $\sigma_k \in \Sigma$ and object tuple $(o_1, \ldots, o_m) \in \Omega_o^m$, a fluent $\mathsf{poss}^k(o_1, \ldots, o_m)$ indicating that after processing the atoms $\{\phi_1, \ldots, \phi_k\}$, $(o_1, \ldots, o_m)$ is a possible joint assignment of objects to the bound variables $x_1, \ldots, x_m$.
- A fluent eval indicating that we are done evaluating a conjunctive query and a fluent acc representing the outcome of the evaluation (true or false).

In the initial state, all fluents above appear as false except slot^0, indicating that we are ready to program and unify the atoms of any conjunctive query. The initial state on other fluents is identical to the original compilation, as is the goal condition. We next describe the set of actions that have to be added to the original compilation to implement the mechanism for programming and evaluating conjunctive queries.

A conjunctive query φ is activated by programming a goto instruction $goto(i', !\varphi)$ on a given line i. As a result of programming the goto instruction, all slots on line i are marked as empty. For each pair of program lines i, i', the action $\mathsf{pgoto}_{i,i'}$ for programming $goto(i', !\varphi)$ on line i is defined as

$$\begin{aligned} \mathrm{pre}(\mathsf{pgoto}_{i,i'}) &= \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{nil}}\}, \\ \mathsf{cond}(\mathsf{pgoto}_{i,i'}) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,\mathsf{goto}(i')}\}, \\ &\qquad \emptyset \rhd \{\mathsf{eslot}_i^1, \ldots, \mathsf{eslot}_i^q\}\}. \end{aligned}$$

The precondition contains two fluents from the original compilation: pc_i, modeling that the program counter equals i, and $\mathsf{ins}_{i,\mathsf{nil}}$, modeling that the instruction on line i is empty.

Once activated, we have to program the individual atoms in the slots of the conjunctive query φ. After programming an atom, the slot is no longer empty. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \mathcal{P}$, variable object

$v \in \Omega_v$ and tuple of bound variables $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, the action $\mathsf{patom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})$ is defined as

$$\begin{aligned}
&\mathrm{pre}(\mathsf{patom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})) = \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{eslot}_i^k\},\\
&\mathsf{cond}(\mathsf{patom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})) =\\
&\qquad \{\emptyset \rhd \{\neg\mathsf{eslot}_i^k, \mathsf{atom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})\}\}.
\end{aligned}$$

Note that the action $\mathsf{patom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})$ assigns a concrete variable object v to its only free variable, i.e. the conjunctive queries that we generate already assign variable objects to free variables.

The key ingredient of the compilation are step actions that iterate over the atoms in each slot while propagating the remaining possible values of the bound variables. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \mathcal{P}$, variable object $v \in \Omega_v$ and tuple of bound variables $(y_2, \ldots, y_{ar(p)}) \in X^{ar(p)-1}$, step action $\mathsf{step}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})$ is defined as

$$\begin{aligned}
&\mathrm{pre}(\mathsf{step}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})) =\\
&\quad \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{atom}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})\},\\
&\mathsf{cond}(\mathsf{step}\text{-}p_i^k(v, y_2, \ldots, y_{ar(p)})) = \{\emptyset \rhd \{\neg\mathsf{slot}^{k-1}, \mathsf{slot}^k\}\}\\
&\cup \{\{\mathsf{poss}^{k-1}(o_1, \ldots, o_m), p(v, o(y_2), \ldots, o(y_{ar(p)}))\} \rhd\\
&\quad \{\mathsf{poss}^k(o_1, \ldots, o_m)\} : (o_1, \ldots, o_m) \in \Omega_o^m\}.
\end{aligned}$$

To apply a step action, an atom has to be programmed first. The unconditional effect is moving from slot σ_{k-1} to slot σ_k. In addition, the step action updates the possible assignments to the bound variables $x_1, \ldots, x_m$.

For an assignment $(o_1, \ldots, o_m)$ to be possible at slot σ^k, it has to be possible at σ^{k-1}, and the atom $p(v, y_2, \ldots, y_{ar(p)})$ programmed at slot k has to induce a fluent that is currently true. Let $o(y)$ denote the object among $o_1, \ldots, o_m$ that is associated with the bound variable y. For example, if $y = x_2$, then $o(y) = o(x_2) = o_2$. Then the induced fluent is given by $p(v, o(y_2), \ldots, o(y_{ar(p)}))$. Note that there is one conditional effect for each possible assignment $(o_1, \ldots, o_m)$. If $k = 1$, the condition

$\mathsf{poss}^{k-1}(o_1, \ldots, o_m)$ is removed since all assignments are possible prior to evaluating the first atom.

Once we have iterated over all atoms, we have to check whether there remains at least one possible assignment, thereby evaluating the entire conjunctive query. For each line i, let eval_i be an action defined as

$$\begin{aligned}\mathrm{pre}(\mathsf{eval}_i) &= \{\mathsf{pc}_i, \mathsf{slot}^q\},\\ \mathsf{cond}(\mathsf{eval}_i) &= \{\emptyset \rhd \{\mathsf{eval}\}\}\\ &\cup \{\{\mathsf{poss}^q(o_1, \ldots, o_m)\} \rhd \{\mathsf{acc}\} : (o_1, \ldots, o_m) \in \Omega_o^m\}.\end{aligned}$$

Action eval_i is only applicable once we are at the last slot σ_q. The conditional effects add the fluent acc if and only if there remains a possible assignment to $x_1, \ldots, x_m$ at σ_q.

Finally, we can now use the result of the evaluation to determine the program line that we jump to. For each pair of lines i, i', let $\mathsf{jmp}_{i,i'}$ be an action defined as

$$\begin{aligned}\mathrm{pre}(\mathsf{jmp}_{i,i'}) &= \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{goto}(i')}, \mathsf{slot}^q, \mathsf{eval}\},\\ \mathsf{cond}(\mathsf{jmp}_{i,i'}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}, \neg\mathsf{acc}, \neg\mathsf{slot}^q, \mathsf{slot}^0\}\}\\ &\cup \{\{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}\}\}\\ &\cup \{\emptyset \rhd \{\neg\mathsf{poss}^k(o_1, \ldots, o_m) : 1 \leq k \leq q, \forall j.o_j \in \Omega_o\}\}.\end{aligned}$$

The effect is to jump to line i' if acc is false, else continue execution on line $i + 1$. We also delete fluents eval and acc, as well as all instances of $\mathsf{poss}^k(o_1, \ldots, o_m)$ in order to reset the evaluation mechanism prior to the next evaluation of a conjunctive feature. The current slot is also reset to σ_0.

3.3 Properties of Planning Programs with conjunctive queries

In this section we show that our planning programs extension is sound and complete. As discussed in Chapter 2, the basic version of planning programs is sound and complete. The arguments for soundness and completeness of basic planning programs still apply to the extended version, therefore in the proofs presented in this section we only refer to the extension of the `goto` instruction.

Theorem 1 (Soundness)**.** *Any plan π that solves $P^m_{n,q}$ induces a program Π with conjunctive queries that solves Ψ.*

Proof. Planning programs with conjunctive queries only extend the `goto` instruction of basic planning programs, while the sequential and terminal instructions remain identical. Therefore we only show that the property holds for the extended `goto` instruction as well. Only empty slots are programmed by assigning atoms, which are fluents of type atom-$p^k_i(\omega)$. Once programmed the atom cannot change because of the precondition eslot^k_i, of action patom-$p^k_i(\omega)$. Therefore a conjunctive query with q atoms on a line i is represented by a set of atoms φ. Hence, we only need to show that after iterating over all the atoms of a conjunctive query φ, the fluent acc becomes true iff φ unifies with the current state s.

The number of atoms in φ is q. By induction on k we show that after iterating over the atoms $\{\phi_1, \ldots, \phi_k\}$, the fluent $\mathsf{poss}^k(o_1, \ldots, o_b)$ is true if and only if the assignment $(o_1, \ldots, o_b)$ to the bound variables $v_1, \ldots, v_b$ causes the set of atoms $\{\phi_1, \ldots, \phi_k\}$ of φ to unify with s. The base case is $k = 0$, in which case the set of atoms $\{\phi_1, \ldots, \phi_0\}$ is empty and thus trivially unifies with s. For $k > 0$, by hypothesis of induction the fluent $\mathsf{poss}^{k-1}(o_1, \ldots, o_b)$ is true if and only if $(o_1, \ldots, o_b)$ causes $\{\phi_1, \ldots, \phi_{k-1}\}$ to unify with s. Because of the definition of action step-$p^k_i(\omega)$, $\mathsf{poss}^k(o_1, \ldots, o_b)$ becomes true if and only if $\mathsf{poss}^{k-1}(o_1, \ldots, o_b)$ is true and the fluent $p(o(\omega))$ induced by p, ω and $(o_1, \ldots, o_b)$ is true in s.

This corresponds precisely to $\{\phi_1, \ldots, \phi_k\}$ unifying with s. □

Theorem 2 (Completeness)**.** *If there exists a planning program* Π *with conjunctive queries that solves* Ψ *such that* $|\Pi| \leq n$*, there exists a corresponding plan* π *that solves* $P^m_{n,q}$*.*

Proof. As before, planning programs with conjunctive queries only extend the `goto` instruction of basic planning programs. For each possible atom $p(\omega)$, there is a corresponding action $\mathsf{patom}\text{-}p^k_i(\omega)$ that programs $p(\omega)$ in slot σ^k of line i. Hence we can emulate any conjunctive query φ by programming the appropriate atoms in the slots of a line. The resulting plan π also has to check whether φ unifies with the current state s, but this is a deterministic process. The only issue is that we have to ensure that the bounds q and m are large enough to accommodate the conjunctive queries of Π. □

3.4 Classification with Planning Programs

Our extension of planning programs with conjunctive queries allows us to model supervised classification tasks as if they were generalized planning problems. Formally, the learning of a noise-free classifier from a set of labeled examples $\{e_1, \ldots, e_T\}$, where each example e_t, $1 \leq t \leq T$, is labeled with a class in $\{c_1, \ldots, c_Z\}$, can be viewed as a generalized planning problem $\Psi = \{P_1, \ldots, P_T\}$ such that each individual planning problem $P_t = \langle F, A, I_i, G_i \rangle$, $1 \leq t \leq T$, models the classification of the t^{th} example:

- $\mathcal{P}, \Omega$ induces the set of fluents F representing the learning examples and their labels.
- A contains the actions necessary to associate a given example with a class. For instance, in a binary classification task: $A = \{\mathsf{setPositive}, \mathsf{setNegative}\}$.

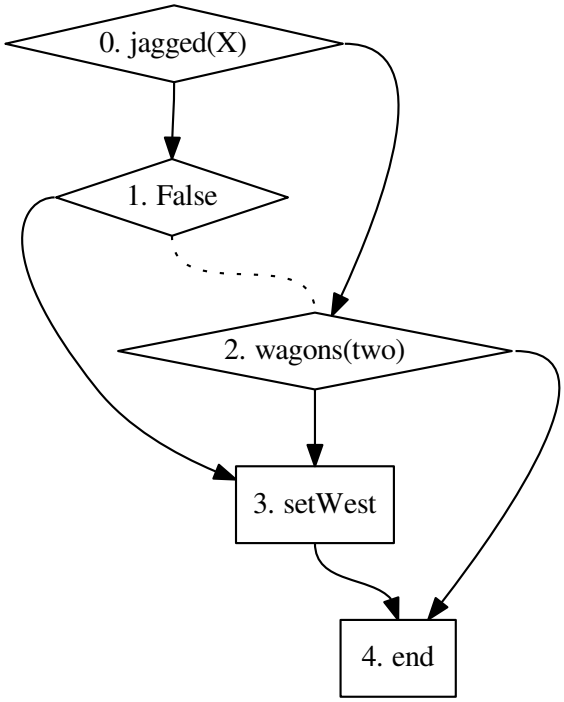


Figure 3.3: Planning program that encodes a noise-free classifier for the Michalski's train problem.

- I_t contains the fluents that describe the t^{th} example and G_t the fluent that describes the label of the t^{th} example.

The solution Π to a generalized planning problem Ψ that models a classification task is a noise-free classifier that covers all learning examples.

This model is particularly natural for classification tasks in which both the examples and the classifier are described using logic. *Michalski's train* (Michalski et al., 2013) is a good example of such tasks. It defines 10 different trains (5 traveling east and 5 traveling west) and the classification target is finding rules that cause a train to travel east or west. Trains are defined using the following relations: which wagon is in a given train, which wagon is in front, the wagon's shape, its number of wheels, whether

it has a roof or not (closed or open), whether it is long or short, the shape of the objects the wagon is loaded with and the class of the train.

In more detail, the generalized planning problem encoding the *Michalski's train* task would be:

- Fluents F induced from $\mathcal{P} = \{$(wagons ?Number), (hasCar ?Car), (infront ?Car ?Car), (shape ?Car ?Shape), (wheels ?Car ?Number), (closed ?Car), (open ?Car), (long ?Car), (short ?Car), (double ?Car), (jagged ?Car), (load ?Car ?Shape ?Number), (eastbound), (westbound)$\}$.
- Actions $A = \{$setWest$\}$. We assume that any example has initial class eastbound, causing the resulting planning programs to be more compact.

$$\begin{aligned} \mathrm{pre}(\mathsf{setWest}) &= \{\emptyset\}, \\ \mathsf{cond}(\mathsf{setWest}) &= \{\emptyset \rhd \neg\mathsf{eastbound}\} \\ &\cup \{\emptyset \rhd \mathsf{westbound}\}. \end{aligned}$$

- Each initial state I_t defines the t^{th} train and G_t defines its associated class (eastbound or westbound).

Figure 3.3 shows a planning program encoding a noise-free classifier for the Michalski's train problem. As explained, the program assumes that examples initially have class eastbound. Line 1 of the program is an unconditional jump to line 3.

3.5 Evaluation

In all experiments, we run the classical planner Fast Downward (Helmert, 2006) with the LAMA-2011 setting (Richter and Westphal, 2010) on a Intel Core i5 3.10GHz x 4 with a 4GB memory bound and time limit of 3600s.

