PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF COMPUTER SCIENCE
POSTGRADUATE PROGRAMME IN COMPUTER SCIENCE

USING OPERATOR PATTERNS TO INFER DOMAIN-KNOWLEDGE IN PLANNING

MAURÍCIO CECÍLIO MAGNAGUAGNO

Dissertation presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Felipe Meneguzzi

Porto Alegre
2016

REPLACE THIS PAGE
WITH THE LIBRARY
CATALOG INFORMATION

REPLACE THIS PAGE WITH THE PRESENTATION TERM

USANDO PADRÕES DE OPERADORES PARA INFERIR CONHECIMENTO DE DOMÍNIO EM PLANEJAMENTO

RESUMO

A utilização de conhecimento de domínio torna possível resolver problemas complexos em menos tempo, focando em decisões chave durante a busca pela solução. Obter esse conhecimento requer um especialista ou um mecanismo de aprendizado e soluções do mesmo domínio para servir como base para extração do conhecimento. Muitas vezes diferentes domínios compartilham sub-problemas que possuem soluções estruturalmente equivalentes. Mesmo que sua descrição use um vocabulário específico ao domínio a forma das transições de um estado para outro segue um padrão. Esse padrão visto nas transições pode servir de base para extração de conhecimento de domínio comum entre diferentes domínios, podendo ser inferido com base apenas na descrição, sem necessidade de soluções ou especialistas. Nossa pesquisa extrai conhecimento de domínio de forma automática, causando uma diminuição ou eliminição da necessidade de um especialista na elaboração do conhecimento de domínio. Diferente de outros trabalhos nosso método não necessita de exemplos como base para extração. Isso resulta em um processo de descrição de domínio mais rápido enquanto erros humanos são evitados.

Palavras Chave: Planejamento Automatizado, Plano, Conhecimento de domínio.

USING OPERATOR PATTERNS TO INFER DOMAIN-KNOWLEDGE IN PLANNING

ABSTRACT

The use of domain knowledge allows algorithms to solve complex problems requiring less time by focusing on key decisions during the search for a solution. An expert or set of solutions is required to be used as a base for knowledge extraction. Several domains share sub-problems with an analogous structure to solve them. Even if the description use a different vocabulary specific to the domain, their shape matches a pattern. Such pattern seen in the operators can be used as a base to common domain knowledge extraction between different domains, being inferred from the domain description, without the need for solutions or experts. Our research extracts domain knowledge in an automated fashion, which leads to a minimization or elimination of an expert in the domain knowledge acquisition process. Unlike other methods ours do not require training examples. This results in less description time while human error is avoided.

Keywords: Automated Planning, Plan, Domain knowledge.

LIST OF ALGORITHMS

LIST OF FIGURES

CONTENTS

1. INTRODUCTION

Finding a sequence of actions to reach a desired state may be considered a trivial problem for a human, but when faced with a large number of possible actions, the task of finding such a sequence becomes a complex problem. Humans are less interested in finding optimal plans or the time taken to plan on a daily basis than they are interested in the effort to execute the plan and the rewards obtained from it. In order to obtain good plans in an specific domain we usually think that a person with expert level in such domain is required to consider the situation, identify relevant information and think about how to proceed in order to reach a goal. Oftentimes, we do not have an expert available or with fast response for a large set of information available, and an automated solution is required. Thus, arises the need to build tools capable of planning in an automated fashion, being able to solve as many different problems as possible, with concerns about plan quality and response time.

Automated planning is a sub-area of artificial intelligence that covers the search for a sequence of actions that, when executed, satisfy a goal. Usually two types of planners are employed, using different levels of knowledge about the environment. The first type uses only transition information, what matters in the current state to successfully apply a transition, while the other also uses a hierarchy. Planners that receive only transition information may take a lot of time to find a solution, even for simple problems, having to explore a vast search space. Smarter algorithms can be employed to prune the search space, otherwise intractable, using mutually exclusive information between actions [2] and heuristics to guide search [3]. The second type of planners is based on the exploration of explicit domain knowledge, often described as a hierarchical task network. Instead of just transitions, the domain knowledge is used to guide search [16]. This type of planner is more dependent on human knowledge, and often does not work without it.

Our main goal is to obtain an automated planning technique that captures part of the knowledge behind the transitions using common transition patterns. Current approaches often employ a library of solved instances, problems with their corresponding plans from the same domain, to serve as a training set to create macros [10, 4]. However, such approaches require a training set that covers a wide range of possible problems and has to be generated from a simpler planner first. This, in turn, requires a lot of computer time to plan and human effort to select interesting problems to generalize and obtain usable domain knowledge. Machine learning methods can also be used to model the action set from plans [12], which could help not just to describe the domain, but also find a simpler or different description that achieves the same result. Unlike existing approaches, we aim to identify operator patterns, how predicates are used in common operators that repeat across domains. With such patterns we can automatically generate domain knowledge related to their usage, avoiding the learning stage while using tailored solutions. The problem is how to find a generic set of operator patterns and how those patterns can be related to the goals in order to build the knowledge we need. Even incomplete domain knowledge created automatically can save time from the human designer while increasing reliability, as human errors may happen during this stage.

The structure of this dissertation is as follows: Chapter 2 reviews background on classical planning. Chapter 3 reviews background on hierarchical planning. In Chapter 4 we define how information present in classical domains can be used to automatically build hierarchical domain knowledge. Chapter 5 shows how particular features of each domain can be exploited and which operator patterns they are related to. A step-by-step example and results from several domains are given. In Chapter 6 we compare our work to recent related work. Chapter 7 summarizes the answer to research questions, closed and open goals and future work.

2. CLASSICAL PLANNING

Russel and Norvig [19] consider planning one of the most important parts of AI. Devising a plan of action to achieve one's goals is seen as a fundamental capability in several examples of autonomous agents. Classical Planning is based on the idea of finding a plan that satisfies a predefined goal. There are two important aspects about the environment that affect which planning techniques can be employed to pursue the goal. The first aspect is determinism, when the outcome of the transitions are always the same, the environment is considered deterministic, otherwise, the environment is non-deterministic with a set of outcomes for each transition with a known or unknown probability. The second aspect to be considered is the observability, some environments are fully observable, such as most board-games, while others are partially observable, as real-world scenarios, or completely non-observable from the agent's point of view, as a robot with sensor failure. In this work we focus on deterministic and observable environments. The complexity of the transitions impact both the description and the performance of the planning system.

2.1 Formalization

Classical planning consist of the following elements:

Definition 1 (Terms)**.** *Terms are either constants or variables. Constant terms may also be called objects, they represent what is possible to handle in the domain.*

Definition 2 (Predicate)**.** *A predicate is denoted by an n-ary predicate symbol* ***p*** *applied to a sequence of zero or more terms (t_1, t_2, ..., t_n). When all terms of a predicate are constants we call it a ground predicate.*

Definition 3 (State)**.** *In planning, a state is a finite set of facts and predicates that represent logical values (true or false) in order to describe the world configuration at a particular time.*

Definition 4 (Operator)**.** *An operator is represented by a 3-tuple* $\boldsymbol{o} = \langle \boldsymbol{name(o)}, \boldsymbol{pre(o)}, \boldsymbol{eff(o)} \rangle$*:* ***name(o)*** *represents the description or signature of* ***o****;* ***pre(o)*** *describes the preconditions of* ***o****, a set of facts or predicates that must be satisfied by the current state for action* ***o*** *to be applied;* ***eff(o)*** *represents the effects of* ***o****. The effects contain positive and negative sets,* ***eff(o)+*** *and* ***eff(o)-****, that add and remove facts or predicates from the state, respectively.*

Definition 5 (Action)**.** *An action is an instantiated operator over free-variables. During the planning process, states are tested to check if they satisfy the preconditions of actions. When they satisfy, the action effects can be applied, creating a new possible state.*

Definition 6 (Initial state)**.** *The initial state is represented by* $\boldsymbol{I} \subseteq \boldsymbol{F}$*, which is defined by a set of facts or predicates that represent the current world.*

Definition 7 (Goal state)**.** *The goal state is represented by* $\boldsymbol{G} \subseteq \boldsymbol{F}$*, which is defined by a set of facts or predicates that we desire to achieve by successfully applying the actions available.*

Definition 8 (Domain)**.** *The Domain is represented by* $\boldsymbol{D} = \langle \boldsymbol{F}, \boldsymbol{A} \rangle$*, which specifies the knowledge of the domain, and consists of a finite set of facts* $\boldsymbol{F}$ *and a finite set of actions* $\boldsymbol{A}$*.*

Definition 9 (Plan)**.** *The plan is represented by a sequence of actions that when applied in an specific order will modify the* $\boldsymbol{I}$ *to* $\boldsymbol{G}$*,* $\pi = \langle a_1, a_2, ..., a_n \rangle$*.*

Definition 10 (Planning Instance)**.** *A planning instance [7] is represented by the tuple* $\boldsymbol{P} = \langle \boldsymbol{D}, \boldsymbol{I}, \boldsymbol{G} \rangle$ *and returns* π *or failure.*

In order to solve a planning instance a planner algorithm is required. If successful, the planner returns a plan $\pi = \langle a_0, a_1, ..., a_n \rangle$ as a sequence of actions that, when applied to **I**, satisfies **G**. An action application is shown in Figure 2.1, testing if *predicate1* is true while *predicate2* is false in order to add *predicate2* and delete *predicate3*.

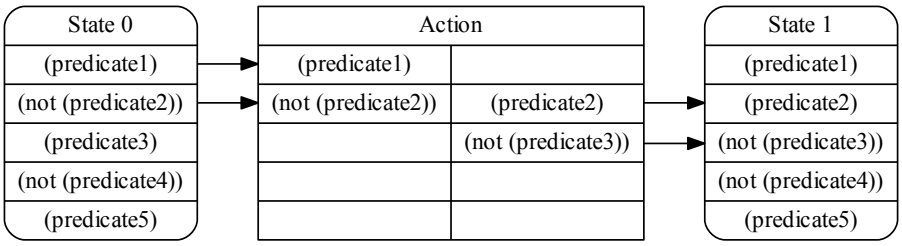

Figure 2.1 – Action application modifies only part of the state, the other predicates remain the same.

2.2 Domain and problem description

In order to better describe and reuse environments, different languages were proposed, separating planner and descriptions.

2.2.1 PDDL

The Stanford Research Institute Problem Solver (STRIPS) [5] is a planner recognized for originating one of the languages used to describe planning instances, referenced as STRIPS-like. The *Planning Domain Definition Language* (PDDL) [15] was created in 1998, extended STRIPS while creating a standard input to allow direct comparisons of efficiency between planning algorithms. The *International Planning Competition* (IPC) is now possible with the same input provided to the participants, in which different planning implementations compete in different categories about resources

or results obtained. In order to describe more complex domains the language supports several more specific features than the features of the STRIPS subset and, for this reason, such features are rare to find in most implementations that are optimized to serve only the most common features. The set of features required to solve each problem is explicitly declared as *requirements*, giving the planner an option to reject problems that require features not supported by its implementation.

The description is broken into a domain file with the operators description, and a problem file with initial and goal states. This separation makes it easier to reuse the domain description for different problems. Therefore planners expect both files as input to create a planning instance. Both files support comments, any text after a semicolon (*;*) is ignored until the end of the line.

A domain file contains at least three components: a domain name, a declaration of the valid predicates in the domain and the operators. Each action is composed of at least a name, a set of parameters, a precondition formula and an effect formula. The name of an action is a signature, used to differentiate which operator was used to instance such action. The set of parameters represent the free-variables the planner must replace by objects present in the problem. Each parameter is prefixed by a question mark (*?*). The precondition formula uses logical operations to describe the preconditions, such as negation, conjunctions and disjunctions of predicates, and must be satisfied by the current state. The effect formula uses the logical operations to describe what is changed in the current state in order to create a new state. Actions also support more complex descriptions through certain requirements, such as *:typing* for typed parameters and *:action-costs* for a cost value associated with the action application.