	Lines	Slots	Vars	Time	Len
List	3	2	1	0.4	135
Summatory	3	2	1	11.0	40
Trains	5	(1,1,1)	(1,1,1)	42.0	101
And	4	(2,2)	(1,1)	0.8	49
Or	4	(2,2)	(1,1)	0.4	49
Xor	4	(2,2)	(2,1)	0.5	44

Table 3.1: Program lines, slots and variables of the features, time (in seconds) elapsed while computing the solution, and plan length required to generate and verify the solution.

We evaluate our method in two kinds of benchmarks. We first consider benchmarks from generalized planning where the target is generating a plan that generalizes without providing any prior high-level representation of the states. This set of benchmarks include iterating over a list and computing the n^{th} term of the summatory series. On the other hand, we consider binary classification tasks which include Michalski's train (cf. Section 3.4) as well as generating the classifiers corresponding to the logic functions $and(X_1, X_2)$, $or(X_1, X_2)$ and $xor(X_1, X_2)$. Table 3.1 summarizes the obtained results. We report the number of program lines used to solve the generalized planning problem, the number of slots and bound variables required to learn the features (a list means that more than one feature was learned), the time taken to generate the program, and the plan length.

We briefly describe the features and the programs, learned for the different domains. In the list domain we learn the feature $i == n$ for the program:

```
0. visit
1. inc(i)
2. goto(0,i!=n)
```

In the summatory domain we learn the feature $b = 0$ for the program:

```
0. sum(a,b)
1. dec(b)
2. goto(0,b!=0)
```

For Michalski's train we learn the program and features shown in Figure 3.3. The programs for $and(X_1, X_2)$, $or(X_1, X_2)$ and $xor(X_1, X_2)$ have the same structure: they learn a first feature that captures if a variable is false (true for the *or* function, and one true and one false for *xor*) and a second feature to capture that the class of the example was set to negative. This is the 4-line program for the $and(X_1, X_2)$ function:

```
0. goto(3, !X = False)
1. setFalse
2. goto(4,class = False)
3. setTrue
4. end
```

3.6 Discussion

In generalized planning problems, high-level state features are traditionally hand-coded which requires significant human expertise. We have proposed a novel approach to automatically generating these features by tightly integrating the computation of planning programs with the computation of the features. This integration is achieved incorporating conjunctive queries into planning programs and extending an existing compilation from generalized planning to classical planning (Jiménez and Jonsson, 2015; Segovia-Aguas et al., 2016) such that it can be exploited by an off-the-shelf planner.

Currently we are only able to generate high-level state features in the form of conjunctive queries, and hence we cannot model features with unbounded transitive or recursive closures. This kind of features are known to be useful for some planning domains, e.g. the *above* feature, the transitive closure of *on*, for the BLOCKS domain.

In addition, our approach naturally models classification tasks in which both examples and classifiers are represented using logic. The aim of this research direction is not competing with existing ML algorithms; indeed, we cannot deal with noisy examples. Instead, our aim is to provide a new formalism capable of representing tasks that integrate classification and planning. Moreover, we bring a new landscape of challenging benchmarks to classical planning.

Part III

Generating HTNs

Chapter 4

GENERATING HTNS

In this chapter we introduce a novel translation algorithm for generating HTNs. Lifted invariant graphs are discussed as a basis for the translation. Precondition ordering is introduced as a necessary optimization, for the practical use of the basic algorithm. We also compare the results of the generated HTNs, with Fast Downward blind search over a set of benchmark domains.

4.1 Introduction

HTNs are frequently used in real-world applications as they offer a potent mechanism for reducing the search effort required to solve a family of large-scale planning instances. This is also the reason that HTNs were so successful in the hand tailored track of early incarnations of the International Planning Competition (IPC): the participants were given access to the planning domains beforehand and designed HTNs that effectively narrowed the search to a tiny portion of the state space. It is not a coincidence that the HTN planner that achieved the largest coverage at IPC-2002 and was for a long time regarded as the state-of-the-art in HTN plan-

ning, SHOP2 (Nau et al., 2003), performs blind search in the task space to compute a valid expansion. Most of the work required to reduce the search effort is performed while designing the HTN, and once this work is done, there is little need to optimize search to solve each instance efficiently. Identifying decomposition strategies is an arduous task even for experts and can even lead to models, which yield solutions that are not sound, once translated to the classical planning setting. HTN instances only define the initial task list which needs to be decomposed in contrast to a goal condition in the classical planning instance. This can lead to difficulties in expressing classical planning goals. Depending on the encoding, tasks often do not directly correspond to the goals of the original domain.

The success of the HTNs in practice, is due to expressive power which enables human experts to encode prior knowledge about the hierarchical structure of a planning domain. Arguably this is more difficult, than creating a flat PDDL domain which specifies only preconditions and effects for each action. The effort of creating HTNs manually is still acceptable for specific planning applications, in which the initial effort is compensated by the subsequent reduction in search time during successive applications of the planner. However, a large body of research in the planning community is dedicated to finding domain-independent approaches to planning. Traditionally, HTNs have not found a place in this body of research because of their domain-dependent emphasis.

A key motivation for this work is to explore whether it is possible to generate HTNs automatically in a domain-independent way. We also want to investigate whether such HTNs can effectively reduce the search space, for any instance of a planning domain. Another benefit to generating HTNs is the possibility of consistent and provable soundness of the generated models. In the literature there exist two techniques that construct HTNs automatically (Hogg et al., 2008; Zhuo et al., 2009). These techniques rely on annotated traces of plans that solve a set of instances from a domain. In contrast, the algorithm presented in this chapter relies only on the planning domain and one example instance that does not need to

be solved.

In this chapter we present a novel algorithm for generating HTNs automatically. Our algorithm takes as input the PDDL description of a planning domain and a single representative instance. Unlike previous approaches, the algorithm does not require solution plans for a subset of instances of the domain. Instead, our approach is to generate HTNs that encode invariant graphs of planning domains. An invariant graph is similar to a lifted domain transition graph, but can be subdivided on types. To traverse an invariant graph we define two types of tasks: one that reaches a certain node of an invariant graph, achieving the associated fluent, and one that traverses a single edge of an invariant graph, applying the associated action. These two types of tasks are interleaved, in that the expansion of one type of task involves tasks of the other type. We test the algorithm on a series of IPC benchmark domains, using the JSHOP2 planner, and compare the results to Fast Downward blind search. The reason we compare to FD blind search is that JSHOP2 also performs blind search, so the comparison gives us a good idea of how much the search space is reduced.

As a running example in this chapter, we use the IPC-2000 LOGISTICS planning domain and this instance:

```
(define (problem logistics-example)
(:domain logistics)
(:objects a1 - airplane ap1 ap2 - airport
          c1 c2 - city l1 l2 - location
          t1 t2 - truck p1 - package)

(:init (at p1 l1) (at a1 ap2)
       (at t1 l1) (incity l1 c1) (incity ap1 c1)
       (at t2 l2) (incity l2 c2) (incity ap2 c2))

(:goal (and (at p1 l2))))
```

4.2 Invariant graphs

Given a STRIPS planning problem $P = \langle F, A, I, G \rangle$, a mutex invariant is a subset of fluents $F' \subseteq F$ such that at most one is true at any moment. Formally, $|F' \cap I| = 1$ and any action $a \in A$ that adds a fluent in F' deletes another. The Fast Downward planning system (Helmert, 2009) uses the domain description to detect lifted invariant candidates. Unlike Fast Downward, which grounds lifted invariants on actual instances, our algorithm operates directly on the lifted invariants.

In LOGISTICS, Fast Downward finds a single lifted invariant, i.e. a set of predicates with associated variable lists: $\{(\texttt{in ?o ?v}), (\texttt{at ?o ?p})\}$.

All objects which appear as parameters of all predicates in the invariant candidate are considered as *bound*, while the rest of the parameters are *free*. In the given example, variable ?o is *bound* while ?v and ?p are *free*. To ground the lifted invariant on an instance $\boldsymbol{p}$, we should create one mutex invariant F' for each assignment of objects to the bound variables, and each fluent in F' is obtained by assigning objects to the free variables. In our running example, assigning the package p1 to ?o results in the following grounded mutex invariant:

$\{(\texttt{at p1 ap1}),(\texttt{at p1 ap2}),(\texttt{at p1 l1}),(\texttt{at p1 l2}),$
$(\texttt{in p1 t1}),(\texttt{in p1 t2}),(\texttt{in p1 a1})\}$.

The meaning of the invariant is that across all LOGISTICS instances, a given object ?o is either in a vehicle or at a location.

If a predicate $p \in \mathcal{P}$ is not part of any invariant but there are actions that add and/or delete p, we create a new invariant
$\{(p\ \texttt{?o1} \cdots \texttt{?ok}),(\neg p\ \texttt{?o1} \cdots \texttt{?ok}))\}$, where all variables are bound and an associated fluent can either be true or false.

Given an invariant, our algorithm generates one or several invariant graphs. In LOGISTICS, all actions affect the lone invariant above. However, when loading or unloading a package, the bound object ?o is a package, when

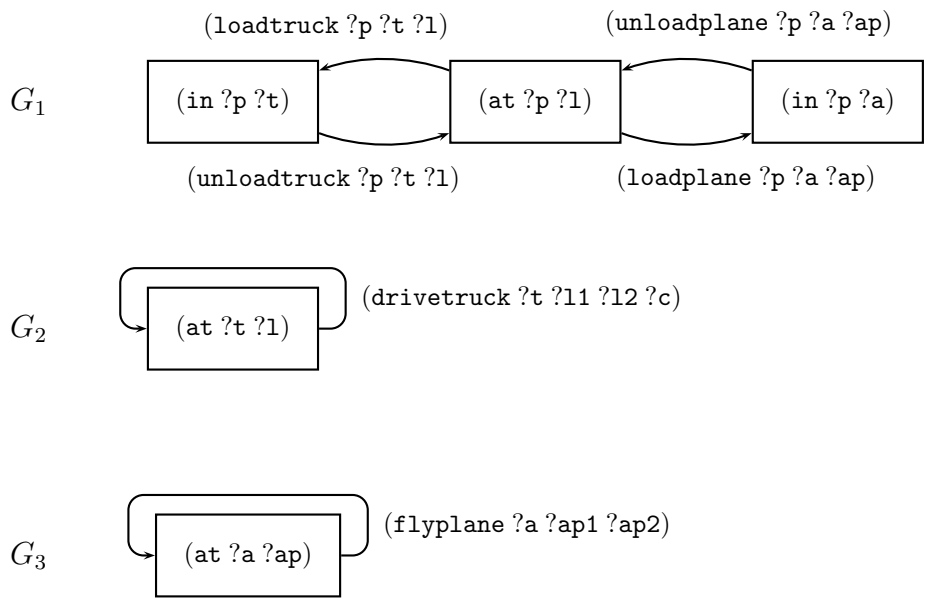


Figure 4.1: Invariant graphs (G_1, G_2 and G_3) in LOGISTICS.

driving a truck ?o is a truck, and when flying an airplane ?o is an airplane. Moreover, we can either load a package into a truck or an airplane. We differentiate between types such that each invariant may generate multiple invariant graphs.

To generate the invariant graphs we go through each action, find each transition of each invariant that it induces (by pairing add and delete effects and testing whether the bound objects are identical), and map the types of the predicates to the invariant. We then either create a new invariant graph for the bound types or add nodes to an existing graph corresponding to the mapped predicate parameters.

Figure 4.1 shows the invariant graphs in LOGISTICS. In the top graph (G_1), the bound object is a package ?p, in the middle graph (G_2) a truck ?t, and in the bottom graph (G_3) an airplane ?a. Note that the predicate in is not actually part of the two bottom graphs, since trucks and planes

cannot be inside other vehicles. Nevertheless, the invariant still applies: a truck or plane can only be at a single place at once.

Each edge of an invariant graph corresponds to an action that deletes one predicate of the invariant and adds another. To do so, the parameters of the action have to include the parameters of both predicates, including the bound objects. In the figure, the invariant notation is extended to actions on edges such that each parameter is either bound or free.

Even if actions preserve the invariant property, the initial state of a planning instance may violate the condition $|F' \cap I| = 1$, in which case F' is not a mutex invariant. To verify that an invariant corresponds to actual grounded invariants, our algorithm needs access to the initial state of an example planning instance $\boldsymbol{p}$ of the domain. If this verification fails, the invariant is not considered by the algorithm.

4.3 Translation algorithm

In this section we describe our algorithm for automatically generating HTN domains. The idea is to construct a hierarchy of tasks that traverse the invariant graphs to achieve certain fluents. In doing so there are two types of interleaved tasks: one that achieves a fluent in a given invariant (which involves applying a series of actions to traverse the edges of the graph), and one that applies the action on a given edge (which involves achieving the preconditions of the action). Figure 4.2 shows the overall architecture facilitating the translation.

Formally, our algorithm takes as input a STRIPS planning domain $\boldsymbol{d} = \langle \mathcal{T}, <, \mathcal{P}, \mathcal{A} \rangle$ and a planning instance $\boldsymbol{p} = \langle \Omega, \mathrm{init}, \mathrm{goal} \rangle$ and outputs an HTN domain $\boldsymbol{h} = \langle \mathcal{P}', \mathcal{A}', \mathcal{C}, \mathcal{M} \rangle$. We assume that the preconditions and goals of the input planning domain are positive. The HTN domain $\boldsymbol{h}$ can then be used to solve any other instance of the domain. Specifically, for each instance $\boldsymbol{p}'$ of the planning domain $\boldsymbol{d}$, we show how to construct

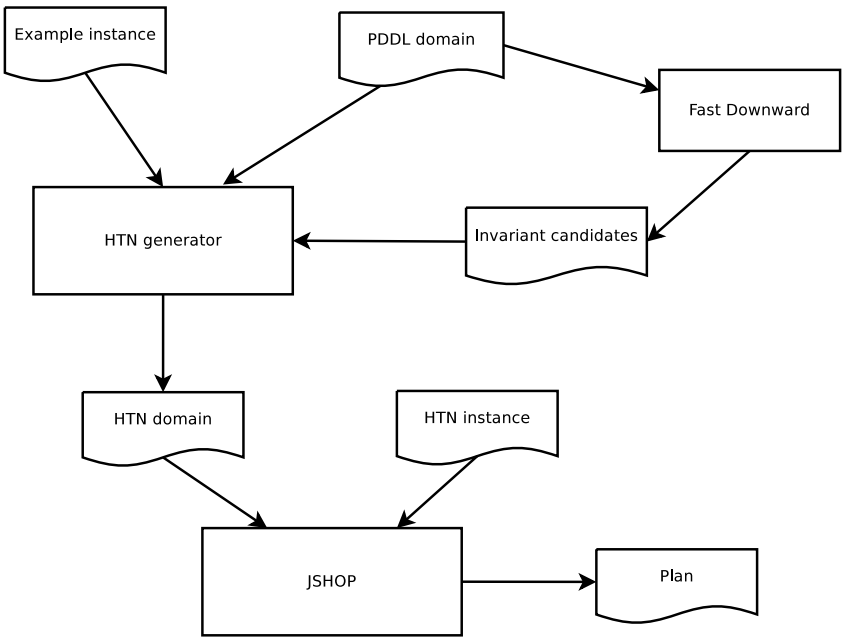


Figure 4.2: Architecture overview

an HTN instance ***s***, for the HTN domain ***h***. Solving the HTN induced by ***h*** and ***s*** returns a plan that can be adapted to solve $\boldsymbol{p}'$.