The problem file contains the domain name, the problem name, the objects and the initial and goal states. The domain name can be used to match with the name from the domain file and warn the user when they are different. The objects declare the possible values terms may assume. The initial state is a set of all the facts that are true before planning takes place. Facts that do not appear in the initial state are considered false. The goal state is a formula that must be satisfied, as the action preconditions, it does not need to declare a conjunction of all possible facts as true or false, only what is desired to be true or false. It is important to note how disconnected the operators are from each other, which simplifies the description process to add or delete operators as long as they correctly use the predicates of the domain.

One of the basic examples is Blocks-World, in which several blocks can form stacks on a table. The blocks can be stacked and unstacked from each other, as well as picked up and put down on the table. The initial state consists of a configuration of blocks, and the goal state another configuration of some or all the blocks previously defined. The domain consists of the generic part, no matter how many blocks were present the relations described through predicates and the operations possible are the same. The specific part, with the current and desired blocks configuration, is the problem. A possible PDDL description is illustrated in Listing 2.1 for the domain, and Listing 2.2 for the problem. The domain presents the name *Blocks-World* in Line 1. The requirements include *:strips* in Line 2, which means it only uses the basic constructions of PDDL. 4 predicates are defined in Line 3. 4 operators are defined in Lines 4-8, 9-13, 14-19 and 20-25. Operators *pickup* and *putdown*, as

```
1  (define (domain Blocks-World)
2    (:requirements :strips )
3    (:predicates (clear ?x) (onTable ?x) (holding ?x) (on ?x ?y) )
4    (:action pickup
5     :parameters (?ob)
6     :precondition (and (clear ?ob) (onTable ?ob) )
7     :effect (and (holding ?ob) (not (clear ?ob)) (not (onTable ?ob)) )
8    )
9    (:action putdown
10    :parameters (?ob)
11    :precondition (and (holding ?ob) )
12    :effect (and (clear ?ob) (onTable ?ob) (not (holding ?ob)) )
13   )
14   (:action stack
15    :parameters (?ob ?underob)
16    :precondition (and (clear ?underob) (holding ?ob) )
17    :effect (and
18      (clear ?ob) (on ?ob ?underob) (not (clear ?underob)) (not (holding ?ob)) )
19   )
20   (:action unstack
21     :parameters (?ob ?underob)
22     :precondition (and (on ?ob ?underob) (clear ?ob))
23     :effect (and
24       (holding ?ob) (clear ?underob) (not (on ?ob ?underob)) (not (clear ?ob)) )
25   )
26 )
```

Listing 2.1 – Blocks-World Domain

```
1  (define (problem pb5)
2    (:domain Blocks-World)
3    (:objects red green blue)
4    (:init (onTable red) (onTable green) (on blue red) (clear green) (clear blue))
5    (:goal (and (on red green) (on green blue) ) ) )
```

Listing 2.2 – Blocks-World Problem

seen in Lines 4 and 9, are used to move from or to the table and only require one parameter, *?ob*, to decide which block is the object being moved. The operators *stack* and *unstack*, as seen in Lines 14 and 20, are used to move blocks from or to other blocks, which requires two parameters, *?ob* for the block that will stay on top and *?underob* for the one which is or was under *?obj*.

The problem is named *pb5* and is part of the Blocks-World domain, as seen in Lines 1-2. 3 objects are available to substitute *?ob* and *?underob* as action parameters, as seen in Line 3. The initial state at Line 4 defines a block at the bottom, a stack and the top block of the stack as clear. The goal state at Line 5 have a conjunction that specifies the position of each block in the stack, but no top or bottom specification is provided.

Figure 2.2 shows the initial state with three different blocks, named *red, green* and *blue* in the description. The goal state is a stack of such blocks. Unlike the initial state, the goal state does not need to consider everything, only the relevant parts. Sometimes we do not know how the objects will be affected, leaving the goal less constrained. In the example of Listing 2.2 some information about

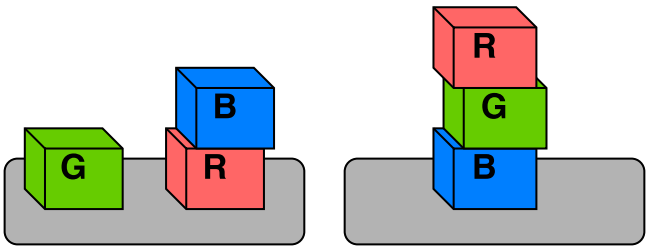

Figure 2.2 – Blocks-World problem, initial state(left) and goal state(right).

the goal is not described (the block on top being clear and the block at the bottom being on the table). Although this missing information does not cause any impact on the result, it may cause impact in the planning process. Some planning techniques may avoid moving the *blue* block once they know this block must remain on the table, which would prune the possible states created by actions that remove *blue* from the table.

2.3 Planning in the state-space

Planning can be seen as a search problem and therefore solved by search algorithms, such as breadth-first search (BFS). With BFS the solution found is certain to be step-optimal, the smallest plan possible, but the time and memory cost to reach this solution is far from optimal. This happens due to the amount of actions tested and states explored during search. Depth-first search can also be used, but must have a limited depth or there is a risk of reaching repeated states, never backtracking to reach a goal state on other branch. In order to avoid exploring repeated states we need to keep track of the visited states, which prunes several branches while solving the depth-first search loop problem.

An implementation of a search based planner is shown in Algorithm 1. This implementation searches state-by-state, testing and applying actions to generate more states to explore. Only new states are explored. It stops when no more states are left to be explored, a plan is found or the computer is out of memory. Several factors affect the performance of such planner: the *amount of states* of the planning instance may be too large to return a plan or failure within memory and time constraints; the *amount of actions* available in the domain will take a lot of processing time testing if preconditions are satisfied; the *visited structure* keeps growing as planning proceeds, which impacts both memory and processing time.

Techniques to speed-up planning focus on one or more of the three factors previously identified. Most domains are simplified in order to bring down the amount of states, but even simplified domains have enough states to require more optimizations to be solved within our time and memory constraints. One technique is to avoid the intermediary states and build plans without removing the free-variables, leaving the hard work for a future step, searching in the plan-space, or select which states are more promising to lead to the goal.

Selecting which states are more promising requires promising to be formally defined. In this case promising can be defined as the distance between current and goal states. With discretized

Algorithm 1 State-space planner using BFS search

```
1:  function BFS_PLANNER(operators, init, goal, objects)
2:      visited ← Set
3:      APPEND(visited, init)
4:      explore ← Queue
5:      APPEND(explore, ⟨init, ∅⟩)
6:      while explore ≠ ∅ do
7:          state, plan ← SHIFT(explore)
8:          for each action ∈ INSTANCE(operators, objects) do
9:              if SATISFIED(state, PRECONDITION(action))
10:                 new_state ← APPLY(state, EFFECTS(action))
11:                 if new_state ∉ visited
12:                     if SATISFIED(new_state, goal)
13:                         return BUILD_PLAN(plan, action)
14:                     APPEND(visited, new_state)
15:                     APPEND(explore, ⟨eff, ⟨act, plan⟩⟩)
16:     return failure
```

states the distance can be measured as the difference between both states or based on the amount of actions required to obtain a certain ground predicate. Giving priority to states that appear to be closer to the goal and we may solve planning instances faster. This mechanism is called heuristic search, it may help the planner expands only promising states. If the distance function, an heuristic, is too simple, we may obtain false positives and explore states that are in fact not promising. If the heuristic function is too complex we lose too much time computing it. It may not help to search faster even if the obtained values are much closer to reality. Some of the most recognized planners that employ this technique are Heuristic Search Planner (HSP) [3] and Fast-Forward (FF) [9].

There is also the *amount of actions* and the *visited structure* consuming resources. An analogous idea could be used to select which actions are more promising for each state. Expand this idea and we have a sequence of actions more promising for each each state. Such sequence could act as a recipe to be followed by the planner if carefully described in the domain. Since the person which describes the domain may have an idea of common recipes it is feasible to think that it is possible to have a set of recipes that solve any problem the planner may face. Such recipes are called domain knowledge. Knowing which actions were applied remove the need to keep track of which states have been visited. In the next chapter we explore such technique.

3. HIERARCHICAL PLANNING

Instead of using heuristics to select which state appears to be more promising we can describe how operators can be applied more successfully towards our goal, encoding domain knowledge. With such knowledge in a hierarchical structure we have hierarchical planning. Hierarchical planning shifts the focus from goal state to tasks to be solved in order to exploit domain knowledge about problem decomposition. The knowledge about the domain is used to build a hierarchy of operators (primitive tasks) and methods (non-primitive tasks) to solve such tasks by decomposition. Unlike classical planning, hierarchical planning considers only plans that can be generated through the decomposition of tasks using the specified domain knowledge. The domain knowledge is the hierarchy itself in this model, which makes the domain description much more complex than the classical planning description, with only operators. Hierarchical planning can be compared to developing tailored search solutions without worrying with the search mechanism, leaving a domain designer with only the domain to be described.

3.1 Formalization

The problem in HTN is given as a set of tasks to be solved. Two types of tasks are present in HTN: primitive tasks analogous to the operators of classical planning and non-primitive tasks, also called methods. HTN consist of the following elements:

Definition 11 (Methods)**.** *Methods are represented by a 3-tuple* $\boldsymbol{m} = \langle \boldsymbol{name(m)}, \boldsymbol{pre(m)}, \boldsymbol{subtasks(m)} \rangle$*:* ***name(m)*** *represents the description or signature of* ***m****;* ***pre(m)*** *describes the preconditions of* ***m****, a set of facts or predicates that must be satisfied by the current state for method* ***m*** *to be decomposed;* ***subtasks(m)*** *represents which tasks are going to occupy* ***m*** *in the task list. The subtasks contain other tasks, primitive or not, that are used to decompose the current task. The set of all methods is represented by M.*

Definition 12 (Tasks)**.** *The tasks are represented by* $\boldsymbol{T} = \langle \boldsymbol{A}, \boldsymbol{M} \rangle$.

Definition 13 (Domain)**.** *The domain is represented by* $\boldsymbol{D} = \langle \boldsymbol{F}, \boldsymbol{A}, \boldsymbol{M} \rangle$ *which specifies the knowledge of the domain, and consists of a finite set of facts* ***F****, a finite set of actions* ***A*** *and a finite set of methods* ***M****.*

Definition 14 (Hierarchical Planning Instance)**.** *A hierarchical planning instance is represented by the tuple* $\boldsymbol{P} = \langle \boldsymbol{D}, \boldsymbol{I}, \boldsymbol{T} \rangle$.

In order to solve a planning instance a hierarchical planner algorithm is required. If successful, the planner returns a plan $\pi = \langle\ a_0, a_1, ..., a_n\ \rangle$ as a sequence of primitive tasks that, when applied to **I**, satisfies an implicit goal state. Although not usual some tasks may have an empty set of subtasks, representing a no-operation due to the current state. This hierarchical structure can contain

loops, when methods contain itself in the subtasks, which may pose as a simple solution or a complex to debug feature.

3.2 Task as goal

The goal in HTN is to decompose all tasks into primitive tasks which results a plan π with the sequence of actions required to satisfy an implicit goal state requirements. The SHOP algorithm [16] is shown in Algorithm 2. It starts with the initial state, a list of tasks and the decomposition structure. The algorithm uses a recursive decomposition that tests whether the current task is primitive or not. The base case of the recursion is an empty list of tasks, which stops the planning process returning an empty plan. Otherwise it continues by picking the next task, using the SHIFT function, which removes a task from the list, as seen in Line 4. Every task can only be applied if its preconditions are satisfied by the current state. Some tasks may have variables to be solved during search, a unification of free-variables, which may generate several APPLICABLE tasks. A task that is applicable at the current state does not guarantee it will lead to a successful path as it may invalid facts or predicates required to keep exploring, which returns a failure. If a failure is returned, the algorithm will retry with other values for the free-variables of the current task. Such operation is called backtracking, as it forces the algorithm to go back and try a different path. If the current task is a primitive and applicable task the decomposition continues with the applied effects of the current task, as seen in Lines 5-7. If this branch eventually complete the tasks the current task is concatenated to the plan, note that this builds the plan in reverse, as seen in Lines 8-9. The other possibility is to have a non-primitive task, as seen in Line 10, in which every applicable decomposition generates a set of subtasks to replace the non-primitive current task in the task list. The plan is returned if this branch eventually complete the tasks. If not applicable the decomposition stops and a failure is returned.

Algorithm 2 SHOP planner

```
1:  function SHOP(state, task_list, decomposition)
2:      if task_list = ∅
3:          return empty plan
4:      current_task ← SHIFT(task_list)
5:      if current_task is a primitive task
6:          for applicable_task ∈ APPLICABLE(current_task, state) do
7:              plan ← SHOP(APPLY(applicable_task, state), task_list, decomposition)
8:              if plan
9:                  return applicable_task · plan
10:     else if current_task is a non-primitive task
11:         for each method ∈ decomposition[current_task] do
12:             for each subtasks ∈ APPLICABLE(method) do
13:                 plan ← SHOP(state, subtasks · task_list, decomposition)
14:                 if plan
15:                     return plan
16:     return failure
```

Backtracking is required for flexibility but can be avoided if look-aheads cover several cases of the domain and guide the search directly to a solution or failure. This alone already makes HTN planning much faster than classical planning, being able to prune several branches from the search. This style of planning is able to describe the same as STRIPS with a built-in heuristic function tailored to the domain and designer preferences [14], with all the non-primitive tasks required beforehand, which consumes human time to consider every single case. The model is not as incremental as classical planning, in which operators could be added or deleted, requiring some modification to the non-primitive tasks when an operator is added or removed. One of the most know implementations of a HTN planner is SHOP [16] and its successors SHOP2 and JSHOP2 [11].

3.2.1 Domain and problem description

A domain description in JSHOP contains two main components: first, a list of operators with the exact same semantics as PDDL, but formalised in a slightly different way and second, a list of methods that describe how tasks in the domain are decomposed. This description follows the Lisp-like style of PDDL, predicates with variables prefixed by a question mark (*?*), without the labels for preconditions, effects or subtasks of the operator or method. The order of the parts imply what they are. The operator represents the same as the classical operator, an action that can take place in this domain. Operators have a name, a set of parameters and three sets. Their name is prefixed by single (*!*) or double exclamation marks (*!!*). Operators prefixed by double exclamation marks do not appear in the final plan, they are considered internal actions relevant only for the planning mechanism. The first set present in the operator represents the preconditions, the second set the negative-effects with what is going to be false in the next state, and the final set the positive-effects with what is going to be true at the next state. Any set may be represented by open and close parentheses (*()*) or *nil* if empty. The methods have a name, parameters and preconditions as well, and instead of effect they have a set of subtasks to be performed in the same order they were given. Methods can also be decomposed in more than one way, each one with its own preconditions and subtasks, and have an optional label or sub-name for each case. The problem contains two sets, the first represents the initial state and the second a list of tasks to be performed. Note that the objects and predicates are implicitly defined in the initial state, unlike PDDL.

3.2.2 Domain example

In order to better understand the difference between classical and hierarchical descriptions we use the travel domain, one classic example of a hierarchical planning domain, shown in Figure 3.1. An incomplete description for the Travel domain is shown in Listing 3.1, and the related problem in Listing 3.2. The task is to move one agent between two destinations, in this case *Bob* is the agent and must move from the *city* to the *jungle*, which is declared by *(travel bob city jungle)*. The agent starts

at a position and is limited by *stamina* and *money* available. Greater distances require more stamina than available and cannot be covered walking by, which requires the agent to spend money on a taxi. If the distance is above a certain threshold and the agent have money enough to pay for the ride, the taxi is preferred. The method precondition is used as a look-ahead, in this example it occurs as the stamina check, as seen in Lines 16-17, and in the payment of the driver in Lines 27-28, as the agent will not ride the taxi without enough money to pay, which avoids backtracking once the payment fails. The use of continuous literals instead of a discrete form makes the money and stamina more realistic while using a scenario were preferences can be used to guide search, giving space for plan quality as the agent only rides the taxi if it considers necessary.

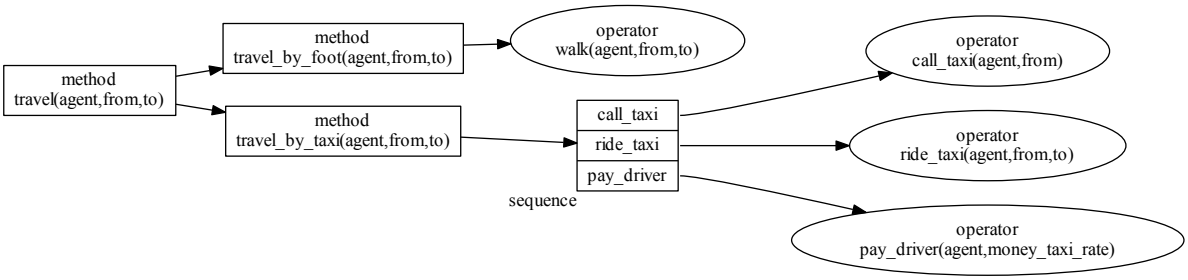

Figure 3.1 – HTN structure of simple travel scenario.

```
(defdomain traveldomain(
  (:operator (!walk ?agent ?from ?to)
    ; precondition
    ( (at ?agent ?from) (stamina ?agent ?s) (distance ?from ?to ?d) )
    ; del-effect
    ( (at ?agent ?from) (stamina ?agent ?s) )
    ; add-effect
    ( (at ?agent ?to) (stamina ?agent (- ?s ?d)) )
  )
   ...
  (:method (travel ?agent ?from ?to)
    ; label
    by_foot
    ; precondition
    (
      (at ?agent ?from) (stamina ?agent ?s) (distance ?from ?to ?d)
      (call < ?d ?s)
    )
    ; subtasks
    ( (walk ?agent ?from ?to) )
  )
  (:method (travel ?agent ?from ?to)
    ; label
    by_taxi
    ; precondition
    (
      (at ?agent ?from) (have_money ?agent ?m) (distance ?from ?to ?d)
      (call >= ?m (call + 1.5 (call * ?d 0.5)))
    )
    ; subtasks
    (
      (call_taxi ?agent ?from)
      (ride_taxi ?agent ?from ?to)
      (pay_driver ?agent (call + 1.5 (call * ?d 0.5)))
    )
  )
)
```

Listing 3.1 – Travel methods

```
(defproblem problem traveldomain
  ; initial state
  (
    (at bob city) (stamina bob 200) (have_money bob 5000)
    (distance city jungle 1200) (distance city farm 100)
  )
  ; tasks
  ( (travel bob city jungle) )
)
```

Listing 3.2 – Travel problem

4. APPLYING OPERATOR PATTERNS TO PLANNING

This chapter discusses our approaches for the identification and use of operator patterns for automated planning. We start with the operator patterns identified based on how predicate usage among the operators induces the operators to be applied in certain orders to reach a goal. With the operator patterns identified we create methods that solve the sub-problems related to operators, which creates a HTN domain, and exploit them in the form of tasks.

4.1 Operator patterns

Classical planning domains are simpler to describe and consist primarily of a set of operators that specify how they change objects in the environment. Since these domains only specify the transition function, they contain no information about which particular sequences of operators are more or less promising in the search for a plan to achieve a goal. Such information is useful in order to prune several actions that do not help to satisfy the goal. One particular approach to specify additional domain knowledge in classical planning is to use hierarchies of tasks and how they are decomposed. Here we use operator patterns to find implicit knowledge often found in planning domains in order to automatically create HTN-style domain knowledge.

In order to automate the process of method composition for classical planning descriptions, we must first identify common patterns of operator usage to obtain a generic database of methods that could be used in planning domains. In this work, we rely on patterns based on predicate usage by operators. Using the predicates as a source of information was explored by Pattison and Long in goal recognition [17], who partition predicates into groups to help differentiate which predicates are more likely to be goals. Since their focus was on the goal recognition process, they explore the likelihood of each predicate to be a goal rather than the potential relations among operators based on the predicates used. Although Pattison and Long only look at partitions as either contributing or not towards a goal, we also look at the partitions in relation to how they contribute to the execution of other operators in the plan. Such partitions are the ones we consider for our work, as follows:

Strictly activating predicates cannot be removed from future states, if they are present in the initial state, no operator has an effect that deletes such predicate.

Unstably activating predicates can be removed by at least one action, but once removed from the current state cannot be re-added.

Strictly terminal predicates do not appear as either precondition or negative effects of any operator, so once added such predicates cannot be removed, meaning they must be part of the final state.

Unstably terminal predicates also do not appear as preconditions to any actions, but unlike strictly terminal predicates, they can be removed once they have been added.

Waypoint predicates involve transforming objects through a chain of related states, all defined by the same waypoint predicate, such as the *(at ?agent ?position)* predicate.

Our approach relies only on operators. The predicates that appear in preconditions are matched against other operators effects in order to find dependence relations between the operators. Once we find how operators are related we are able to infer how they can be applied. With such information we can exploit patterns to solve the underlying problems they are related to. Such approach involves focusing on one operator, such as the operator *move* of Listing 4.1, and matching it to a template operator, such as the one presented in Listing 4.2, which contains only information related to operator usage within the patterns. Some operator templates may involve relating the operator we are focusing on with other operators present in the domain. With the operators classified we can compose methods. Since some variable names may differ among the operators we need to generalize the variables between operators that appear in the same subtask-list, which is also required to have consistent method preconditions. Without generalization a predicate (have ?ag ?i) from the effects of operator X would not relate to operator Y, which contains the predicate (have ?agent ?item) in the preconditions. The methods may also suffer modifications in order to be connected with the rest of the HTN. With the methods we select candidates to achieve our goals, which are going to be the tasks of the HTN problem.

Using the operator *move* from Listing 4.1 as an example, in Lines 4 and 7 it is possible to see the operator swapping the value of the *at* predicate, subject to some constraint on the values of these predicates, in this case caused by the use of *connected*, as seen in Line 5 of Listing 4.1. This operator pattern consists of repeated executions of the same operator, or set of operators, related to the same predicate until an intermediate goal is satisfied. A generic version of the operator pattern can be found in Listing 4.2, in which we expect zero or more applications of the operator to achieve *(predicate ?obj ?goal)* to satisfy part of the goal state or other operator precondition. The matched operators must change the truth value of a predicate and both values (*?current* and *?goal*) must be constrained by another predicate that remains unchanged by this operator application.

Some domains may present several swap operators related to the same predicate, using specific versions of the operator, e.g. *move-horizontal* and *move-vertical*, or completely different, e.g. *walk*, *jump*, *fly*, *drive*. Such operators have different constraints, and some problems may require not a successive application of one or another operator, but an interleaved application of such operators in order to achieve a specific sub-goal, such as a position. We can think about the predicate constraints

```
1 (:action move
2  :parameters (?bot - robot ?source ?destination - hallway)
3  :precondition (and
4    (at ?bot ?source) (not (at ?bot ?destination))
5    (connected ?source ?destination)
6  )
7  :effect (and (not (at ?bot ?source)) (at ?bot ?destination) )
8 )
```

Listing 4.1 – Move operator

as the edges of a graph as shown by Figure 4.1 that can be traversed by the application of an action, moving an agent or object between two nodes that are appear as the constraint parameters. In this graph we represent two predicates as possible constraints: *(connected ?location1 ?location2)* and *(near ?location1 ?location2)*. The graph becomes more complex as more swap operators are involved.