The input planning instance ***p*** is used for two purposes:

1. To verify that an invariant candidate is actually an invariant by testing the condition $|F' \cap \text{init}| = 1$.

2. To extract a subset of predicates $\mathcal{P}_G \subseteq \mathcal{P}$ that are part of the goal.

The algorithm first constructs the invariant graphs $G_1, \ldots, G_k$ described above. In the following subsections we describe the components of the HTN domain ***h***.

4.3.1 Predicates

The set $\mathcal{P}' \supseteq \mathcal{P}$ extends $\mathcal{P}$ with the following predicates:

- For each predicate $p \in \mathcal{P}$, a predicate `visited-`p with arity $\alpha(p)$ indicating that p has already been visited during search.
- For each predicate $p \in \mathcal{P}$, a predicate `achieving-`p with arity $\alpha(p)$ indicating that p or another predicate in the same invariant are already being achieved.
- For each goal predicate $p \in \mathcal{P}_G$, a predicate `goal-`p with arity $\alpha(p)$ indicating that a fluent derived from p is a goal state.
- For each type $\tau \in \mathcal{T}$, a type predicate τ with arity 1, since HTN domains are untyped.
- `ogoal-`p, with arity $\alpha(1+\alpha(p))$ indicating that a fluent derived from `goal-`p is an ordered goal state. The extra parameter is an ordinal cnt which denotes the order of the goal fluent.
- `ordered-goal-`p with arity $\alpha(p)$, which marks that the fluent instantiated from `goal-`p has been assigned an ordinal.

4.3.2 Primitive tasks

The set $\mathcal{A}'$ contains the following actions:

- Each action $a \in \mathcal{A}$. For each element $\beta_k(a) \in \mathcal{T}$ of the type list of a, we add an additional precondition $(\beta_k(a), \varphi_k)$. where the argument map φ_k maps the argument x_k of a to the lone argument of the type predicate $\beta_k(a)$, ensuring that argument x_k has the correct type.
- For each $p \in \mathcal{P}$, an action `visit-`p with arity $\alpha(p)$ that marks p as visited by adding `visited-`p.

- For each invariant graph G_i, an action `occupy-`i whose arity equals the number of bound objects in G_i, and that marks each predicate in G_i as being achieved.
- For each invariant graph G_i, an action `clear-`i whose arity equals the number of bound objects in G_i, and that deletes `visited-`p and `achieving-`p for each predicate p of G_i.
- For each goal predicate $p \in \mathcal{P}_G$, an action `test-order-`p with arity 0 and no effects, whose precondition tests if *all* goal fluents derived from p have been ordered.
- `finish-ogoal-`p with arity $1+\alpha(p)$, assigns an ordinal cnt to a goal fluent by adding the `ogoal-`p fluent. It also adds `ordered-goal-`p to mark that the goal has been ordered.
- `test-order-`p with arity 0, checks if all goal fluents instantiated by the same predicate $p \in \mathcal{P}_G$, have been assigned an ordinal number.
- For each $p_i \in \mathcal{P}_G$ we add a `test-`p_i task, with arity 0, which checks if all goal fluents instantiated by the same predicate $p_i \in \mathcal{P}_G$, hold in the current state.

Note that only action schemata in $\mathcal{A}$ add or delete predicates in the original set $\mathcal{P}$. All other actions only have effects on predicates in $\mathcal{P}' \setminus \mathcal{P}$.

4.3.3 Compound tasks

The set $\mathcal{C}$ contains six types of compound tasks:

- For each predicate $p \in \mathcal{P}$ that appears as positive in any invariant graph, a task `achieve-`p.
- For each invariant graph G_i and each $p \in \mathcal{P}$ that is positive in G_i, a task `achieve-`p`-`i.
- For each invariant graph G_i, each predicate $p \in \mathcal{P}$ in G_i, and each

outgoing edge of p (corresponding to an action $a \in \mathcal{A}$), a task `do-`p`-`a`-`i.

- `order` task, with arity 1, which assigns an ordinal number to each goal fluent from the input planning problem. The only parameter is the ordinal cnt, which is assigned to the current goal fluent.
- `solve` task, with arity 1. Same as `order` task, the only parameter is cnt, used to identify the current fluent to be achieved.
- `find-plan` is the root task, used to impose an order between the decomposition of `order` and `solve` tasks.

The `achieve-`p task is a wrapper task that achieves a predicate p in any invariant, while the second and third task are the interleaved tasks for achieving p by traversing the edges of an invariant graph G_i. Since the preconditions and goals of the input planning domain are positive, we never have to achieve a negated fluent. The root task `solve` is the wrapper task which selects the next `achieve-`p tasks to be achieved according to the order imposed on the goal fluents. Since this is the only task specified in the task list of any problem instance it is used to recursively specify the goal fluents which need to be achieved according to the goal ordering. The goal ordering for fluents that we consider in this section is equivalent to the ordering in the original PDDL instance. In the basic version of the algorithm the `order` task assigns ordinals to the `goal-`p fluents, by the order in which they appear in the PDDL instance. The only task that appears in the initial task list to be achieved is `find-plan`. This task decomposes further to the `order` and `solve` and enforces the decomposition of `order` task before the `solve` task.

4.3.4 Methods

Finally, the set $\mathcal{M}$ contains the following decomposition methods. We describe methods in pseudo-SHOP2 syntax in the following format:

(:method (⟨name⟩[⟨arguments⟩])
(⟨precondition⟩)
(⟨tasklist⟩))

For each method in the first line we specify name and arguments, in the second line we give precondition list, and finally in the third we specify task list to which method decomposes.

Achieve methods

The first type of compound task, achieve-p, has one associated method for each invariant graph G_i in which p appears. An outline of this method is given by:

(:method (achieve-$p[x]$)
((¬achieving-$p[x]$))
((!occupy-$i[\varphi_i(x)]$) (achieve-p-$i[x]$) (!clear-$i[\varphi_i(x)]$)))

The argument map φ_i maps the arguments of p to the bound variables of the invariant graph G_i. Intuitively this method delegates achieving p to the task achieve-p-i for some invariant graph G_i. The method first adds achieving-p' for each predicate p' in G_i, and clears the flags after achieving p. The precondition (¬achieving-$p[x]$) prevents us from achieving p if it is part of an occupied invariant graph, which could potentially lead to an infinite recursion.

Achieve in graph and Do methods

The second type of compound task, achieve-p-i, has one associated method for each predicate p' in the invariant graph G_i and each outgoing edge of p' (corresponding to an action a), and one method which is a "base case" for a recursion:

1. (:method (achieve-p-$i[x]$)
   (($p'[\varphi'(x)]$) (¬visited-$p'[\varphi'(x)]$))
   ((!visit-$p'[\varphi'(x)]$) (do-p'-a-$i[\varphi_a(x)]$) (achieve-p-$i[x]$))).

   Action a appears on an outgoing edge from p', i.e. a deletes p'. Intuitively, one way to achieve p in G_i, given that we are currently at some different node p', is to traverse the edge associated with a using the compound task do-p'-a-i. Before doing so we mark p' as visited to prevent us from visiting p' again. After traversing the edge we recursively achieve p from the resulting node. The argument map φ' should set the bound objects of p' while leaving other arguments of p' as free variables. Likewise, the argument map φ_a should set the bound objects of a. The precondition (¬visited-$p'[\varphi'(x)]$) prevents us from visiting the same node p' twice.

   In essence, the result is a depth-first search through the invariant graph G_i, which stops when we reach $p[x]$. Recall that the flags visited-p and achiveing-p are cleared by the parent method once we reach $p[x]$, using the clear-i action.

2. To stop the recursion we define a "base case" method:

   (:method (achieve-p-$i[x]$)
   (($p[x]$))
   ())

   This method is applicable when $p[x]$ already holds and has empty task list.

The third type of compound task, do-p-a-i, has only one associated method. The aim is to apply action a to traverse an outgoing edge of p in the invariant graph G_i. To do so, the task list has to ensure that all preconditions $p_1, \ldots, p_k$ of a hold (excluding p, which holds by definition, as well as any static preconditions of a). We define the method as

(:method (do-p-a-$i[x]$)
(($p[\varphi(x)]$))

$(((\texttt{achieve-}p_1[\varphi_1(x)]) \cdots (\texttt{achieve-}p_k[\varphi_k(x)]))\ (!a[x]))$

Here, (p, φ) is the precondition of a associated with p, while (p_j, φ_j), $1 \leq j \leq k$, are the remaining preconditions of a. The decomposition achieves all preconditions of a, then applies a. Note that if action a has no preconditions except p, the mutual recursion stops since the decomposition does not contain any task of type achieve-p_j. In this case our approach is to simplify the definition of other methods by replacing any instance of do-p-a-i with the action a itself.

Order and Solve methods

The SHOP2 syntax allows ordinal variables. We use ordinals to order the goal fluents and to achieve them in the specified order. We always impose an ordering over the goal fluents of any given planning instance. The order task encodes the goal ordering strategy. In the basic translation, the goal fluents are ordered in the order of appearance in a goal condition of a given instance. To decompose the order task we define two types of methods, for each goal predicate $p \in \mathcal{P}_G$, that appears in the goal of the example instance:

1. The first method decomposes to the primitive task finish-ogoal-$p[x, cnt]$, which adds the fluent ogoal-$p[x, cnt]$, where the ordinal cnt denotes when the specific goal fluent should be achieved. The same primitive task also adds the ordered-goal-$p[x]$ fluent, to mark that the fluent goal-$p[x]$ has been ordered. Finally the first method adds the order$[cnt + 1]$ task, in order to continue the recursion.

   (:method (order$[cnt]$)
   ((goal-$p[x]$) ($\neg$ordered-goal-$p[x]$))
   ((!finish-ogoal-$p[x, cnt]$) (order$[cnt + 1]$)))

2. The second type of the order method decomposes to the (!test-order-p) with no preconditions. The (!test-order-p) operator has

empty add and del lists, and is used only to terminate recursion if and only if all goals have been ordered.

(:method (order[cnt])
()
((!test-order-p)))

The solve task has three decomposition methods:

1. The first type of method is generated for each predicate $p \in \mathcal{P}_G$. The method attempts to achieve predicate p, only if it does not already hold in current state, and if it should be achieved next, according to the goal order. To track which goal fluent should be achieved next, the planner internally maintains a current value of the counter, and the current value has to match the parameter of the order method. After achieving p the counter is reset to zero, before the recursively decomposing the solve task, as some of the goals that were previously achieved may have been deleted in order to achieve p.

   (:method (solve[cnt])
   (($\neg p[x]$) (ogoal-$p[x, cnt]$))
   ((achieve-$p[x]$) (solve[0]))

2. The second type of method is also generated for each predicate $p \in \mathcal{P}_G$. The method decomposes task solve, increases the counter cnt if the goal fluent p already holds in current state and continues with the recursion. Effectively, this method skips the fluents which have already been achieved, only if they hold in the current state, otherwise a different decomposition method should decompose the solve task:

   (:method (solve[cnt])
   ($p[x]$)(ogoal-$p[x, cnt]$))
   ((solve[$cnt + 1$]))

3. For the `solve` task, we also add a method which tests if all fluents specified in the goal of the original planning instance, have been achieved. This method is used to determine if the goal state has been reached and is a "base case" for the recursive `solve` methods. The method decomposes to a list of primitive tasks. Each `test-`p_i task, checks if all goal fluents instantiated from a predicate $p_i \in \mathcal{P}_G$, have been achieved. Therefore, the method can only be decomposed if all the fluents specified in the goal condition of the original instance, hold in the current state.

   (:`method` (`solve`$[cnt]$)
   ()
   ((!`test-`p_1) $\cdots$ (!`test-`$p_{|\mathcal{P}_G|}$))

Find-plan method

To restrict the choices when traversing the HTN, we impose a total order on all task lists of methods, except tasks (`achieve-`p_1) $\cdots$ (`achieve-`p_k) of the method `do-`p-a-i, since it may be difficult to determine in which order to achieve the preconditions of an action. Therefore, we add the `find-plan` method, which enforces the decomposition of the `order` task before the `solve` task.

(:`method` (`find-plan`)
()
((`order 0`)(`solve 0`)))

4.3.5 Planning Instances

Once we have generated the HTN domain $\boldsymbol{h}$ we can apply it to any instance of the domain. Given a STRIPS instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, we construct an HTN instance $\boldsymbol{s} = \langle \Omega, \text{init}', \langle \texttt{find-plan}, \emptyset \rangle \rangle$ as follows. The

set of objects $\Omega = \Omega_1 \cup \cdots \cup \Omega_n$ is identical to that of $\boldsymbol{p}$. The initial state init' is defined as

$$\text{init}' = \text{init} \cup \{\tau_j[\omega] : \tau_j \in \mathcal{T}, \omega \in \Omega_j\} \cup \{\texttt{goal-}p[x] : p[x] \in \text{goal}\}.$$

We thus mark the type τ_j of each object ω using the fluent $\tau_j[\omega]$, and we mark all fluents $p[x]$ in the goal state using the fluent $\texttt{goal-}p[x]$. The initial task network consists of a single task `find-plan` and an empty set of relations: $\langle \texttt{find-plan}, \emptyset \rangle$.