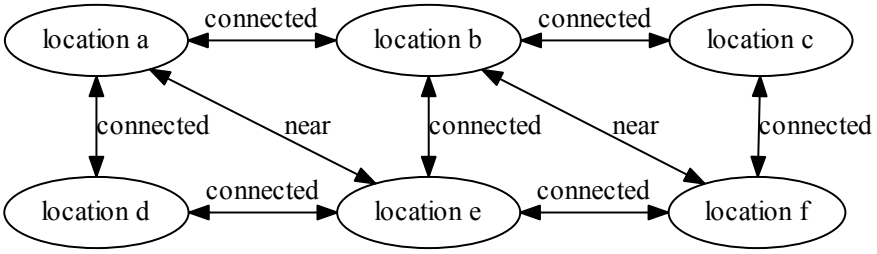

Figure 4.1 – Graph analogy used by the swap operator.

In cases where our constraints remain fixed one could think about solving such sub-problems separately from the rest, substituting the matched operator for an specialized solution, using a placeholder. We need to take care with the removal of constraints and understand the loss of a guarantee to find an optimal plan. The *connected* in our example of movement is one of the several possible names given to the predicate symbol that represent edges to our possible *at* position vertices. In Figure 4.2 it is possible to see the problem that a single missing edge from *b* to *f* creates, illustrated by non-filled arrows, splitting the single graph in two, making our problem resemble two islands. Any plan that required our agent to move between the islands would fail in the expansion process, which could ask for a different plan and restart this process until exhaustion.

If we use a specialized solver that relates different swap operators during the search process we can solve this problem more efficiently. In the next sections we exploit operator patterns based on generic problems within classical domains to compose HTN methods.

```
1 (:action swap_predicate
2  :parameters (?obj ?current ?goal)
3  :precondition (and
4    ( predicate ?obj ?current ) (not ( predicate ?obj ?goal))
5    ( constraint ?current ?goal)
6  )
7  :effect (and (not ( predicate ?obj ?current )) ( predicate ?obj ?goal) )
8 )
```

Listing 4.2 – Swap operator template

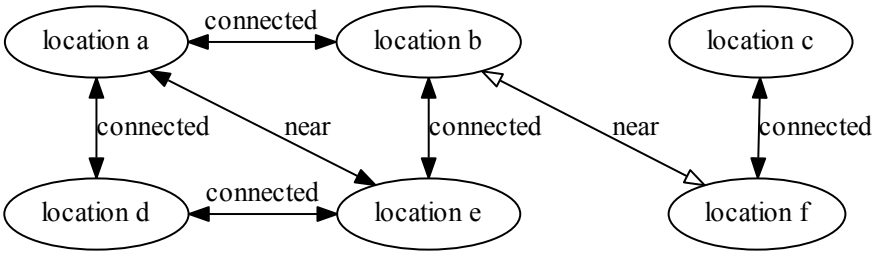

Figure 4.2 – Different regions are created based on operator constraints.

4.1.1 Representation

To represent the composition clearly, we use regular expressions to describe how operators are related, ignoring their parameters. Our representation require the following constructions:

Definition 15 (Grouping)**.** *Grouping using parentheses (()) to indicate a scope, e.g (op).*

Definition 16 (Disjunction)**.** *Disjunction using pipe (|) to indicate that a match must happen with one or other element, a sequence of disjunctions is allowed, e.g (op1|op2|op3).*

Definition 17 (Zero or more)**.** *Zero or more of the prefix using the star (*) to indicate a possible repetition of an element, e.g. (op1|op2)* matches ∅, {op1}, {op2}, {op1, op2}, {op2, op1}, and so on.*

Definition 18 (Zero or one)**.** *Zero or one of the prefix using the question mark (?) to indicate a possible use of an element, e.g. (op1|op2)? matches ∅, {op1}, {op2} and nothing else.*

The constructions can be combined as in the examples given to build more complex expressions. Other regular expressions constructions could be used,

4.1.2 Swap pattern

Consider a domain containing the *move* operator from Listing 4.1, which requires a robot to be at a certain location, and results in the robot being at a destination, subject to an adjacency constraint. Our first operator pattern is the *swap*, which requires at least one operator in the domain swapping the truth value between two instances of the same predicate (waypoint predicate partition). This pattern often appears in scenarios with discretized spaces where an agent swaps its current position among adjacent and free coordinates in an N-dimensional space, where N is the arity of the position predicate. The position is encoded in the predicate that is swapped by the execution of the *move* operator, while the adjacency is a constraint that implies this operator may be executed several times in order to traverse a discretized space. This operator pattern is related to the path-finding

sub-problem and was already identified and exploited by other planners to speed-up search. Hybrid STAN [6] is one such planner; it uses a path planner and a resource manager to solve sub-problems with specialized solvers.

Swap operators contain a constraint in the preconditions, otherwise the swap would have no restrictions requiring only one operator to solve the sub-problem and no method composition would be required. Since several operators may include the swap pattern over the same predicate, they can be merged into a single method with different constraints, such as a climb operator that changes the agent position like a move operator, but only if there is a wall nearby the current position. Swap identification can also be used to infer that an agent or object will never be at two different configurations in the same state, proving that no plan exists for such goal state.

The regular expression $(swap - operators)*$ represents how operators must be applied in order to build the swap method, going from *source* to *destination* using zero or more times the identified *swap* operators. If more than one operator swaps the same predicate we can represent the swap method with $(swap - operator1|swap - operator2|...|swap - operatorN)*$. Some domains may require application of the swap method until a certain predicate becomes true, this predicate is not necessarily the same one the operators are swapping and is present in the effects of the operators, e.g. an agent moves until a certain location is visited.

Two operators are used to track which configurations have already been explored: *visit_predicate* and *unvisit_predicate*. The first is used to mark the configuration as visited to the state, while the second is used to remove a visited configuration from the state.

Listing 4.3 shows two decompositions for *swap_at*. The first decomposition acts as the base of the recursion, with the agent at the goal position. The second decomposition moves one more step, marks the current position to avoid loops, recursively calling *swap_at* and unmarks the visited positions to be able to reuse the path later. The Figure 4.3 shows the two possible paths to be taken by the method, with predicates defined by ellipses and actions by boxes. The base of the recursion when current and destination positions are the same, which makes the goal state satisfied without actions being applied. And the body of the recursion when intermediary positions must be occupied by the agent to reach the destination.

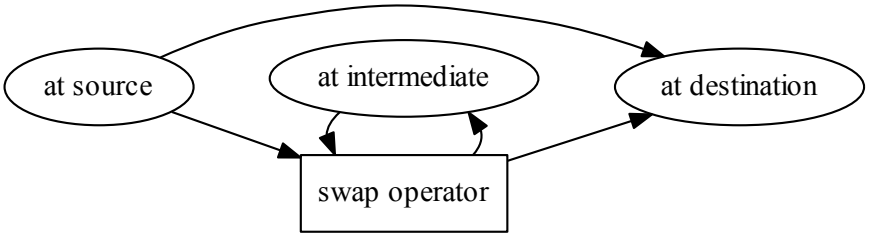

Figure 4.3 – Swap operators are applied zero or more times in a row, passing through several intermediate configurations until the goal configuration is reached.

```
(:method (swap_at ?object ?goal)
  ; label
  swap_at_base_at
  ; precondition
  ( (at ?object ?goal) )
  ; subtasks
  ()
  ; label
  swap_at_until_at_using_move
  ; precondition
  (
    (at ?object ?current) (not (at ?object ?goal))
    (connected ?current ?intermediate)
    (not ( visited_at ?object ?intermediate))
  )
  ; subtasks
  (
    (!move ?object ?current ?intermediate)
    ( !!visit_at ?object ?current)
    (swap_at ?object ?goal)
    ( !!unvisit_at ?object ?current)
  )
)

(:operator ( !!visit_at ?object ?current)
  ; Mark configuration as visited
  nil nil ( ( visited_at ?object ?current) )
)

(:operator ( !!unvisit_at ?object ?current)
  ; Unmark configuration as visited
  nil ( ( visited_at-robot ?object ?current) ) nil
)
```

Listing 4.3 – Methods for *swap* operator pattern using JSHOP description.

4.1.3 Dependency pattern

The Dependency operator pattern requires the existence of at least two operators with a dependency between the effects of the first with the preconditions of the second operator. Based on this requirement we know that every time we want to achieve one of the effects of the second operator we may have to apply the first operator, as seen in Figure 4.4, with predicates defined by ellipses and actions by boxes. If more than one operator may achieve the goal predicate, we must decide which one to test first.

The regular expression *((first)?second)?* shows the possible paths that may be taken during search to reach the goal. We apply the first operator when one of its effects is required to fulfill the preconditions of the second operator. If the preconditions are already fulfilled or the first operator was already applied, we can apply the second operator and obtain the goal. It is also possible that the goal has been achieved, which requires no action to be applied and no task to be decomposed. Expanding the regular expression we obtain Figure 4.4 with the three possible cases.

```
1  (:method (dependency_first_before_second  ?parameter)
2    ; label
3     dependency_first_before_second_with_goal_satisfied
4    ; precondition
5    ( ( goal_predicate ) )
6    ; subtasks
7    nil
8
9    ; label
10    dependency_first_before_second_satisfied
11   ; precondition
12   ( ( predicate  ?parameter) )
13   ; subtasks
14   ( (!second  ?parameter) )
15
16   ; label
17    dependency_first_before_second_unsatisfied
18   ; precondition
19   ( (not ( predicate  ?parameter)) )
20   ; subtasks
21   ( ( !first  ?parameter) (!second  ?parameter) )
22 )
```

Listing 4.4 – Methods for *dependency* operator pattern using JSHOP description.

Converting the three cases to JSHOP code we obtain Listing 4.4. Three different decompositions may take place based on the *goal_predicate* and *(predicate ?parameter)*. If the goal is satisfied the method decomposes to an empty set of tasks, as seen in Lines 2-7. If the precondition is satisfied the method decomposes to *(second ?parameter)*, as seen in Lines 9-14, otherwise *(first ?parameter)* must be applied before *(second ?parameter)*, as seen in Lines 16-21. Note that the subtask list of *dependency_first_before_second_unsatisfied* could also be written as *((first ?parameter) (dependency_first_before_second ?parameter))*.

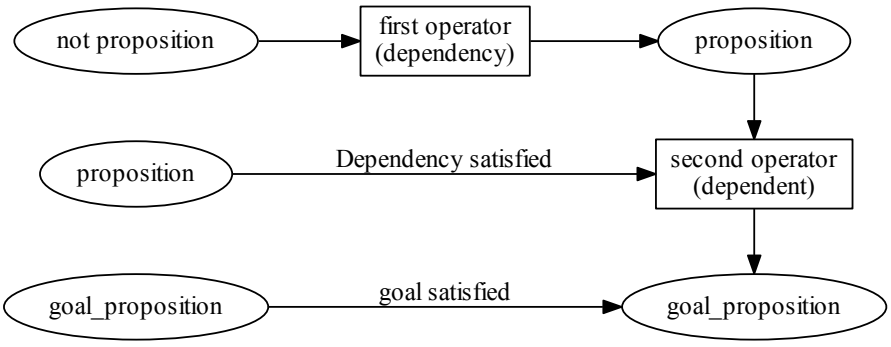

Figure 4.4 – Dependency may require the effects of other operator before application of the operator that achieves a goal predicate.

The *ratchet effect* is a special case of dependency that affects the ordering of actions. The *ratchet effect* pattern is, as the name implies, an operator that has an effect that cannot be undone by

other operators, which is related to unstably activating and strictly terminal predicate partitions. Some operators can only happen after the *ratchet effect* takes place, while others that require the negation of the effect can only happen before. As an example we can think of a domain with a *destroy* operator that prevents an object from being used by other operators once destroyed. The effect cannot be changed if no operator is able to recycle such destroyed object. In domains like this one the object remains in this state, creating a ratchet effect.

Any operator that requires such object to be usable must happen before the destroy action takes place, while others operators that require the object to be destroyed can only happen afterwards. Other operators that do not require the object state in their preconditions are considered unrelated and may happen before or after the destruction. This operator pattern can also be used to prune the search if certain conditions are met, if we seek two effects where each one appear in a different *ratchet effect* operator and those operators are mutually exclusive, the plan is impossible.