4.3.6 Example

In LOGISTICS, our algorithm generates two wrapper tasks `achieve-in` and `achieve-at`, and four tasks `achieve-in-1`, `achieve-at-1`, `achieve-at-2`, and `achieve-at-3`, corresponding to the predicates in the three invariant graphs. The task `achieve-at-1` has five associated methods: one for each edge of the graph G_1, plus the base case method.

The algorithm also generates six tasks `do-at-loadtruck-1`, `do-at-loadplane-1`, `do-at-unloadtruck-1`, `do-at-unloadplane-1`, `do-at-drivetruck-2`, and `do-at-flyplane-3`, corresponding to the six edges of the graphs. The latter two do not have preconditions besides `at` (the predicate `incity` in the precondition of `drivetruck` is static). The remaining four tasks each achieve a single precondition: the truck or plane being at the associated place.

To illustrate the tasks and associated methods, we sketch the task expansions of the HTN instance generated from our running example. The only goal is (`at p1 ap1`), so the task `solve` has a single valid decomposition that contains the task (`achieve-at p1 l2`). Table 4.1 shows the first five task expansions of `solve` task. In each case, the compound task to be decomposed is underlined, and the new tasks inserted as a result of the decomposition are colored in the next step. Note that the `solve` task is decomposed with parameter 0, assuming that the only goal fluent has been assigned ordinal 0 during the goal ordering.

(solve 0)	(achieve-at p1 l2)	(!occupy-1 p1)	(!occupy-1 p1)	(!occupy-1 p1)	(!occupy-1 p1)
	(solve 0)	(achieve-at-1 p1 l2)	(!visit-at p1 l1)	(!visit-at p1 l1)	(!visit-at p1 l1)
		(clear-1 p1)	(do-at-loadtruck-1 p1 t1 l1)	(achieve-at t1 l1)	(!occupy-2 t1)
		(solve 0)	(achieve-at-1 p1 l2)	(!loadtruck p1 t1 l1)	(achieve-at-2 t1 l1)
			(!clear-1 p1)	(achieve-at-1 p1 l2)	(!clear-2 t1)
			(solve 0)	(!clear-1 p1)	(!loadtruck p1 t1 l1)
				(solve 0)	(achieve-at-1 p1 l2)
					(!clear-1 p1)
					(solve 0)

Table 4.1: The first five task expansions of the `solve` task generated from the running example in LOGISTICS. The colored tasks are those added by the decomposition of the underlined task in the previous step.

The second decomposition is produced by the lone method for (achieve-at p1 l2). The current node associated with p1 in G_1 is (at p1 l1), with two outgoing edges, corresponding to actions loadtruck and loadplane. Applying the method for (achieve-at-1 p1 l2) associated with (at p1 l1) and loadtruck produces the third expansion. The only method for (do-at-loadtruck-1 p1 t1 l1) expands to (achieve-at t1 l1), which in turn expands to (achieve-at-2 t1 l1) (the last expansion shown).

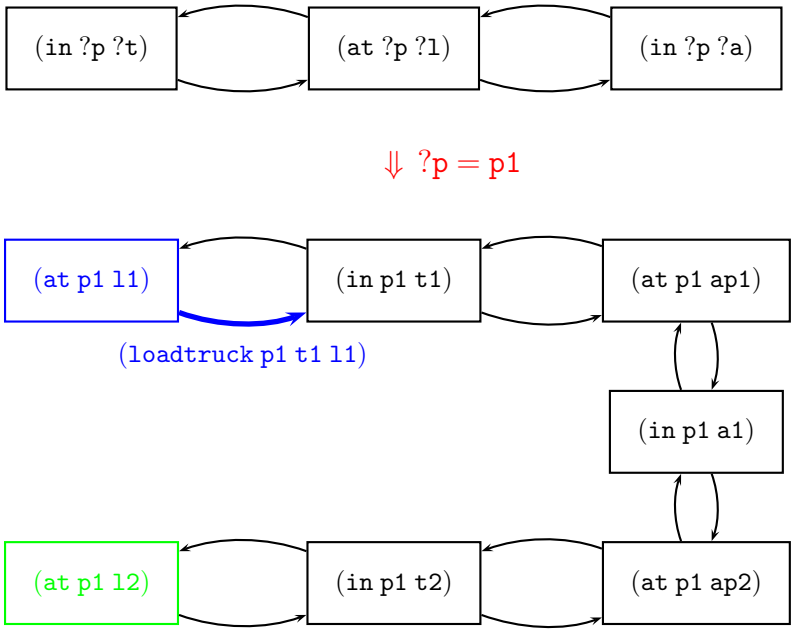


Figure 4.3: Example of search in the invariant graph G_1 in LOGISTICS domain.

The expansions shown in the Table 4.1, represent the search through the grounded invariant graph G_1, of the LOGISTICS domain. The search through the instantiated ground invariant graph G_1 and the assignment p1 to the package, is also illustrated in the Figure 4.3, where the *bound* parameter ?p representing the package, takes the value of p1, due to the goal

(`at p1 l2`). As depicted the search starts with the node (`at p1 l1`), and at least six edges need to be traversed to reach the goal (`at p1 l2`). Note that decomposing the HTN, does not ground the whole invariant graph at the beginning of the search, rather the grounding is performed gradually. As the decompositions are performed, the free parameters of the methods are chosen in a nondeterministic manner, and thus the invariant graph is explored. In the given example of decompositions, object `a1` which represents an airplane, does not appear as a parameter any any of the methods or actions. However, given the example instance, the appropriate object (airplane `a1`) is non-deterministically chosen by the HTN planner, in the decompositions which follow the ones shown in the Table 4.1. This is also one of the reasons why an HTN planner has to perform search to generate a solution.

4.4 Ordering Preconditions

Achieving the preconditions of an action a in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly to find a correct order. For this reason, we include an extension of our algorithm that uses a simple inference technique to compute a partial order in which to achieve the preconditions of a.

We define a set of predicates whose value is supposed to persist, and check whether a path through an invariant graph is applicable given these persisting predicates. While doing so, only the values of bound variables are known, while free variables can take on any value. Matching the bound variables of predicates and actions enables us to determine whether an action allows a predicate to persist.

Consider a task of type `do-`p`-`a`-`i, i.e. using action a to delete p. Figure 4.4 shows how to order all preconditions of a except p. In the algorithm, V is the set of preconditions to be ordered, while Z is a sequence of preconditions, initially empty. The algorithm considers one precondition $p' \in V$ at

```
1: function ORDER(a, p)
2:     V ← pre(a) \ {p}, Z ← ⟨⟩
3:     repeat
4:         for p′ ∈ V do
5:             W ← {p} ∪ V \ {p′}
6:             for each invariant graph G_j containing p′ do
7:                 Perform a backwards BFS in G_j from p′
8:                 Test if paths applicable when W persists
9:             end for
10:            if each path achieving p′ is applicable then
11:                V ← V \ {p′}
12:                Z ← ⟨p′, Z⟩
13:            end if
14:        end for
15:    until V, Z converge
16:    return (V, Z)
17: end function
```

Figure 4.4: Algorithm ordering preconditions of a except p.

a time and checks if we can achieve p' while all remaining preconditions persist. If so, we remove p' from V and place it first in Z. We then iterate until no more preconditions can be removed from V, and return (V, Z). In the method m associated with `do`-p-a-i, the preconditions in Z can be achieved in order. On the other hand, we cannot say anything about the order of preconditions that remain in V.

4.5 Properties of the generated HTNs

In this section we will discuss soundness and classify the domains for which the translation is complete. We show that the HTN translation is sound, both with and without any optimizations.

The set of auxiliary operators is given as $\mathcal{A}' \setminus \mathcal{A}$, and contains the operators which decompose all of the primitive tasks, except for the ones based on action schemata $a \in \mathcal{A}$ from the input domain $\boldsymbol{d}$. The auxiliary operators do not modify the fluents from the original input domain $\boldsymbol{d}$.

Theorem 3 (Soundness)**.** *Let π be a valid decomposition for $\boldsymbol{s}$, and construct π' by removing operators in $\mathcal{A}' \setminus \mathcal{A}$. Then π' solves $\boldsymbol{p}$.*

Proof. The initial state of fluents in F induced from the original planning instance, is the same as in the input planning problem. For π to be a valid decomposition for $\boldsymbol{s}$, the precondition of each operator o_i has to hold following the application of $o_1, \ldots, o_{i-1}$ and it has to terminate with operator `test-achieved-`p, which verifies that the goal state G on fluents in F holds at the end of π. Since we do progression, each time an operator is applied in a planning state, we verify that its preconditions hold and update the planning state. Since all of the operators from $\mathcal{A}' \setminus \mathcal{A}$ are removed, the subsequence π' of π only contains operators that directly correspond to actions of $\mathcal{A}$ of the input domain $\boldsymbol{d}$.

Consider now the subsequence π' of π that only contains operators which correspond to actions in $\mathcal{A}$, and consider only the fluents in F. Since π is a valid progression and satisfies the goal condition G in the end, the operators in π' have to be the ones that achieve G, as no other operators have effects on fluents in F. Since progression verified that the precondition of each operator is satisfied, π' has to be a valid plan in the original planning instance, and since G holds at the end, π' solves the original instance. □

Once the HTN is generated, as long as the methods that decompose the solve task do not modify or introduce an operator of the generated HTN, and do not use the operator `test-`p, the HTN translation will remain sound. The only way to terminate execution is to apply the `test-`p operator, which is only accessible by the method which decomposes the `solve` task. The operators would remain the same, thus only the operators based on the original actions would retain the ability to modify the original fluents.

While the HTN translation provides a sound framework it is not complete in the general case. One reason is that the goal ordering in the basic variant of our algorithm (HTNPrecon), orders the goal fluents in the same order, as they appear in the original instance, and goal ordering optimization is not applicable to all domains. This could possibly be alleviated by exploring all possible goal ordering, however this approach would not be computationally feasible in practice.

A valid PDDL domain allows for the construction of the invariant graphs, such that a precondition must be achieved before the edge that directly requires the precondition is even reached. One such example is presented in the Figure 4.5. Suppose that the initial state of the planning instance is $\texttt{init} = \{\texttt{x}, \texttt{u}\}$ and that the goal is $\texttt{goal} = \{\texttt{z}\}$. Given that the action $(\texttt{yz})$ has one precondition $\texttt{v}$, and the only precondition of the $(\texttt{uv})$ is $\texttt{x}$, obviously, in any valid plan, first the action $(\texttt{uv})$ would have to be applied.

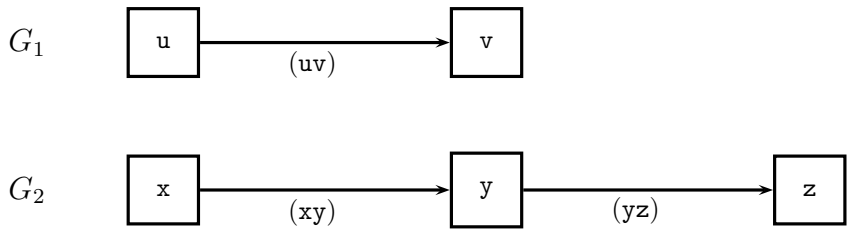


Figure 4.5: Invariant graphs (G1, G2), where $(\texttt{uv})$, $(\texttt{xy})$, $(\texttt{yz})$ represent actions, while the nodes represent predicates. The preconditions of each action are: $prec(uv) = \{x\}$, $prec(xy) = \emptyset$, $prec(yz) = \{v\}$

In that case predicate $\texttt{z}$ would have to be achieved in the second invariant graph. The only graph in which $\texttt{z}$ can be achieved is G_2. Therefore following the HTN definitions the compound task `achieve-z2` will always be decomposed to achieve $\texttt{z}$. First the edge $(\texttt{xy})$ would be traversed. This would lead to a state which does not contain $(\texttt{x})$ and thus the edge $(\texttt{yz})$ would not be traversable. Since the search started in G_2, even if the

planner tries to traverse the edge in G_1 it can not be traversed after the action $(\texttt{xy})$ has been applied. Only when node `y` is reached, the achieving the precondition `v` of the action $(\texttt{yz})$ is attempted, however by then it is already too late.

As demonstrated in the example, the search can not initially advance in the graph that doesn't contain the goal predicate as a node, since the corresponding `achieve-p-`i method will not be generated. Since the compilation is in general not complete, we consider a restricted class of planning instances that satisfies the following restrictions:

- We consider the version of the translation algorithm with no optimizations.
- The goal fluents are trivially serializable
- The set of goal predicates $\mathcal{P}_G$ of the example instance, is equal to the set of goal predicates of any instance ***p*** in a given domain ***d***.
- Each domain transition graph is reversible
- The causal graph of the input planning domain is acyclic.

We will refer to this restricted class of planning instances as "**RA**" class.

Theorem 4 (Completeness)**.** *Let h be the HTN domain translation of the planning domain d. Then for a given planning instance **p** which belongs to the **RA** class, if a valid plan π exists, the decomposition of tasks in **s** using methods defined in **h** will return a valid plan π'. Otherwise "fail" will be returned.*

Proof. The instance ***s*** consists of a single task `find-plan`. It decomposes to `order` and `solve` task, in the respective order. The `solve` solve task iterates over the `achieve-`p methods. These methods achieve the goal fluent p from ***p*** in any invariant graph which contains the target predicate as a node.

In an invariant graph G_i, between any two connected nodes `t` and `p`, an

edge can be traversed by applying an action a, iff `t` holds in the current state along with all preconditions of a. If no actions fulfill this condition, in any of the invariant graphs which contain `p` as a node, then the algorithm returns "fail". If the edge is traversed and no preconditions need to be achieved, it will only traverse an edge in the graph G_i, as the causal graph is acyclic.

If the preconditions of `a` do not hold, they will first be achieved. This can cause a traversal of an edge in another invariant graph G_{prec}. However, as the causal graph is acyclic, each of the `achieve-`p tasks will only traverse an edge in one invariant graph only. Therefore each `achieve-`p forms an independent search in an invariant graph.

By decomposing the `achieve-`p tasks, any fluent instantiated from predicate p can be achieved in an invariant graph which contains p, iff p is reachable from the initial state. Since the goals are trivially serializable, no goal will have to be reachieved.

□

4.6 Experimental results

To test how well the HTN domain restricts search, we performed experiments with a basic version of our algorithm and the precondition ordering optimization (HTNPrecon) over a set of IPC domains. We compare to Fast Downward with blind search (FDBlind). In all experiments we used JSHOP2 (the Java implementation of SHOP2), which uses blind search to compute a valid expansion. We used a memory limit of 4GB and a timeout of 1,800 seconds. The comparison is relevant, because Fast Downward instantiates the variables in the multivalued representation, based on mutex groups. Mutex groups are instantiated from lifted invariant candidates, which HTNPrecon uses to generate the invariant graphs. However, the Fast Downward algorithm further optimizes the set of variables, by approximating a set cover of the mutex groups, and performs other opti-

mizations before planning. In contrast, HTNPrecon does not have access to the grounded invariant graphs at the beginning of the search. Another reason that the comparison is relevant is that performing blind search in both cases tells us something about how restricted the search in the generated HTN is.

		HTNPrecon			**FDBlind**		
Domain	Instances	#s	t	#b	#s	t	#b
Freecell	60	0	-	-	5	53.6	4344
Blocks	103	24	91	8118	18	4.9	1111
Rovers	20	20	0.6	1.2	6	44.3	6719
Logistics	80	80	2.9	44	10	0.5	140
Driverlog	20	0	-	-	7	11.9	1358
Zenotravel	20	4	25.6	4101	8	20.9	841
Miconic	150	150	0.66	0	55	50.5	8273
Satellite	20	7	0.59	1.2	6	221.9	7181
Depots	22	8	22.6	1867	4	18.7	2231

Table 4.2: Results in the IPC-2000 and IPC-2002 domains, with the total number of instances of each domain shown in brackets. For each solver we report number of solved instances (#s), average time in seconds (t) and average number of backtracks in thousands (#b) respectively

Table 4.2 shows the results for 9 IPC domains. HTNPrecon solves all the instances of LOGISTICS, ROVERS and MICONIC domains. In FREECELL domain, the generated invariant graphs proved to be too large for exploration by blind search. In addition each task is encoded with a large number of free parameters. Most of the free parameters in this domain are represented by auxiliary predicates, such as `number` and `successor`. This causes a high arity of the actions (implying that there are more possible bindings of objects to the argument of actions), and is used to impose an order for many different stacks. Our approach can solve instances of a simplified domain representation for FREECELL.

A similar encoding with auxiliary predicates affects HTNPrecon, in ZENO-

TRAVEL, where the `fuel-level` has to be guessed, along with the correct city, due to lack of constraints on the free parameters of the tasks. This leads to excessive backtracking, and in instances with large number of objects the plan is not found in the allotted time.

Another domain in which HTNPrecon does not work well, is DRIVERLOG. While the invariant graphs of this domain do not have a high branching factor, there are a multitude of ways to achieve each type of goal in this domain. The goal types are interdependent. Specifically the drivers have many ways to reach the location, and paths which are represented as links impose connectivity restrictions. They can either walk or drive to a specific location. If they attempt to drive a truck this generates a series of recursive task expansions attempting to first bring the truck to the driver. Therefore, the generated HTNs are ineffective in such domains.

Overall the comparison shows that our translation with precondition ordering is competitive with Fast Downward blind search. Compared to other approaches to generating HTNs, the advantage of our approach is that it does not need to learn from examples. The results show limitations of this approach, in domains with invariant graphs that have a high branching factor. Another limitation is the usage of auxiliary predicates, and high arity of actions. However, the benchmark domains with such features can possibly be rewritten in a more suitable way. For example, an action with high arity might be rewritten as several actions of lower arity.

4.7 Discussion

In this chapter we presented a domain-independent algorithm for generating HTNs. While other approaches learn from examples, ours can be seen as a compilation from PDDL to HTN. The algorithm takes as input the planning domain, a set of invariant candidates and one representative instance.

For the invariant graphs to be constructed, the representative instance needs to describe all relevant fluents. The set of invariant candidates is provided by Fast Downward algorithm for invariant synthesis. The output of the translation is an HTN domain. Additionally each original planning instance $\boldsymbol{p}$ is trivially converted to an HTN instance format. The representative instance is used to check if the invariant candidates hold and to determine the goal predicates. We show that the translation is sound. We also identify the class of planning domains for which the translation is complete.

We compare the basic version of the algorithm with precondition ordering against Fast Downward blind search. Although Fast Downward performs search over grounded and optimized version of invariant graphs, the HTN translation is comparable. While in some domains the planner using the HTN translation solves all instances, in some domains the HTNs are not constrained enough. Since the constructed HTNs are not constrained enough, blind search is not suitable for exploring invariant graphs with high branching factor. Actions with high arity, augment the search space as well, in relation to the number of objects in a given instance. The results further show, that in certain domains simplicity of the invariant graphs can be exploited by the HTN planner.

While the success of the algorithm is limited in some domains, there are still many potential benefits. The algorithm takes a fraction of a second to generate HTNs given a PDDL domain and a single example instance. The example instance does not need to be solved, and no plan traces are required. Since the algorithm is domain-independent it does not require any intervention and is easy to run. Therefore the resulting HTN could potentially be useful even in cases where it does not perform well right after the compilation, e.g. by extracting useful subtasks.

Chapter 5

OPTIMIZATIONS OF GENERATED HTNS

In this chapter, optimizations over the basic algorithm for generating the HTNs, are presented. A version of the basic algorithm with the precondition ordering optimization, is then compared with a fully optimized version, over a set of benchmark domains. Finally, we discuss the features of the benchmark domains and show properties of the HTN compilation.

5.1 Introduction

The basic algorithm for generating hierarchical task networks introduced in the previous chapter, produces HTNs which would require a lot of backtracking to achieve the goal, even for simple planning domains. This was the reason for introducing the precondition ordering, in the previous chapter. Large number of backtracks points to the fact that the search space has not been restricted enough in certain domains. The utility of the basic compilation for practical usage is therefore limited. The

JSHOP2 planner performs a simple depth-first search over the task network. Though the algorithm is domain-independent it relies on the model itself to utilize the added expressiveness to reduce the search space. For example, the usage of unconstrained free parameters in the decomposition methods, enlarges the search space by the total number of objects defined in a single instance. Therefore one way to optimize is to impose constraints on such parameters. Such constraints are usually tightly coupled with the specific domain, and can be thought of as a form of high-level features presented in Chapter 3. As an example, in the LOGISTICS domain, we could fly the airplane directly to the designated airport,as opposed to performing a search in the space of all airports.

Due to the simplicity of the JSHOP2 planner, some optimizations which work only for a particular type of the planning domain, can still be useful. The domain analysis techniques similar to those presented in the previous chapter, can be used to identify the type of the domain, in the preprocessing step of the compilation. Identifying the type of the domain, ensures that only the domains with features that allow for a certain optimization, will be optimized. In such cases we should ensure that if the optimization fails, JSHOP2 still explores the full search space. One way to optimize is to impose constraints on the tie-breaking mechanisms of the JSHOP2 planner, which are explained later in this chapter. Another way to optimize is to specify the features in the preconditions of the decomposition methods from the previous chapter, and create alternative versions of such methods. However, optimizations which impose a strict order on the goals and preconditions, do not allow for a complete search space exploration.

In the experiments we test our approach on planning benchmarks from the IPC and the instances used in experiments with HTN-MAKER (Hogg et al., 2008). To solve the instances we use the JSHOP2 planner (Nau et al., 2003), which performs blind search in the task space to compute a valid expansion. We show that the HTNs constructed by our algorithm solve all training and test set instances used to evaluate HTN-MAKER.

To illustrate the optimizations to our base algorithm, we will use the IPC-

2000 BLOCKS planning domain and this instance:

```
(define (problem blocks-example)
(:domain blocks)
(:objects A B C D - block)


(:init (on A B) (on C D)
      (clear A) (clear C) (ontable D)
      (ontable B))

(:goal (and (on A B) (on B C) (on C D)))
```

Some of the optimizations that are introduced in this work, would not fully reflect, on our running example in LOGISTICS domain. The reason is that in the LOGISTICS domain, the hierarchical structure is easier to identify in comparison with some other IPC benchmark domains. As shown in previous chapter, the basic compilation with precondition ordering, performs well in LOGISTICS domain. While the optimizations of our algorithm are general and are identified in a domain-independent manner, some are introduced only for the domains with certain features, which can be identified by analyzing the example instance.

5.2 Goal Ordering

Just as for preconditions, achieving goals in any order results in significant backtracking. To order the goals we implement an algorithm similar to the one for ordering preconditions. While the ordered preconditions are coded into the HTN, the goals are different for each instance of the domain. Since HTNs are instance-independent, our approach is to define new tasks that compute a goal ordering as a preprocessing step.

To accomplish this, we first order the goals of the representative instance passed to the algorithm. We run the precondition ordering algorithm on

the set of goal predicates $\mathcal{P}_G \subseteq \mathcal{P}$, i.e. predicates whose associated fluents appear in the goal. For each $p \in \mathcal{P}_G$ and each pair of fluents $p[x]$ and $p[y]$, we check if $p[y]$ is achievable when $p[x]$ persists (i.e. we are not allowed to delete $p[x]$). Each invariant graph that contains p is partially grounded on $p[x]$, while the preconditions of actions that directly achieve p are grounded on $p[y]$. Given an ordering of the predicates in $\mathcal{P}_G$, we then order the set of fluents of each predicate $p \in \mathcal{P}_G$, again using the precondition ordering algorithm. To do so, the invariant graphs need to be partially grounded, on each pair of fluents to be ordered. If this partial grounding violates the invariant, $p[y]$ should be ordered before $p[x]$. Once the invariant is invalidated by partial grounding, the algorithm stores the indices of the parameters of p that invalidated the invariant.

Since goal ordering would not fully reflect in the LOGISTICS domain, to the same extent as in the BLOCKS domain, we will consider the `on` goal predicate from the BLOCKS domain. Since the only goal predicate is `on`, $\mathcal{P}_G = \{\texttt{on}\}$, so the method for `solve` decomposes to `achieve-on` to achieve goals in this domain. Figure 5.1 shows one of the invariant graphs in BLOCKS that contains `on`. To define the method for `order` we test each pair of goal fluents to establish an order among them. Consider two goal fluents $\texttt{on}[\texttt{a}, \texttt{b}]$ and $\texttt{on}[\texttt{b}, \texttt{c}]$. If we fix the fluent $\texttt{on}[\texttt{a}, \texttt{b}]$ and attempt to achieve $\texttt{on}[\texttt{b}, \texttt{c}]$, the only operator $\texttt{stack}[\texttt{b}, \texttt{c}]$ that directly achieves $\texttt{on}[\texttt{b}, \texttt{c}]$ has precondition $\texttt{holding}[\texttt{b}]$, which violates the invariant since $\texttt{on}[\texttt{a}, \texttt{b}]$ is assumed to hold. Thus $\texttt{on}[\texttt{b}, \texttt{c}]$ should be ordered before $\texttt{on}[\texttt{a}, \texttt{b}]$. We can generalize this knowledge and derive a rule that whenever two goal fluents of type `on` have the same object as the first and second parameter, respectively, the former should be ordered before the latter.

This general rule is coded into the methods which decompose the `order` task. Just as in the previous chapter, all of the goal fluents from the instance are encoded in the initial state, by instantiating the `goal-`p predicate to denote fluents which need to be ordered. The ordering is performed by decomposing the `order` task. As the goals are ordered, the `ogoal-`p fluent is added for each goal which has been ordered in the HTN

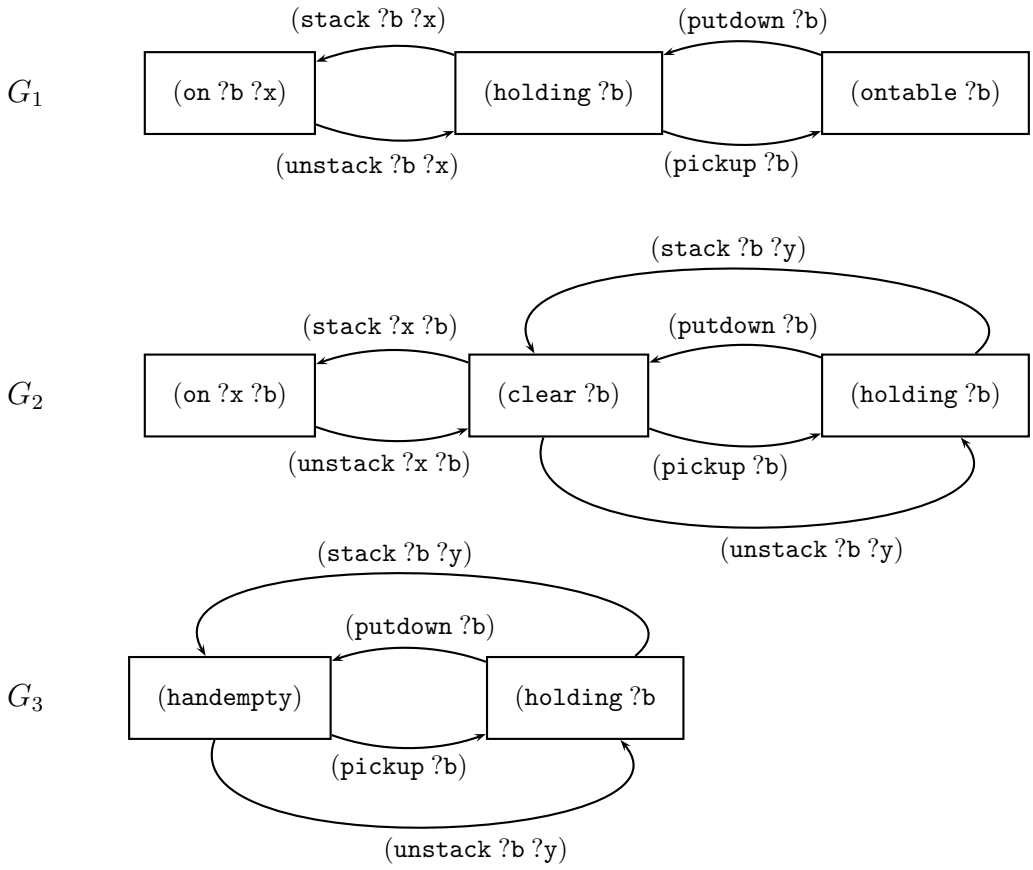


Figure 5.1: Invariant graphs (G1, G2 and G3) in the BLOCKS domain.

instance. As before, the `solve` task then attempts to achieve each fluent in order, starting from 0 each time, and restarting from 0 after a goal fluent has been achieved.