The ratchet effect dependency pattern can be seen in Figure 4.5, with predicates defined by ellipses and actions by boxes. An operator separate which other operators may happen before and after its application. The ellipses represent states that contain *(not (ratchet))* or *(ratchet)* as a fact. Operators unrelated to the ratchet effect may happen at any state, while other operators can only be applied before or after the ratchet effect takes place. If the ratchet effect is present in the initial state, only unrelated operators or operators with the ratchet precondition can be applied.

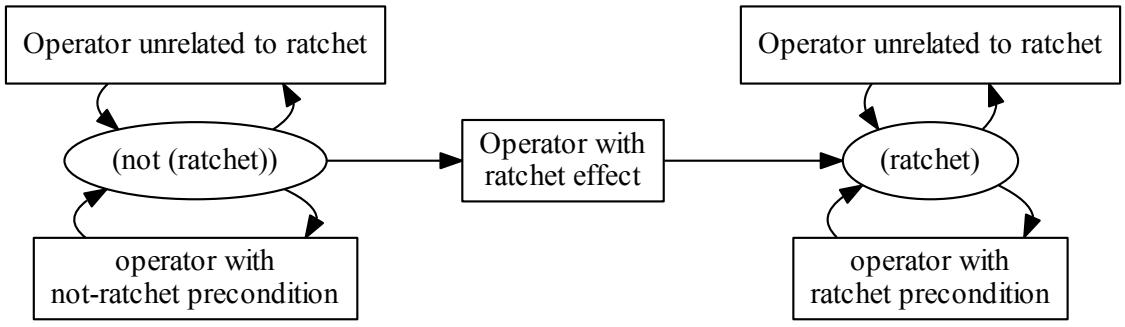

Figure 4.5 – Ratchet effect dependency pattern.

4.1.4 Free-variable pattern

Some predicates that appear in the goal state may not specify enough information to map to a task with all parameters replaced by constant values, e.g. *(occupied location1)* can be achieved by a move operator that requires an agent, the current location of the agent and the destination to occupy. Any method that have an operator that achieves *(occupied ?location)* is a potential task candidate. Operators with no swap or dependency patterns can also be candidates, primitive tasks.

```
(:method (apply_op1 ?p1 ?p2 ?p3)
  ; label
  apply_op_with_three_parameters
  ; precondition
  ( (precond1 ?p1 ?p2) (precond2 ?p2 ?p3) )
  ; subtasks
  ( (!op1 ?p1 ?p2 ?p3) )
)

(:method (unify_apply_op1 ?p1 ?p2)
  ; label
  unify_parameter_p3
  ; precondition
  ( (precond2 ?p2 ?p3) )
  ; subtasks
  ( (apply_op1 ?p1 ?p2 ?p3) )
)
```

Listing 4.5 – Methods for *free-variable* operator pattern using JSHOP description.

The *?location* value is bound to *location1* based on the goal predicate. Any agent and its current position are possible values to be used in such task. Instead of picking one agent at this point we will consider both values as free-variables. The possible values of such free-variables will be replaced at run-time until successful ones are found. The Free-variable pattern takes place as a task with the single purpose of finding such values before propagating them to the selected candidate task that achieves our goal.

In order to unify variables such as the agent from the previous example we create a method *unify_original_method* that unifies according to the constant preconditions of the related operator. Therefore we add a new level to the hierarchy with a method that simply unifies and propagates to the next level with all variables bound. Listing 4.5 shows a possible scenario where a method named *apply_op1* have an operator *op1* that achieves a goal predicate. The *op1* operator requires 3 parameters to be applied. Terms *p1* and *p2* are known based on information from goal state, while *p3* is left to be decided during run-time. It is possible to apply directly the original method in cases where the three terms are know. We do not merge both methods in one method to explicitly say that we are looking for the value of *p3*.

4.2 Composing methods and tasks

With the previous algorithms we can discover domain knowledge from a classical planning description. Such domain knowledge is exploited in this chapter to create a HTN planning description, with methods describing how operators can be applied and tasks defining which methods must be applied to achieve the goal state. In order to convert goals to tasks we first explore possible relationships among the operators and exploit the ones that are related to the goals we want to achieve. The first step is to differentiate between constants and mutable predicates. Algorithm 3 shows how we extract this information from the operators. For every predicate p present in the domain we clas-

sify how it is used by the operators. If p appears in the effect list it is considered mutable. If p only appears in the precondition list it is considered a constant, as no operator modifies it during planning. Otherwise the predicate is irrelevant. Now we know which predicates are used and we can infer which parts of the goal we can modify using the operators available in the domain. The predicate partitions of Pattison and Long only partition mutable predicates, since they are working with goal recognition which never consider constant predicates as part of a goal, as no action is required to achieve.

Algorithm 3 Classification of predicates into irrelevant, constant or mutable

```
1:  function CLASSIFY_PREDICATES(predicates, operators)
2:      predicate_types ← Table
3:      eff ← EFFECTS(operators)
4:      pre ← PRECONDITIONS(operators)
5:      for each p ∈ predicates do
6:          if p ∈ eff
7:              predicate_types[p] ← mutable
8:          else if p ∈ pre
9:              predicate_types[p] ← constant
10:         else
11:             predicate_types[p] ← irrelevant
12:     return predicate_types
```

The process starts classifying the operators from the classical domain according to the swap or dependency operator patterns. The goals are matched against the effects of each operator to discover possible ways to achieve each goal. Methods are composed based on the previously identified operator patterns among the operators. Tasks are created by mapping which methods have an operator that achieves each goal. Use the free-variable pattern when tasks have values to be solved during run-time.

Algorithm 4 presents the the swap classification algorithm. The algorithm receives as input the operator set from the domain and the *predicate_types* obtained by Algorithm 3 and returns a table with the operators classified as swap and which predicates they are swapping. The algorithm compares each operator precondition with its own effects, trying to find which operators swap a predicate value that uses the same terms present in a constant precondition, a constraint. The algorithm starts with a table to be filled by the swaps found, as seen in Line 2. Next we iterate over the operator set, accessing each operator preconditions and effects, as seen in Lines 3-8. The *predicate_types* is used to partition the preconditions into what is mutable and constant Constant preconditions are obtained with CONSTANT_POSITIVE_PRECONDITIONS and represent the constraints we have when swapping the predicate values. The mutable preconditions are obtained by MUTABLE_POSITIVE_PRECONDITIONS and MUTABLE_NEGATIVE_PRECONDITIONS. Swaps are identified by predicates present as positive preconditions that are removed by the effects, as seen in Line 9, and added by the effects again with other terms, as seen in Line 10. Since the terms may be different we look for a predicate with the same predicate symbol. If such predicate reappears in the delete effects we filter which parameters are exclusive to each predicate, as seen in Lines 11-12. Such parameters are expected to be constrained by

some constant precondition. If they are we store which precondition was swapped and the constraint used by the operator, as seen in Lines 13-16.

Algorithm 4 Classification of swap operators

```
1:  function CLASSIFY_SWAP_OPERATORS(operators, predicate_types)
2:      swaps ← Table
3:      for each op ∈ operators do
4:          constraints ← CONSTANT_POSITIVE_PRECONDITIONS(op, predicate_types)
5:          pre+ ← MUTABLE_POSITIVE_PRECONDITIONS(op, predicate_types)
6:          pre- ← MUTABLE_NEGATIVE_PRECONDITIONS(op, predicate_types)
7:          eff+ ← ADD_EFFECTS(op)
8:          eff- ← DEL_EFFECTS(op)
9:          for each pre ∈ pre+ ∩ eff- do
10:             pre2 ← FIND_BY_PREDICATE(PREDICATE_NAME(pre), eff+)
11:             if pre2
12:                 cparam ← (params(pre) - params(pre2)) ∪ (params(pre2) - params(pre))
13:                 for c ∈ constraints do
14:                     if c ⊆ cparam
15:                         swaps[op] ← ⟨pre, constraint⟩
16:                         break
17:     return swaps
```

Algorithm 5 presents the dependency classification algorithm. The algorithm receives as input the operator set from the domain, the *predicate_types* obtained by Algorithm 3 and a table with the swap operators, from Algorithm 4, and returns a table with the operators classified as dependency and which predicates they connect two operators. The algorithm compares each operator preconditions with the effects of another, trying to find which operators may depend on the effects of other operators. We start with an empty table that will store the dependencies we find. This time we iterate over two operators, *op* and *op2*, trying to find when *op2* must be applied before *op*. Unlike swaps, we avoid dependencies that were already classified. In this case we ignore when both operators were classified as swaps and relate to the same predicate, as seen in Line 11. We also ignore cases where *op* is the same as *op2* or every precondition of op2 is contained within the effects of *op*, otherwise we would apply actions that achieve the original state, as seen in Line 14. Now we look for a precondition of *op* that is satisfied by an effect of *op2*, as seen in Lines 16-17. Any dependency found is stored, which may lead to several possible dependencies for *op*, as seen in Lines 18-20

Dependency may also happen between methods, sometimes one or more methods may have one operator in their task list that require other operators to be applied before them to satisfy one or more preconditions. With both dependency methods identified we can make the connection between the two. In order to so we use the Algorithm 6. In this algorithm we check if each method have such operator that depend on other operator that appears in a different method. First we pick a dependency method *met*, with a *dependency* operator that must happen before a *dependent* operator. Two cases were identified, in the first case we have a dependency of a dependency, as seen in Lines 6-8, and in the second case, we have a swap dependency before the dependent, as seen in Lines 9-12.

Algorithm 5 Classification of dependency operators

```
1: function CLASSIFY_DEPENDENCY_OPERATORS(operators, predicate_types, swaps)
2:     dependencies ← Table
3:     for each op ∈ operators do
4:         pre+ ← POSITIVE_PRECONDITIONS(op, predicate_types, mutable)
5:         pre- ← NEGATIVE_PRECONDITIONS(op, predicate_types, mutable)
6:         eff+ ← ADD_EFFECTS(op)
7:         eff- ← DEL_EFFECTS(op)
8:         for each op2 ∈ operators do
9:             swap_op ← swaps[op]
10:            swap_op2 ← swaps[op2]
11:            continue if swap_op and swap_op2 and swap_op[0] = swap_op2[0]
12:            pre2+ ← POSITIVE_PRECONDITIONS(op2)
13:            pre2- ← NEGATIVE_PRECONDITIONS(op2)
14:            continue if op = op2 or (pre2+ ⊆ eff+ and pre2- ⊆ eff-)
15:            eff2+ ← ADD_EFFECTS(op2)
16:            for each pre ∈ pre+ do
17:                continue if not FIND_BY_PREDICATE(PREDICATE_NAME(pre), eff2+)
18:                if dependencies[op] = ∅
19:                    dependencies[op] ← Set
20:                APPEND(dependencies[op], ⟨op2, pre⟩)
21:    return dependencies
```

The first case is represented by Figure 4.6, with three dependency methods, where the left path of each dependency method shows the unsatisfied branch with dependency and dependent and the right path with the satisfied branch with only the dependent. In this case we have *c* as a dependency of *b* that may affect the application of *a*. The injection procedure will connect both methods and evaluate at run-time the need for *c* to be applied, if the preconditions of *b* are unsatisfied.

The second case is represented by Figure 4.7, with three dependency methods, and follows the same principle of the previous example. Now we have a swap dependency for *a*, the dependent operator. Instead of one substitution we replace both branches by the swap dependency.