5.3 Ordering the invariant graphs

Invariant graphs are generated based on invariant candidates which are in turn generated by Fast Downward. Since the methods are generated based on the invariant graphs, the ordering is thus imposed by the order in which candidate invariants are evaluated at the time invariant graphs are created. The methods that decompose these tasks are generated in such an order as well. Since JSHOP2 uses depth-first search to expand methods which can bind with certain parameters it has to impose an order on how the candidate methods are evaluated. In case of SHOP2 this is done by the order they appear in the file which describes the hierarchies. Creating a full or partial order of methods can thus be seen as a form of search preference. Forming such orderings can reduce the number of backtracks significantly, if methods that reduce the search effort can be identified and ordered first.

In case of HTNs generated by invariant analysis, it could be possible to achieve certain predicate in more than one invariant graph. This can be seen in the example of BLOCKS invariant graphs in Figure 5.1. For each predicate, our algorithm generates a wrapper method, which then selects the graph in which the fluent will be achieved. Therefore, identifying in which invariant graph it is easier to achieve a certain predicate, would lead to less search at the time of finding the proper decompositions. Ordering these methods would simply impose a preference at the time of achieving a predicate p, in a certain invariant graph. Should the search fail to decompose the compound task in such a way to achieve p in the preferred invariant graph it would proceed to attempt decomposition of the remaining tasks, which are generated based on other invariant graphs.

Looking at the BLOCKS invariant graphs in Figure 5.1, the `on` predicate, can be achieved either in G_1 or G_2. While G_2 is more complex, as it has more edges that would require additional methods to be generated, it is easier to decompose the methods that achieve the `on` predicate based on G_2, than those based on G_1. This means that we simply need to order the `achieve-on` methods which decompose to `achieve-on2`, before the

`achieve-on` methods that decompose to `achieve-on1`.

The reason for this is the index of the bound object in the goal predicate. For example in the BLOCKS domain, in Figure 5.1 the bound object in any invariant graph is marked as $?b$. In graph G_1, in the parameters of the `on` predicate, $?b$ is the first object, while in G_2 it is second. The methods for achieving it, in these two graphs would thus be different. Intuitively these methods would encode two different strategies, for achieving a goal `on`$[\texttt{a}, \texttt{b}]$ for example. One would be to place the block `a` on an arbitrary block, and repeat the procedure, until it is placed on the correct block `b`. Another strategy is to place arbitrary blocks atop the block `b`, until block `a` is placed on top. Using the goal ordering strategy, we can infer, that once an arbitrary block is placed on top of block `b`, no further blocks can be placed directly on top of that block. Therefore if the search starts within G_1 it is possible that the search will backtrack, due to the fact that the block `a` is held by the gripper over the block `b`, but that block could initially be under a number of arbitrary blocks, which should then be cleared first. On the other hand if we start the search in G_2, `b` would already be the bound block, and thus arbitrary blocks can be placed on top of it only after `b` becomes clear.

The goal ordering strategy gives information not only on predicates for which invariant graphs can be ordered, but also on indices of the parameters in that predicate, which become blocked by the object placed in the relation. This can be generalized to any n-ary predicate $\texttt{p}[x_1 \cdots x_n]$, such that $n > 1$, for which there is a goal ordering as specified in the goal ordering section. The goal ordering strategy gives as output the set of indices $I_n = \{i_1, \ldots, i_k\}$, such that $1 \leq k < n$. These indices represent the objects which are unreachable, without reversing the effects of the action that placed them in relation p, with indices in I_n. Intuitively, we assume that the identified indices in I_n functionally determine the indices which are not in I_n.

For each predicate p which has been ordered in the goal ordering step, the invariant ordering identifies all the invariant graphs where the predicate p can be achieved. The graphs are then identified, so that no bound objects

for p in the graph appears in the set I_n. The achieve-p methods which perform search in the identified graphs, are placed before other achieve-p methods in the output HTN domain. This will ensure that these methods are explored first during the search, as the achieve-p methods select the invariant graphs in which the search will be performed.

5.4 Sorting the bindings of the free variables

Free variables in our HTN task decomposition correspond to free variables in the nodes of invariant graphs. The blind search over the HTN, originally does not impose any order on the binding of free variables that appear in the methods. This can lead to traversal of the wrong edge in the graph, and possibly to backtracking. The reason for this is that the invariant graphs are considered only in the lifted variant during the translation to HTN. However, during the actual search, a grounded version of the invariant graph needs to be traversed to achieve each goal fluent, as illustrated in Figure 4.3 in the previous chapter. In that case, intuitively, the number of possible bindings of the free parameters becomes a multiplier for the edges in the invariant graph, given a particular planning instance. Any values that satisfy the preconditions of a method can be used to ground methods with free parameters during task decomposition. This means that decompositions of do-p-a-i methods do not immediately aim for the target node, even if this node is just one step away. We exploit the ability of SHOP2 to order the bindings of free variables by imposing an order that first attempts to traverse an edge to the target node. If this is not possible, blind search later explores all other possibilities.

One example in the BLOCKS domain is the achieve-on1[?b] method that decomposes directly to the primitive task stack[?b, ?x]. It is generated to traverse the edge between holding[?b] and on[?b, ?x] in the invariant graph G_1 in Figure 5.1. The ?x of the on predicate is the free parameter which would be arbitrary chosen during search by JSHOP2. This means that the bound block will be placed on arbitrary blocks, until the correct

block, specified in the goal, is reached. However, if the goal is to achieve on[a, b], then the bound parameter is block a and the parameters should be sorted in such a way, to place b first in the list of objects that can be assigned to the free parameter ?x.

5.5 Experimental Results

We ran our algorithm on all instances in the training and test set of HTN-MAKER (LOGISTICS, SATELLITE and BLOCKS). The tests of the algorithm were also performed in other STRIPS planning domains from the IPC. The experiments were performed with two versions of our algorithm.

The first version, HTNPrecon, achieves the goals in the order they appear in the PDDL definition. The second version, HTNGoal, implements our goal ordering, invariant graph ordering and parameter sorting strategies, in addition to precondition ordering. In all experiments we used JSHOP2 (the Java implementation of SHOP2), which uses blind search to compute a valid expansion. We used a memory limit of 4GB and a timeout of 1,800 seconds.

Comparison with HTN-MAKER

We compare to the latest results of HTN-MAKER (WeakS), which uses a C++ implementation of SHOP2 and one hour of CPU time to solve the instances (Hogg et al., 2016). However, the instances from that paper are not publicly available, so we could not compare directly, and we just show their reported coverage results for reference.

Table 2 shows the coverage results over HTN-MAKER's experiment files. We tested HTN-MAKER's output domains from the fifth and final trial (Hogg et al., 2008) and we ran it over the test set only, while we ran our algorithm over both the test and training sets of the experiment instances.

	HTNPrecon	HTNGoal	HTN-MAKER	WeakS
LOGISTICS	93%	100%	100%	93,6%
SATELLITE	100%	100%	92%	100%
BLOCKS	0,36%	100%	63,5%	99%
ROVERS	100 % *	100% *	-	100%
ZENOTRAVEL	20% *	100% *	-	99,8%

Table 5.1: Coverage of HTN-MAKER's experiment instances, scores marked with * are scores over IPC instances

To achieve the reported results, HTN-MAKER needed 420 training instances in BLOCKS and 75 training instances in LOGISTICS and SATELLITE, while our algorithm used the domain and a single example instance with no need for solving the instance or annotating the plan beforehand. For ROVERS and ZENOTRAVEL we report coverage over the IPC-2002 instances for HTNPrecon and HTNGoal. The improvement of HTNGoal over HTNPrecon is due to goal ordering in BLOCKS (since blocks have to be stacked in a specific order) and parameter sorting in LOGISTICS and ZENOTRAVEL (since airplanes can fly directly to the target destination). The instances used in HTN-MAKER's experiments are generated by a random generator with very few objects (e.g. maximum five blocks), while our HTNs can be used to solve much larger instances. For example, in BLOCKS, IPC benchmark instances include up to 50 blocks.

Comparison of HTNPrecon with HTNGoal

We ran both the HTNPrecon and HTNGoal versions of our algorithm on other benchmark instances from IPC-2000 and IPC-2002. The results from these experiments appear in Table 5.2.

HTNGoal achieves full coverage over the IPC-2000 and IPC-2002 benchmark instances of BLOCKS, SATELLITE, ROVERS, LOGISTICS, ZENOTRAVEL, DEPOTS and MICONIC. The improvement in average number of backtracks is most clearly visible in the BLOCKS domain, and is due

		HTNPrecon			**HTNGoal**		
Domain	Instances	#s	t	#b	#s	t	#b
Freecell	60	0	-	-	0	-	-
Blocks	103	24	91	8118	103	0.6	0
Rovers	20	20	0.6	1.2	20	0.5	1.1
Logistics	80	80	2.9	44	80	2.4	23.7
Driverlog	20	0	-	-	3	0.4	1.3
Zenotravel	20	4	25.6	4101	20	0.5	0.3
Miconic	150	150	0.66	0	150	0.63	0
Satellite	20	7	0.59	1.2	20	0.37	0.04
Depots	22	8	22.6	1867	22	88.4	0

Table 5.2: Results in the IPC-2000 and IPC-2002 domains, with the total number of instances of each domain shown in brackets. For each solver we report number of solved instances (#s), average time in seconds (t) and average number of backtracks in thousands (#b) respectively

to our goal ordering strategy. It also allowed for an increase in number of instances solved in the DEPOTS domain, as these two domains are the most sensitive to goal ordering. The combination of goal ordering and free variable sorting shows an increase in performance in other domains as well. This allows HTNGoal to solve all instances in many domains, with very few backtracks. These domains however tend to be the ones with a lower branching factor in the invariant graphs.

The results of our approach are comparable to that of the algorithms that learn from examples. In some domains generated by HTNGoal the planner is able to find a solution backtrack-free for any instance appearing in the benchmark. In case of MICONIC it is very easy to create a backtrack-free policy and even HTNPrecon generates one, however in domains like BLOCKS and DEPOTS some optimizations are needed. Namely ordering of the invariant graphs helps in these cases, due to the ability to identify invariant graphs in which the search will not result in excessive backtracking. The sorting of the invariant graphs is however limited to those

domains where the goal-ordering strategy can be inferred, and requires a certain fluent to be achievable in more than one invariant graph. These domains are usually "stacking" domains.

The unsolved instances are not due to failed decomposition, rather the alloted time was insufficient to perform search in an insufficiency constrained HTN. While the results of our approach are comparable to those of HTN-MAKER, in some domains the generated HTNs do not perform well due to excessive backtracking (e.g. we do not solve any instances of the IPC-2002 FREECELL domain, which is more puzzle-like and therefore harder to serialize as our HTNs do by achieving one goal fluent at a time).

To test our hypothesis, that the translation does not solve any instances of the FREECELL domain, due to high branching factor, and high arity of the actions, we constructed a different PDDL encoding of FREECELL. This version has actions similar to the BLOCKS domain with limited table surface, and does not rely on auxiliary predicates to define stacks of cards. In testing over the converted IPC instances, and under same conditions, the JSHOP2 planner solves 25 out of 60 instances. This result is comparable to the result of the ICARUS planner, which bootstraps the learning by providing hand-tailored concepts. While the 25 instances are solved in less than a second on average, in all but one instance with more than 6 cards, the planner exceeds the allotted time due to backtracking. The reason is that the resulting policy of the invariant graph search, resembles a greedy policy, which sends all blocking cards of a single column to the free cells, and thus exhausts the resource fast. The blind search then explores all possible options, which is not computationally feasible.

5.6 Discussion

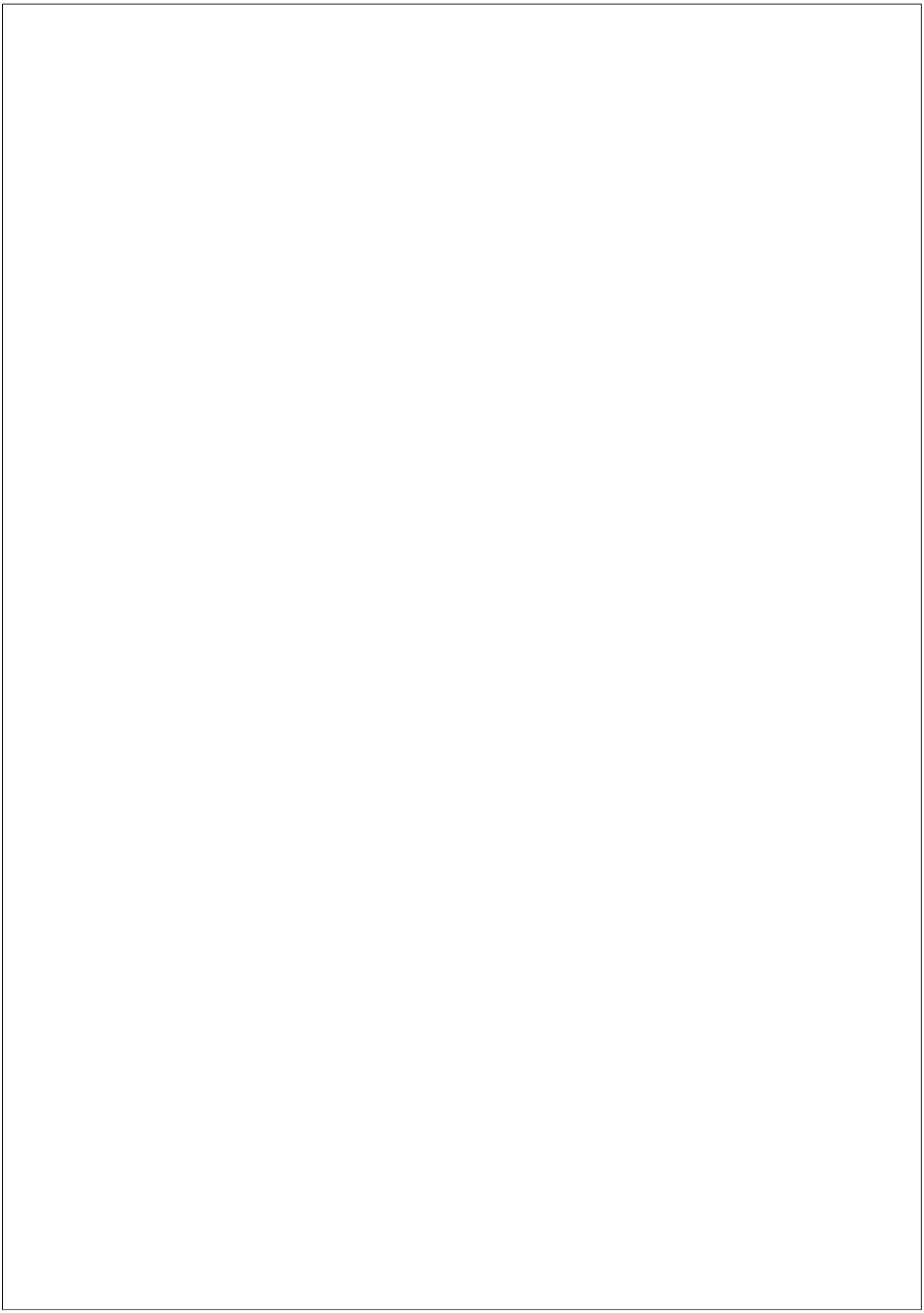
In this chapter we presented the optimizations over the basic algorithm for compilation from PDDL to HTN. We show that the fully optimized ver-

sion of the algorithm, is competitive with the state-of-the-art algorithm for HTN synthesis HTN-MAKER. Moreover, the results show that in some domains the instances can be solved with no, or very few backtracks.