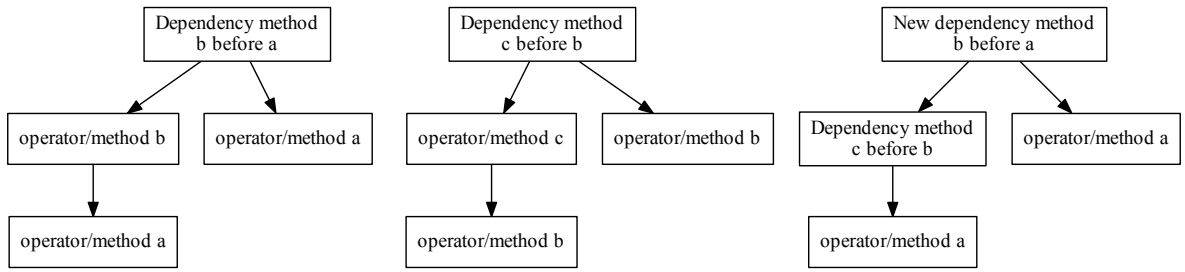

Figure 4.6 – Two dependencies (left and center) are merged by dependency injection (right) to satisfy the preconditions of *b*.

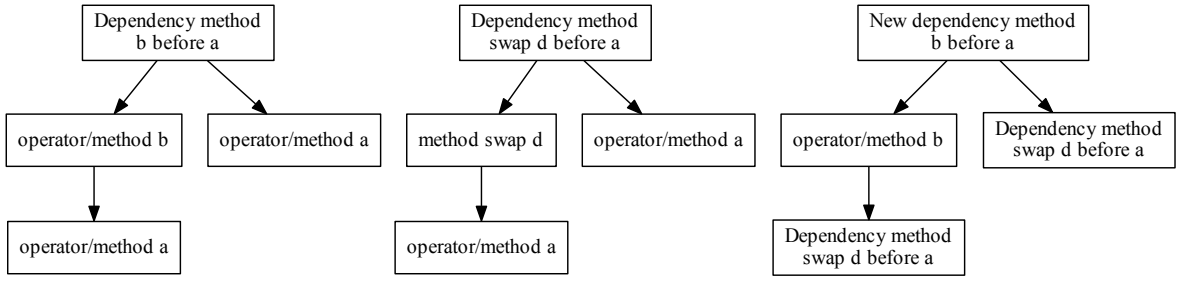

Figure 4.7 – Two dependencies (left and center) are merged by dependency injection (right) to satisfy the preconditions of a.

Algorithm 6 Dependency injection mechanism for methods

```
1:  procedure INJECT_METHOD_DEPENDENCIES(methods)
2:      for each met ∈ methods do
3:          if DEPENDENCY_GENERATED(met)
4:              dependency ← GET_DEPENDENCY(met)
5:              dependent ← GET_DEPENDENT(met)
6:              substitution1 ← FIND_METHOD_WITH_DEPENDENCY(methods, dependency)
7:              if sub and sub ≠ dependent
8:                  REPLACE(met, unsatisfied branch, dependency, substitution1)
9:              substitution2 ← FIND_METHOD_WITH_SWAP_DEPENDENCY(methods, dependent)
10:             if substitution2 and substitution2 ≠ met
11:                 REPLACE(met, unsatisfied branch, substitution2)
12:                 REPLACE(met, satisfied branch, dependent, substitution2)
```

Algorithm 7 shows the stages required to convert a classical description into a HTN. The identification process is based on the predicates in the preconditions and effects of every operator with our knowledge about which predicates are constant or not during planning. The first step is to get access to the operators, predicates and goals of the classical description, as seen in Lines 2-4. Methods and tasks start as empty sets, as seen in Lines 5-6. The methods set will contain the methods composed from the operators patterns found in the current domain. The tasks set will contain tasks that achieve the original goal predicates using the composed methods. The operator patterns found in the current domain are stored in *swaps* and *dependencies*, as seen in Lines 8-9. We order the pattern matching to test for swaps and then the most general dependency. We avoid classifying operators in more than one operator pattern, so the swap information is used by CLASSIFY_DEPENDENCY_OPERATORS.

The next step is to identify which operators can achieve part of our goal state. The operators that are more closely related to goals are the ones with a goal predicate in their effect list. We can use such goal operators to identify which sub-problems we are trying to solve, as seen in Lines 10-14.

With all the information about the operators and goals extracted we start the method and task composition. We use the operator patterns identified to generate methods for the HTN domain description. The methods are added independently of usage by the tasks, as some methods may give

a hint to the designer about the relation among operators even when a method is not connected to the rest of the hierarchy, as seen in Lines 15-16. Since HTN planners only consider what appears during decomposition this does not cause any slow down. The next step is to create and add methods to the HTN based on the operator patterns previously identified Specific swap methods are created for the predicates being swapped containing all operators that relate swap operators to the same predicate. If an operator matched the dependency pattern, a method is created for both cases of the precondition, when the precondition is already satisfied it only decomposes to this operator, otherwise it decomposes to the dependency and this operator. If the dependency is a swap identified operator we can already inject the swap method in the unsatisfied branch. Once we generate the methods using the operator patterns we need to add the tasks with the corresponding objects taken from the goal predicate to achieve such goals, as seen in Lines 17-23.

The complex dependencies are injected at this moment, if done earlier it would be more complex to analyze which methods are task candidates, as seen in Line 24. Sometimes more objects are involved in the method application than what was obtained in the previous step, taking information from the goal state. For those cases the free-variable pattern applies, it constructs a method that receives the original objects we knew before through propagation and unify the variables left open and required by the original method, propagating all the objects required, as seen in Lines 25-29.

The new operator set, method set and tasks returned by the algorithm represent an analogous version of the classical planning instance used as input. In the same sense as the classical goal where each goal predicate could be achieved at one step instead of the entire goal state at once, the tasks are unordered. Each task will reach part of the goal state when decomposed and applied, but to reach to the goal state all tasks may require decomposition. In order for such sequence of decompositions to occur successful, it is required for the tasks to be applied at a certain order. Such order is currently not decided at project time, it is left to be decided by the HTN planner. An interference detection mechanism or expert is required to check possible relations between the tasks according to the methods and operators available in the domain in order to have ordered tasks. The ordered task decomposition does not imply a faster execution, as some domains may be decomposed successfully by the first task ordering decided. With the new HTN planning instance an expert can decide if modifications are required for the system to work properly or faster.

Algorithm 7 Convert goals to tasks

```
1:  function GOALS_TO_TASKS(domain, problem)
2:      op ← OPERATORS(domain)
3:      pred ← PREDICATES(domain)
4:      goals ← GOALS(problem)
5:      tasks ← Set
6:      met ← Set
7:      predicate_types ← CLASSIFY_PREDICATES(op, pred)
8:      swaps ← CLASSIFY_SWAP_OPERATORS(op, predicate_types)
9:      dependencies ← CLASSIFY_DEPENDENCY_OPERATORS(op, predicate_types, swaps)
10:     goal_op ← Array
11:     for each o ∈ op do
12:         for each goal ∈ goals do
13:             if goal ∈ EFFECTS(o)
14:                 APPEND(goal_op, ⟨goal, o⟩)
15:     ADD_SWAP_METHODS(swaps, op, met, predicate_types)
16:     ADD_DEPEND_METHODS(swaps, dependencies, op, met, predicate_types)
17:     goal_tasks ← Array
18:     for each m ∈ met do
19:         for each d ∈ DECOMPOSITION(m) do
20:             for each ⟨goal, o⟩ ∈ goal_op do
21:                 if o ∈ SUBTASKS(d)
22:                     m2 ← UNIFY_VARIABLES(m, o)
23:                     APPEND(goal_tasks, ⟨goal, m2⟩)
24:     INJECT_METHOD_DEPENDENCIES(swaps, met)
25:     for each ⟨goal, m2⟩ ∈ goal_tasks do
26:         if free-variable ∈ m2
27:             APPEND(tasks, UNIFY_METHOD(m2))
28:         else
29:             APPEND(tasks, m2)
30:     return ⟨op, met, tasks⟩
```

5. DOMAINS

In order to evaluate our approach we selected and created domains with the operator patterns identified previously to find potential variations. Our goal is to cover several domains independently from their description style. The chapter starts by examining how various domains relate to swap or dependency operator patterns through examples and details of each domain that must be considered while composing methods. After the examination we see how such details appear in the Rescue Robot Robby domain in a step-by-step example. The chapter finishes with use cases found in IPC domains. Figures from this chapter define predicates by ellipses and actions or methods by boxes.

5.1 Swap pattern related domains

The most suitable experiments for the *swap* operator pattern are the Traveling Salesman Problem (TSP)[13] and the IPC Logistics domains. Both domains have an move operation which implicitly contain the swap operator. The TSP is usually concerned with a single agent, which means there is no need to add a free-variable that only unifies to this single agent object. On the other hand, Logistics differs which agents can traverse certain parts of the map based on their type. Airplanes being able to *fly* between cities, while trucks are able to *drive* within the city. Most problems resemble Figure 5.1, with packages to be moved across the locations.

Unlike other domains, the agent in the TSP domain does not have only a destination location among the goals, but have several locations to be visited. The only operator of TSP, *move*, is identified as a *swap* and is related to visited and final location goals. If we consider each ground predicate declared in the goal as one task, it is possible that, depending on the order of the goals and the initial location of the agent, some tasks may visit other locations until the destination is visited. This could be interpreted as a constructive interference between tasks, the current goal location was visited on a previous task, which removes the need to expand the current task. Unlike the swap present in other domains, the stop condition of the swap recursion may be the location to be visited or currently occupied by the agent.

This domain requires two different swap methods with different stop conditions (base of recursion) to solve the problem, one to solve the visited locations and the other to reach the final destination. This may not be a problem if the location was not visited before, as the agent must occupy the location to mark it as visited. The problem appears when the agent already visited that location and currently occupies the final destination, which will cause backtracking. This is not the only problem we encounter in this domain while looking for a generalized method to fulfill swaps. The next problem is the salesman himself, with only one agent in the domain there is no need to add an object to represent it and the operator move this agent implicitly, which is one of the description style problems we knew would appear, requiring a very flexible swap identification.

The Logistics domain of the IPC from 1998 also uses a different style to describe the constraints of the operator *drive*. Unlike the *fly* operator, that moves an airplane from/to any city with airports, *drive* requires the destination to be in the current city of a truck. The problem lies in the description style, instead of an *(adjacent ?from ?to)* or *(same-city ?from ?to)* predicates, two predicates *(in-city ?from ?city)* and *(in-city ?to ?city)* are used as preconditions. This example illustrates the need to consider variations of the constraints, or a merging process of the constant predicates before the identification process, otherwise we may not classify or use correctly the operators of the domain.

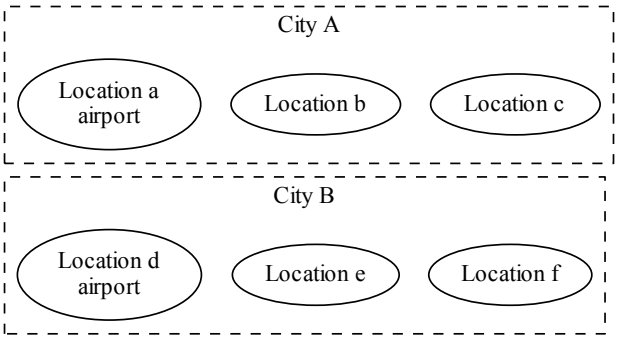

Figure 5.1 – Locations are not explicitly connected in the Logistics domain.

5.2 Dependency pattern related domains

The *dependency* operator pattern is more commonly found in planning domains than the swap pattern, since it is a general case of a predicate that connects the effects of one operator to the preconditions of another one. Here we consider simple examples to illustrate details of our current approach.