While HTN-MAKER and other approaches induce the HTN representation from a given set of labeled examples, our algorithm takes as input a planning domain and an example instance. In addition to invariant candidate checks, for the fully optimized HTN generator, the example instance is used for goal ordering optimization. While the algorithm remains domain-independent, we show that the compilation can utilize the preprocessing step to classify the planning domain and generate HTN which are optimized for a certain type of the planning domain. The translation remains sound, as the `solve` methods and the operators, remain unaffected by the optimizations introduced in this chapter.

Part IV

Related Work



Chapter 6

RELATED WORK

Approaches to generalized planning can be divided into *inductive* and *deductive*. While the inductive approach is to induce a generalized plan from a set of examples, deductive approach is to generate a generalized plan directly from a given model of actions. Moreover generalized plans can be represented as decision lists, finite-state machines, programs, etc. Some of the representations encode a policy which represents a unique solution for all the planning instances in a domain. Other representations encode a domain-specific knowledge which reduces the search space, allowing for efficient solutions of large-scale instances.

6.1 High-Level State Features

Our approach for learning high-level state features (and classifiers) is inspired by *version space learning* (Mitchell, 1982). The hypothesis to learn consists of logic clauses and examples are logic facts that restrict the hypothesis forcing it to be consistent with the examples. Inductive Logic Programming (ILP) (Muggleton, 1999) also intersects Machine Learning (ML) and Logic Programming to generate hypotheses from examples.

ILP has traditionally been considered a binary classification task but, in recent years, it covers the whole spectrum of ML such as regression, clustering and association analysis. The main contribution of our approach with respect to version space learning and ILP is the use of a classical planner to build and validate the learned hypotheses.

Previous work on computing generalized plans in the form of generalized policies already attempted to automatically generate higher-level state representations (Khardon, 1999). Two examples are learning generalized policies from solved instances using description logic (Martín and Geffner, 2004) and taxonomic syntax (Yoon et al., 2008) to represent and reason about classes of objects. In these works, planning and learning were clearly separate phases producing noisy learning examples in many cases. In contrast, our approach tightly integrates planning and learning.

Generating high-level state features for generalized planning is also related to previous work on First Order MDPs (Boutilier et al., 2001; Gretton and Thiébaux, 2004). These works adapt traditional dynamic programming algorithms to the symbolic setting and automatically generate first-order representations of the value function with first-order regression. The main contribution of our approach with respect to this research line is that we follow a compilation approach to generate useful state abstractions with off-the-shelf planners.

6.2 Generating HTNs

A basic, and the only compilation from STRIPS to HTN that we know of, was first defined by Erol et al. (1994). This compilation constructs primitive tasks for each STRIPS operator and a single compound task. While this compilation is both sound and complete, it is used only for the purposes of theoretical analysis and it does not impose any restrictions on the task network that would make it solve problems more effectively.

The other approaches for generating hierarchical task models are learning from plan traces. However these approaches not only need to learn from examples but also rely on some given task-subtask decompositions, annotated plans, or manually added concepts. Some of them use different methods to encode the hierarchal structure. For example ICARUS (Langley and Choi, 2006) uses teleoreactive logic programs to represent the hierarchies. It specifies a set of primitive and compound *skills* to represent decompositions and *concepts* represent the state features as Horn clauses. It is able to incrementally learn new skills, using means-ends analysis over the given set of examples. It is also able to learn new skills directly from problem solving, i.e. when the solver cannot find a skill to achieve a certain goal. However ICARUS depends heavily on compound skills and concepts provided before the learning process starts.

Another algorithm that learns HTNs from plan traces is HTN-MAKER (Hogg et al., 2008). This algorithm can be used to solve "classically-partitionable" problems. These problems cannot be expressed in classical-planning without making extensions which would allow for marking tasks rather than goals in the classical sense. It uses a set of annotated plans to learn the hierarchy by using means-ends analysis. In contrast, our algorithm requires less semantic information, and although we could use the representative instance (or multiple instances) to generate solution plans for learning, these plans would still have to be annotated to be used by HTN-MAKER.

Our approach to generating HTNs from the model and a single planning instance and using them to solve larger instances of the same planning domain can be viewed as a form of generalized planning, which has received a lot of recent attention, most notably in the form of the learning track of the IPC. One popular approach to generalized planning is to identify macros (Botea et al., 2005; MacGlashan, 2010; Muise et al., 2009; Newton et al., 2007), i.e. sequences of operators that frequently appear in the solutions to example instances. Once identified, such macros can then be inserted into the action space of larger instances in order to speed up search. However, even though macros can be parameterized, they do not

offer the same flexibility as HTNs in terms of representing a solution to all instances of a planning domain.

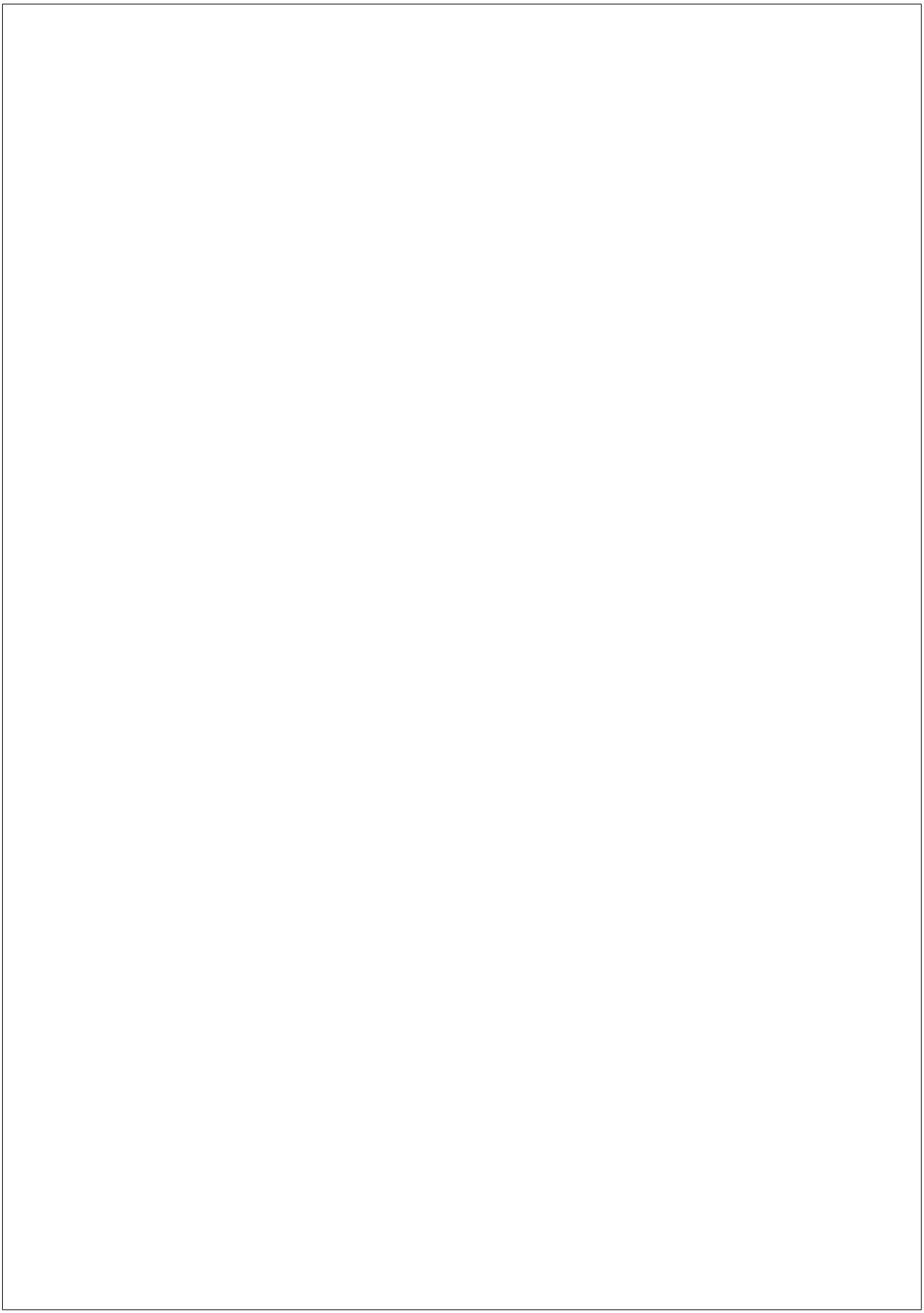
Another approach to generalized planning is to learn reactive policies for planning domains (Khardon, 1999; Levine and Humphreys, 2003; Martin and Geffner, 2000; Yoon et al., 2008). A third approach is to learn domain-specific knowledge in order to improve heuristic estimates computed during search (de la Rosa et al., 2011; Yoon et al., 2008). In contrast to most of these techniques, which are *inductive*, ours is a *generative* approach, as we construct HTNs directly by analyzing the domain model.

Achieving fluents by traversing the edges of domain transition graphs is the strategy used by DTGPlan (Chen et al., 2008) and similar algorithms. There also exist other inference techniques that can solve many individual instances backtrack-free (Lipovetzky and Geffner, 2009, 2011). The novelty of our approach compared to previous work is the ability to do this in an instance-independent way.

Our work is also related to other approaches to hierarchical planning (Holte et al., 1996; Marthi et al., 2008; Elkawkagy et al., 2012), with the difference that we generate the hierarchies automatically.

Part V

Conclusions and Future Work



Chapter 7

CONCLUSIONS

In this chapter we summarize the contributions of this thesis and discuss the most promising avenues for future work. The main body of this work focuses on planning programs and HTNs as representations of generalized plans. We presented novel algorithms for generating both HTNs and an extended version of planning programs. We show the synthesis of planning programs with conjunctive queries, which requires a set of planning instances as an input. We also present an approach which generates an HTN, directly from the planning domain and one representative instance.

7.1 Contributions

In this section we list main contributions of this work:

1. The extension of planning programs, which allows for usage of high-level state features in the form of conjunctive queries. Moreover the algorithm couples the generation of the features with the generation of the program.

2. The noise-free classifiers represented by an extension of planning programs. While not competitive with ML classifiers, the high-level state features with planning programs bring a novel type of domain and possible benchmark to classical planning.

3. The domain-independent algorithm for generating hierarchical task networks based on invariance analysis. While the basic compilation from PDDL to HTN existed before (Erol et al., 1994), it is basically identical to the original planning problem, and is used for theoretical purposes only. In this work we present a compilation which captures the abstractions, which can effectively reduce the search space in many planning domains.

4. A sound framework for HTN planning. We show that the HTN generator algorithm is sound in all presented variants. The algorithm also provides a sound framework, as certain types of decomposition methods can be removed or introduced without affecting the soundness of the plan.

5. The optimized version of generated HTNs. The introduced optimizations are domain-independent. Some of the presented optimizations work by pruning the search space (i.e. goal ordering) and have possible implications on completeness, while others are used in form of the heuristic information (i.e. invariant graph ordering). We show that the optimizations improve coverage and reduce backtracking over the IPC domains, and make the translation competitive with state-of-the-art HTN learning algorithms.

7.2 Future work

In this section we explore possible research directions that could arise from the work presented in this thesis.

7.2.1 Planning Programs with High-Level State Features

The extension of basic version of planning programs presented in Chapter 3, only allows for conjunctive queries to be evaluated in the `goto` statements of the planning program. Such an extension is not enough for expressing more complex features, which are needed for certain domains. For example, in the BLOCKS domain, it would be helpful to define the `above`, a feature that detects whether a block appears above another block in a given tower of block. This feature cannot be expressed as a conjunctive query, due to its recursive definition. However, PDDL supports derived predicates, from version 2.2 onwards.

Derived predicates significantly enhance the expressive power of PDDL (Thiébaux et al., 2005). In the example of BLOCKS, we can express the `above` feature as a derived predicate:

```
(:derived (above ?x ?y)
        (or (on ?x ?y) (exists (?z)
            (and (on ?x ?z) (above ?z ?y)))
        )
)
```

The features in the form of conjunctive queries are computed in tight coupling with the computation of the planning program, since the evaluation and generation of such features is computationally feasible. The reason is that the conjunctive query is subdivided into slots which are then programmed by assigning atoms to the slot. Such an approach would not be possible in the case of general derived predicates, due to the different structure.

Derived predicates can be generated directly from a PDDL domain (Miura and Fukunaga, 2017). Miura and Fukunaga show that useful derived predicates, can be extracted from a number of IPC benchmark domains. A similar approach can be used to extract useful features in a preprocessing step of the planning program synthesis. Subsequently the useful features

can be selected during the programming phase, and programmed as a condition of the `goto` statement.

7.2.2 Generating Hierarchical Task Networks

The basic translation of PDDL to HTN, in practice relies on a number of domain-independent optimizations. The reason is that the exploration of unoptimized invariant graphs, using blind search, is computationally infeasible. One approach, which has been explored to an extent in this work, is to generate the constraints, which can be imposed on the generated hierarchical task network. The results show that such optimizations presented in Chapter 5, can significantly and consistently reduce the search space in many planning domains.