We converted to PDDL the basic domain example of JSHOP2, in which the task is to swap two objects. This domain has two operators: *pick-up* and *drop*, and is entirely made around a single predicate: *have*. The domain can be seen at Listing 5.1. It is only possible to pick an object that you do not have and only drop one you have. The simple dependency pattern between pick-up and drop happens in both ways: we can only drop something that we carry and only pick something dropped. Since we were using other domains that did not include such details to guide our experiments, we never had to check special cases such as this one, in which dependent operators may undo previous effects. Three paths were identified: the sub-goal is done, the dependency is satisfied, or the dependency is not satisfied. With the sub-goal done we can decompose the current task to an empty set of operators. With the dependency satisfied we can apply the second operator directly. Otherwise we must apply the first operator to create a state in which the second operator is applicable. In this case the unsatisfied dependency and the sub-goal are the same: *(not (have item))*.

```
(define (domain basic)
  (:requirements :strips :negative-preconditions)
  (:predicates (have ?a) )
  (:action pickup
   :parameters (?a)
   :precondition (and (not (have ?a)) )
    :effect (and (have ?a) )
  )
  (:action drop
   :parameters (?a)
   :precondition (and (have ?a) )
    :effect (and (not (have ?a)) )
  )
)
```

Listing 5.1 – Basic domain in PDDL.

Even without specific tests, our method is able to solve this basic problem by applying the *pick before drop* method created, which causes the agent to pick and drop the same item. It is possible to add tests to avoid cycles such as this one were the method application was not required since the goal was already fulfilled, but we wanted to show how dependencies may over-complicate the problem if the goal to task conversion does not check such details.

The last example was created specifically to demonstrate a special case of multiple dependencies among the operators. We created the gift-giver domain, present in Listing 5.2, with three operators: *work*, *buy* and *give*. In this domain several agents may *work* to obtain money, which brings sadness to the agents who does not like to work. With money the agent can *buy* a gift, which can be *given* to any agent, even itself. The receiver always gets happy with the gift, but since this was a gift it cannot be sold or given again. The dependency between the operators is: $work \rightarrow buy \rightarrow give$.

If we consider only operators that may achieve one of our goals to create the methods, we would identify only that *give* depends on *buy*. If we consider all operators, we find two dependencies: the previous one and that *buy* depends on *work*. Here pairs of operator dependency start showing that we need to relate more than operators. If we desire to apply *give* we need to apply the *dependencies* of *buy*, not just the operator *buy*. Generalizing this, we need to build the hierarchy recursively based on this information, stopping when deconstruction or repetitions appear. Figure 5.2 shows the resulting hierarchy from this process.

5.3 Rescue Robot Robby domain example

In order to illustrate how our operator patterns can be applied to a concrete domain, we use the Rescue Robot domain as an example, as several patterns are identified in the operator set. The domain description of the Rescue Robot domain is shown by Listing 5.3. This domain has a small operator set and can be represented by a 2D map, which means we can explore it deeply without complex constructions. The map contains rooms and hallways as locations where the rescue robot

```
(define (domain gift-giver)
  (:requirements :strips :typing :negative-preconditions)
  (:predicates (have ?a ?x) (got_money ?a) (happy ?a) )
  (:action work
   :parameters (?a - agent)
   :precondition (and (not (got_money ?a)) )
   :effect (and (not (happy ?a)) (got_money ?a) )
  )
  (:action buy
   :parameters (?a - agent ?x - object)
   :precondition (and (got_money ?a) (not (have ?a ?x)) )
   :effect (and (not (got_money ?a)) (have ?a ?x) )
  )
  (:action give
   :parameters (?a ?b - agent ?x - object)
   :precondition (and (have ?a ?x) (not (have ?b ?x)) )
   :effect (and (not (have ?a ?x)) (have ?b ?x) (happy ?b) )
  )
)
```

Listing 5.2 – Gift-giver domain in PDDL.

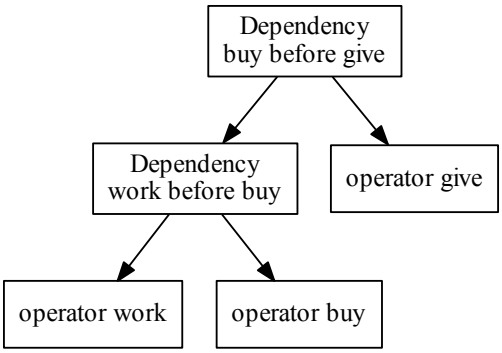

Figure 5.2 – Dependency over methods for the gift-giver domain.

and beacons may be located. The robot must be in the same hallway or room of a beacon to report the status. The set of operators include:

- *Enter* a room connected to the current hallway.
- *Exit* the current room to a connected hallway.
- *Move* from the current hallway to a connected hallway.
- *Report* status of beacon in the current room or hallway.

We can use our operator patterns to infer how such operators are related to the goal. The *Enter*, *Exit* and *Move* operators swap predicate *at*. They all require source and destination to be connected locations, which matches our constraint requirement. *Move* creates a dependency for *Enter*,

```
(define (domain robby)
  (:requirements :strips :typing :negative-preconditions)
  (:types
    robot beacon location - object
    hallway room - location
  )
  (:predicates
    (at ?bot - robot ?place - location)
    (in ?thing - beacon ?place - location)
    (connected ?place1 - location ?place2 - location)
    (reported ?bot - robot ?thing - beacon)
  )
  (:action enter
    :parameters (?bot - robot ?source - hallway ?destination - room)
    :precondition (and (at ?bot ?source) (not (at ?bot ?destination ))
      (connected ?source ?destination ) )
     :effect (and (not (at ?bot ?source)) (at ?bot ?destination ) )
  )
  (:action exit
    :parameters (?bot - robot ?source - room ?destination - hallway)
    :precondition (and (at ?bot ?source) (not (at ?bot ?destination ))
      (connected ?source ?destination ) )
     :effect (and (not (at ?bot ?source)) (at ?bot ?destination ) )
  )
  (:action move
    :parameters (?bot - robot ?source - hallway ?destination - hallway)
    :precondition (and (at ?bot ?source) (not (at ?bot ?destination ))
      (connected ?source ?destination ) )
     :effect (and (not (at ?bot ?source)) (at ?bot ?destination ) )
  )
  (:action report
    :parameters (?bot - robot ?source - location ?thing - beacon)
    :precondition (and (at ?bot ?source) (in ?thing ?source)
      (not (reported ?bot ?thing)) )
     :effect (and (reported ?bot ?thing))
  )
)
```

Listing 5.3 – Rescue Robot Robby domain in PDDL.

as *Enter* creates a dependency for *Exit*, but since they are already considered *swap* operators, a special case of dependency, we can prioritize *swap* over *dependency*. Swap *at* may be needed zero or more times to match the destination. Listing 5.4 presents the swap operator as separate patterns that, when related to the same predicate, can be clustered to form a single method. The automata generated is the same of Figure 5.3.

```
(Enter)*  (Move)*  (Exit)*
Swap_At = (Enter|Move|Exit)*
```

Listing 5.4 – Swap *at* operators clustered.

Dependency is the next pattern we explore, testing the operators that were not classified as Swap operators. Only one operator remains unclassified in this domain, *Report*, which has a precondition *at* to be satisfied. *Report* have a dependency with the swap *at* method previously built,

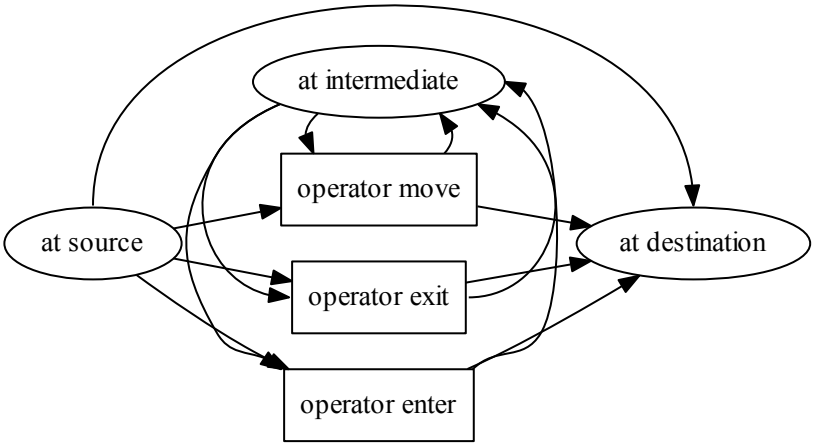

Figure 5.3 – Swap at predicate method composed to solve the Rescue Robot Robby problems.

```
1 ((Enter)?Exit)?   ((Enter)?Report)?
2 ((Exit)?Move)?    ((Exit)?Report)?
3 ((Move)?Move)?  ((Move)?Enter)?     ((Move)?Report)?
```

Listing 5.5 – Matching dependency before swap clustering generate several cases.

generating $(Swap\_At)Report$. Checking for dependencies first would generate several methods we do not use. The entire set of possible methods to be generated are in Listing 5.5.

Report is the only operator with an effect that cannot be undone, once a beacon is *reported* no operator removes this predicate from the state. In this case there is no operator that has *reported* in the preconditions, which makes the position of *Report* unconstrained in the method composition. Otherwise we would have an ratchet effect and operators that required *reported* as a precondition could only happen after *Report* is applied. Since no operator have *reported* as a precondition it never forces other operator to be applied before or after it was applied. At this point no operator of this domain remains unclassified and the goals are used to select which methods are required to solve the problem.

Instead of just finding the set of tasks, we need to define the methods called by the tasks to reach our goal. Some predicates in the goal may be achieved by the plan from a single method decomposition and the order in which the predicate become true is not relevant, as they must be achieved by the same state. The method application on the other hand must take ordering into account, as some effects may break posterior preconditions.

Rescue Robot's goal states involve beacons being reported, and sometimes a final position being achieved. The predicates in the goal state may not contain the position of each beacon to be reported, as in *(reported robot beacon1)*. The position is an important information, it appears in three operators effects and it is a precondition of another. We could extract the position based on the initial state in order to create a task with all terms unified, but other domains may present several objects

possible for the free-variable and we would be committing to a single object. To avoid committing to a single object, we use a unify free-variable task, where the position is unified during search.

With the position found, the robot must travel to each beacon, swapping the value of the predicate *at* using enter, exit or move zero or more times; and apply a report action at the beacon position. The regular expression $((Swap\_At)Report) * Swap\_At$ shows the expected pattern all plans should match for the solvable problems within this domain. This idea that every beacon location must be reached before the final destination is related with the goals as effects of the *dependencies* found.

5.4 Applying operator patterns to different domains

Each domain presents different operator patterns based on their description. In this section we discuss the operator patterns found and how they were exploited by the method composer. We limit ourselves to the domains that generated useful and generic methods, as more complex HTNs are difficult to evaluate and usually focus on specific domains that do not share sub-problems with others. Most of the domains selected are from the optimal track(seq-opt) from the IPC 2014 Benchmarks or from previous iterations of the IPC. Table 5.1 contains information about the domains. Each line refers to a domain, with N operators. Such operators were used to identify swaps and dependencies in the domain. The source represents if the description was made by us or from the IPC. The last column refers to the output, a little, most or complete HTN obtained using the method described in the previous chapter.

Table 5.1 – Domains used to evaluate the proposed method.

Domain	Operators	Swaps	Dependencies	Source	HTN
CaveDiving	8	3	26	IPC	little
ChildSnack	6	0	8	IPC	most
Citycar	7	0	25	IPC	little
Floortile	7	4	18	IPC	most
Gift-Giver	3	0	2	Personal	complete
Goldminer	3	1	4	Personal	complete
Grid	3	1	4	IPC	most
Parking	4	0	20	IPC	little
Rescue Robot Robby	4	3	3	Personal	complete
Transport	3	2	9	IPC	most
VisitAll	1	1	0	IPC	complete

The following graphs were generated automatically based on data obtained from the previous algorithms and represent how actions are related with each other by the use of predicates. Actions are represented by boxes and predicates by ellipses. The actions are connected based on which predicates they have as preconditions and effects. As an example using brackets as boxes and parentheses as ellipses, *[action-a]* $\rightarrow$ *(pred)*$\rightarrow$ *[action-b]* means that *action-a* have *pred* as an effect that appears as a precondition of *action-b*, which creates a dependency pattern between the actions, requiring

action-a to be applied when *pred* is missing. The other possible case is to have *[action-a]* $\leftrightarrow$ *(pred)*, which means we may need to apply *action-a* zero or more times to achieve *pred* with an specific value, which creates a swap pattern for this action. Swap patterns usually involve one action that moves an agent or object in the scenario, but may appear as several specific actions, as it happens in the Floortile domain where the movement action is broken into each direction. The parameters are also taken in consideration when making nodes, otherwise conflicts could appear.