The ordering of the invariant graphs optimization, shows that even simple changes in the task network can lead to large improvements, in the coverage over the IPC benchmark domains. Further, the invariant graphs used for the translation are not optimized in any way. One simple approach to extending the translation, would be to prune the invariant graphs. Some edges or nodes of an invariant graph could be removed in order to reduce the search space Such pruning would have to be consistent, so that all goal fluents have to be reachable in some invariant graph. This can be validated on several example instances. The decomposition methods would have to be adjusted, to ensure that the correct invariant graphs are selected. This optimization could work similarly to optimization performed by Fast Downward, which approximates a set cover over mutex groups.

The generated HTNs can be optimized by adding new decomposition methods, with additional preconditions, which can in turn represent certain features in a single planning domain. Such features can be learned from a set of solved planning instances or from the analysis of lifted invariant graphs. The decomposition methods with more specific preconditions, would then be ordered before the more general ones, guaranteeing that no loss of generality occurs. Such decomposition methods could di-

rectly encode a strategy for traversing the lifted invariant graph from a certain node to the target node. This approach would still use the deductive approach to generate the basic HTN, and then refine it using a set of examples.

For example in the LOGISTICS domain, the decomposition method (`achieve-at-1` $?p$ $?l$), where $?p$ is a package, and $?l$ is the target location, can be encoded in a way that transports the packages which are not in the target city, directly to the airport. For brevity we omit the type predicates in the following example:

```
(:method (achieve ?p ?l)
        ( (not (at ?p ?l)) (in-city ?l ?c)
          (at ?p ?l1) (not (in-city ?l1 ?c))
          (in-city ?l1 ?c1) (at ?t ?l2)
          (in-city ?l2 ?c1)
        )
        (       (!drive-truck ?t ?l2 ?l1 ?c1)
                (!load-truck ?p ?t ?l1)
                (!drive-truck ?t ?l1 ?ap ?c1)
                (!unload-truck ?p ?t ?ap)
        )
)
```

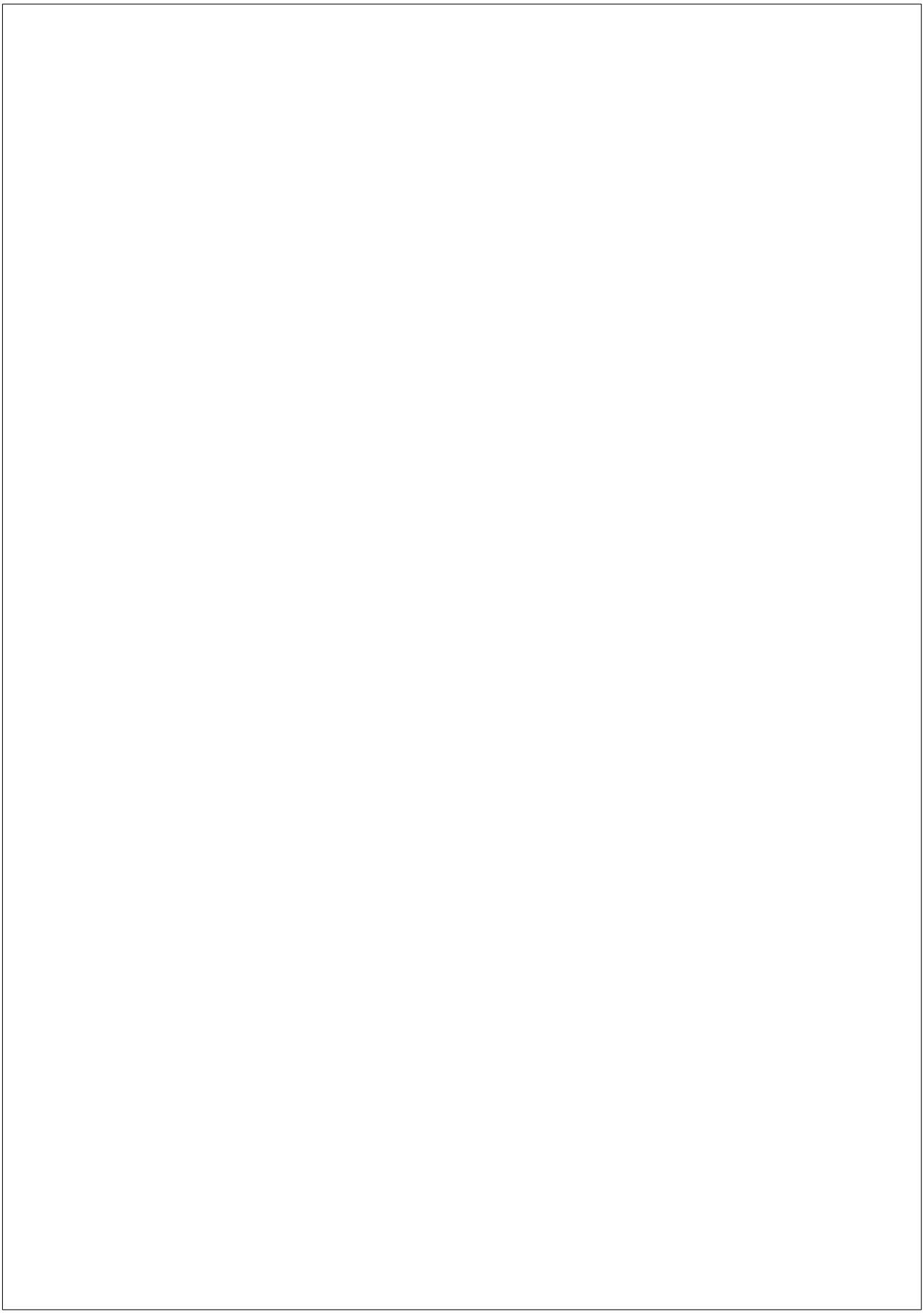
Such a method could be decomposed only in a specific case, and would override the decompositions of `do-`p-a-i types of tasks.

The SHOP2 planner relies on the domain encoding to narrow the search space, and uses blind search. Currently there are few domain-independent heuristics, used in HTN planning. Recently a domain-independent admissible heuristic, based on task decomposition graph (TDG), has been proposed (Bercher et al., 2017). An interesting application, of the invariant graph structures, used for the compilation presented in Chapter 4, would be to extract the heuristics for HTN planners. Such heuristic information can be used to guide the task-subtask decomposition. This should at least be possible for the HTNs which are derived from a PDDL rep-

resentation. As explained in Chapter 4, even though the generated HTN is based on the invariant graphs, no graph is grounded at the beginning of the search. However, as the search progresses, it is essentially equivalent to exploration of the grounded invariant graph. The fully grounded invariant graphs, could provide heuristic, not only for selecting the invariant graph in which the fluent should be achieved, but also for selecting the parameters of the do-p-a-i methods in our HTN translation.

Part VI

Appendix



Bibliography

Bäckström, C., Jonsson, A., and Jonsson, P. (2014). Automaton plans. *Journal of Artificial Intelligence Research*, 51:255–291.

Bercher, P., Behnke, G., Höller, D., and Biundo, S. (2017). An admissible htn planning heuristic.

Bonet, B., Palacios, H., and Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24:581–621.

Boutilier, C., Reiter, R., and Price, B. (2001). Symbolic dynamic programming for first-order mdps. In *IJCAI*, volume 1, pages 690–700.

Chandra, A. and Merlin, P. (1977). Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM Symposium on Theory of Computing*.

Chen, Y., Huang, R., and Zhang, W. (2008). Fast Planning by Search in Domain Transition Graphs. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 886–891.

Cresswell, S. and Coddington, A. M. (2004). Compilation of ltl goal formulas into pddl. In *ECAI*, pages 985–986.

de la Rosa, T., Jiménez, S., Fuentetaja, R., and Borrajo, D. (2011). Scaling up Heuristic Planning with Relational Decision Trees. *Journal of Artificial Intelligence Research*, 40:767–813.

Edelkamp, S. and Hoffmann, J. (2004). Pddl2.2: The language for the classical part of the 4th international planning competition. Technical report, Technical Report 195, University of Freiburg.

Elkawkagy, M., Bercher, P., Schattenberg, B., and Biundo, S. (2012). Improving Hierarchical Planning Performance by the Use of Landmarks. In *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI'12)*.

Erol, K., Hendler, J., and Nau, D. (1994). HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 1123–1128.

Fern, A., Khardon, R., and Tadepalli, P. (2011). The first learning track of the international planning competition. *Machine Learning*, 84(1-2):81–107.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

Fox, M. and Long, D. (2002). Pddl+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34.

Geier, T. and Bercher, P. (2011). On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1955–1961.

González-Ferrer, A., Fernández-Olivares, J., and Castillo, L. (2013). From Business Process Models to Hierarchical Task Network Planning Domains. *Knowledge Engineering Review*, 28(2):175–193.

Gretton, C. and Thiébaux, S. (2004). Exploiting first-order regression in inductive policy selection. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 217–225. AUAI Press.

Helmert, M. (2004). A planning heuristic based on causal graph analysis. In *ICAPS*, volume 4, pages 161–170.

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535.

Hoffmann, J. and Edelkamp, S. (2005). The deterministic part of ipc-4: An overview. *Journal of Artificial Intelligence Research*, pages 519–579.

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 950–956.

Hogg, C., Muñoz-Avila, H., and Kuter, U. (2016). Learning hierarchical task models from input traces. *Computational Intelligence*, 32(1):3–48.

Holte, R., Perez, M., Zimmer, R., and MacDonald, A. (1996). Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pages 530–535.

Hu, Y. and De Giacomo, G. (2011). Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 918.

Hu, Y. and De Giacomo, G. (2013). A generic technique for synthesizing bounded finite-state controllers. In *International Conference on Automated Planning and Scheduling*.

Hu, Y. and Levesque, H. J. (2011). A correctness result for reasoning about one-dimensional planning problems. In *International Joint Conference on Artificial Intelligence*, pages 2638–2643.

Ivankovic, F. and Haslum, P. (2015). Optimal planning with axioms. In *International Joint Conference on Artificial Intelligence*, pages 1580–1586. AAAI Press.

Jiménez, S., De la Rosa, T., Fernández, S., Fernández, F., and Borrajo, D. (2012). A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467.

Jiménez, S. and Jonsson, A. (2015). Computing Plans with Control Flow and Procedures Using a Classical Planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, pages 62–69.

Jonsson, A., Jonsson, P., and Lööw, T. (2013). When acyclicity is not enough: Limitations of the causal graph. In *ICAPS*.

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

Langley, P. and Choi, D. (2006). Learning recursive control programs from problem solving. *The Journal of Machine Learning Research*, 7:493–518.

Levine, J. and Humphreys, D. (2003). Learning Action Strategies for Planning Domains Using Genetic Programming. In *EvoWorkshops,* volume 2611 of *Lecture Notes in Computer Science*, pages 684–695.

Lipovetzky, N. and Geffner, H. (2009). Inference and Decomposition in Planning Using Causal Consistent Chains. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

Lipovetzky, N. and Geffner, H. (2011). Searching for Plans with Carefully Designed Probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*.

MacGlashan, J. (2010). Hierarchical Skill Learning for High-Level Planning. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10)*.

Marthi, B., Russell, S., and Wolfe, J. (2008). Angelic Hierarchical Planning: Optimal and Online Algorithms. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, pages 222–231.

Martin, M. and Geffner, H. (2000). Learning Generalized Policies in Planning Using Concept Languages. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 667–677.

Martín, M. and Geffner, H. (2004). Learning generalized policies from planning examples using concept languages. *Appl. Intell*, 20:9–19.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl-the planning domain definition language.

Menif, A., Guettier, C., and Cazenave, T. (2013). Planning and Execution Control Architecture for Infantry Serious Gaming. In *Proceedings of the 3rd International Planning in Games Workshop (PG'13)*, pages 31–34.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (2013). *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

Miller, C., Goldman, R., Funk, H., Wu, P., and Pate, B. (2004). A Playbook Approach to Variable Autonomy Control: Application for Control of Multiple, Heterogeneous Unmanned Air Vehicles. In *Annual Meeting of the American Helicopter Society*.

Mitchell, T. M. (1982). Generalization as search. *Artificial intelligence*, 18(2):203–226.

Miura, S. and Fukunaga, A. (2017). Automatically extracting axioms in classical planning. In *AAAI*, pages 4973–4974.

Muggleton, S. (1999). Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114(1):283–296.

Muise, C., McIlraith, S., Baier, J., and Reimer, M. (2009). Exploiting N-Gram Analysis to Predict Operator Sequences. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*.

Munoz-Avila, H., Aha, D., Breslow, L., and Nau, D. (1999). HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 870–875.

Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404.

Newton, M. H., Levine, J., Fox, M., and Long, D. (2007). Learning Macro-Actions for Arbitrary Planners and Domains. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pages 256–263.

Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35:623–675.

Richter, S. and Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

Sánchez-Garzón, I., Fernández-Olivares, J., and Castillo, L. (2013). An Approach for Representing and Managing Medical Exceptions in Care

Pathways Based on Temporal Hierarchical Planning Techniques. In *Process Support and Knowledge Representation in Health Care (ProHealth'12), Lecture Notes in Computer Science 7738*, pages 168–182.

Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, page 32.

Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2016). Generalized planning with procedural domain control knowledge. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Srivastava, S. (2011). Foundations and applications of generalized planning. *AI Communications*, 24(4):349–351.

Srivastava, S., Immerman, N., Zilberstein, S., and Zhang, T. (2011). Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*, pages 226–233.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of pddl axioms. *Artificial Intelligence*, 168(1-2):38–69.

van der Sterren, W. (2009). Multi-Unit Planning with HTN and A*. In *AIGameDev Paris Game AI Conference*.

Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: The prodigy architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120.

Winner, E. and Veloso, M. (2003). Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*, pages 800–807.

Wu, D., Parsia, B., Sirin, E., Hendler, J., and Nau, D. (2003). Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings*

of the 2nd International Semantic Web Conference (ISWC'03), pages 195–210.

Yoon, S., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718.

Younes, H. L. and Littman, M. L. (2004). Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*.

Zhuo, H., Hu, D., Hogg, C., Yang, Q., and Munoz-Avila, H. (2009). Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1804–1809.