5.4.1 CaveDiving

A group of divers have a set of tasks to be fulfilled in an underwater cave. They can carry 4 tanks of air and are able to drop tanks and take photos. The dropped tanks can be used by other divers in future expeditions, since some actions consume air and because of decompression at the end they are limited to one trip each. Some divers have no confidence in others work dropping the tanks correctly and will refuse to work after them. The confidence and decompression act as restrictions of this domain.

In this domain the sub-problems are not separated at first glance, as seen in Figure 5.4, having several edges connecting the tank preparation with the photograph and decompression tasks. This happens because of the nature of the problem, agents need to change their tanks in order to keep taking pictures underwater. The high number of dependencies involved makes the HTN generated too complex, which implies that methods for this domain must take several points in consideration before starting any task decomposition. One of the many dependencies of this domain, the *hire-diver* with *prepare-tank* dependency, is in Listing 5.6.

5.4.2 ChildSnack

Plan how to make and serve sandwiches for a group of children. The restriction appear as the amount of trays available and with gluten free or not ingredients to be used to serve allergic children among the group. Note that a kid without allergy may consume a total or partial gluten free sandwich. The goal consists of all children being served.

Figure 5.5 shows that any of the two actions related with sandwich making (*make_sandwich_no_gluten* and *make_sandwich*) must happen before *put_on_tray* can have the precondition *at_kitchen_sandwich* satisfied. In order to serve a sandwich the precondition *ontray* must be satisfied, *no_gluten_sandwich* must also be satisfied for gluten free sandwiches. The tray must be in the kitchen or in the same place as the child in order to put or serve a sandwich, respectively. Note that this domain only contains operators with the dependency pattern. This type of domain have actions with a clear dependency and the problem is not in the method construction itself, but in the description process in which human errors could occur. The current HTN obtained fails to see where to start, which would be moving the tray to the kitchen before making a new sandwich according to the

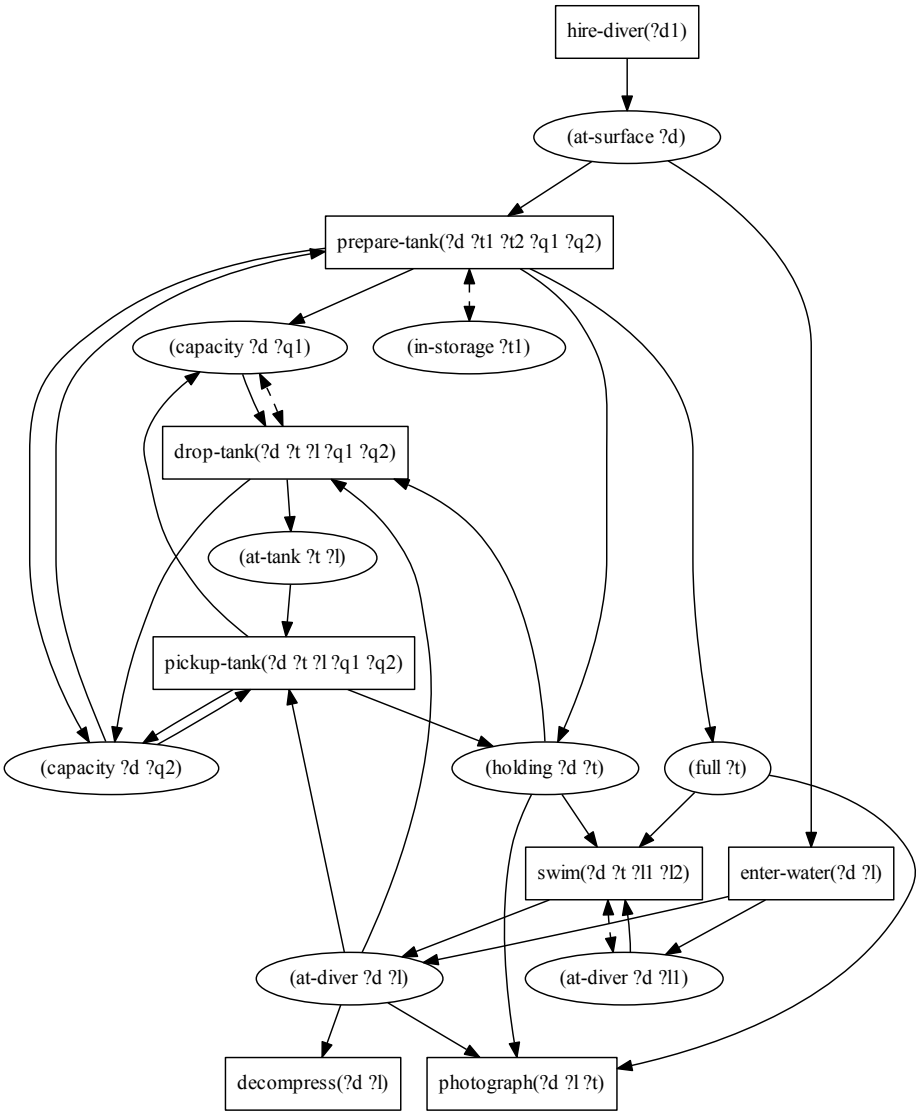

Figure 5.4 – CaveDiving operator patterns with 3 swaps and 26 dependencies.

child it wants to attend, allergic or not. Yet all methods were generated according to the dependencies found, which requires little work to fix. A few of the HTN methods generated are in Listing 5.7. It is possible to see the dependencies *make_sandwich_no_gluten* with *put_on_tray* and *move_tray* with *serve_sandwich_no_gluten*.

5.4.3 Citycar

The domain contains a traffic simulation in which roads must be build in order to help traffic flow. The roads can be build over the edges of a graph that represents a city, nodes represent junctions. Cars must reach their final position but can only move between empty roads. Since there is a limited

```
(:method (dependency_hire-diver_before_prepare-tank ?d1 ?d ?t1 ?t2 ?q1 ?q2)
   dependency_hire-diver_before_prepare-tank_satisfied
  (
    (diver ?d1) (diver ?d) (tank ?t1) (tank ?t2) (next-tank ?t1 ?t2)
    (quantity ?q1) (quantity ?q2) (next-quantity ?q1 ?q2) (at-surface ?d)
  )
  (
    (dependency_swap_capacity_until_capacity_before_prepare-tank ?d ?q1 ?t1 ?t2 ?q2)
  )
)
(:method (dependency_hire-diver_before_prepare-tank ?d1 ?d ?t1 ?t2 ?q1 ?q2)
   dependency_hire-diver_before_prepare-tank_unsatisfied
  (
    (diver ?d1) (diver ?d) (tank ?t1) (tank ?t2) (next-tank ?t1 ?t2)
    (quantity ?q1) (quantity ?q2) (next-quantity ?q1 ?q2) (not (at-surface ?d))
  )
  (
    (!hire-diver ?d1)
    (dependency_swap_capacity_until_capacity_before_prepare-tank ?d ?q1 ?t1 ?t2 ?q2)
  )
)
```

Listing 5.6 – *hire-diver* with *prepare-tank* dependency HTN methods obtained for CaveDiving using JSHOP description.

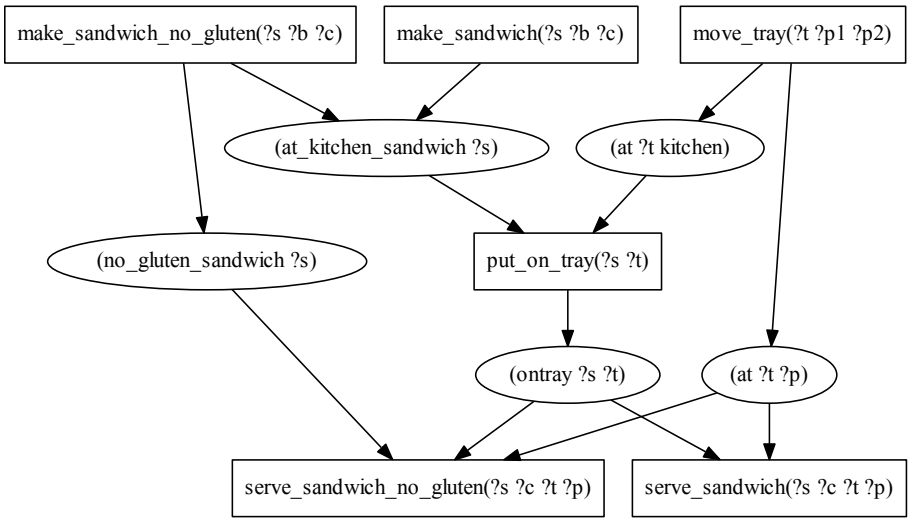

Figure 5.5 – Child Snack operator patterns with 0 swaps and 8 dependencies.

number of roads available, it is possible to remove empty roads and place somewhere else in order to fulfill the goal.

The Citycar domain appears to have one main HTN line, as shown by Figure 5.6, starting and moving the car until it is not possible anymore, which requires roads to be built, in straight or diagonal fashion, and connected. Connected roads can be destroyed to be reused in other places. The process repeats (move, destroy, build) until the car reaches its destination. For the same reasons of

```
(:method (dependency_put_on_tray_before_serve_sandwich_no_gluten ?s ?t ?c ?p)
   dependency_put_on_tray_before_serve_sandwich_no_gluten_satisfied
  (
    (sandwich ?s) (tray ?t) (child ?c) (place ?p)
    ( allergic_gluten ?c) (waiting ?c ?p) (ontray ?s ?t)
  )
  ( (!serve_sandwich_no_gluten ?s ?c ?t ?p) )
)
(:method (dependency_put_on_tray_before_serve_sandwich_no_gluten ?s ?t ?c ?p)
   dependency_put_on_tray_before_serve_sandwich_no_gluten_unsatisfied
  (
    (sandwich ?s) (tray ?t) (child ?c) (place ?p)
    ( allergic_gluten ?c) (waiting ?c ?p) (not (ontray ?s ?t))
  )
  (
    (dependency_make_sandwich_no_gluten_before_put_on_tray ?s ?b ?c ?t)
    (!serve_sandwich_no_gluten ?s ?c ?t ?p)
  )
)
(:method (dependency_move_tray_before_serve_sandwich_no_gluten ?t ?p1 ?p2 ?s ?c ?p)
  dependency_move_tray_before_serve_sandwich_no_gluten_satisfied
  (
    (tray ?t) (place ?p1) (place ?p2) (sandwich ?s) (child ?c)
    (place ?p) ( allergic_gluten ?c) (waiting ?c ?p) (at ?t ?p)
  )
  ( (!serve_sandwich_no_gluten ?s ?c ?t ?p) )
)
(:method (dependency_move_tray_before_serve_sandwich_no_gluten ?t ?p1 ?p2 ?s ?c ?p)
  dependency_move_tray_before_serve_sandwich_no_gluten_unsatisfied
  (
    (tray ?t) (place ?p1) (place ?p2) (sandwich ?s) (child ?c)
    (place ?p) ( allergic_gluten ?c) (waiting ?c ?p) (not (at ?t ?p))
  )
  (
    (!move_tray ?t ?p1 ?p2)
    (!serve_sandwich_no_gluten ?s ?c ?t ?p)
  )
)
```

Listing 5.7 – A few HTN methods obtained for ChildSnack using JSHOP description.

ChildSnack, the problem is where to start in the HTN structure. Since the process must be repeated a number of times it is hard to infer which path would be interesting to decompose first.

5.4.4 Floortile

A group of robots must paint the floor using different colors. The robots can move in four directions, paint what is in front or behind them, change paint color and cannot walk over the painted tiles. Chessboard configurations pose a hard problem, requiring robots to constantly change paint color since it is not possible to paint all tiles with the first color and paint the remaining tiles with the second color.

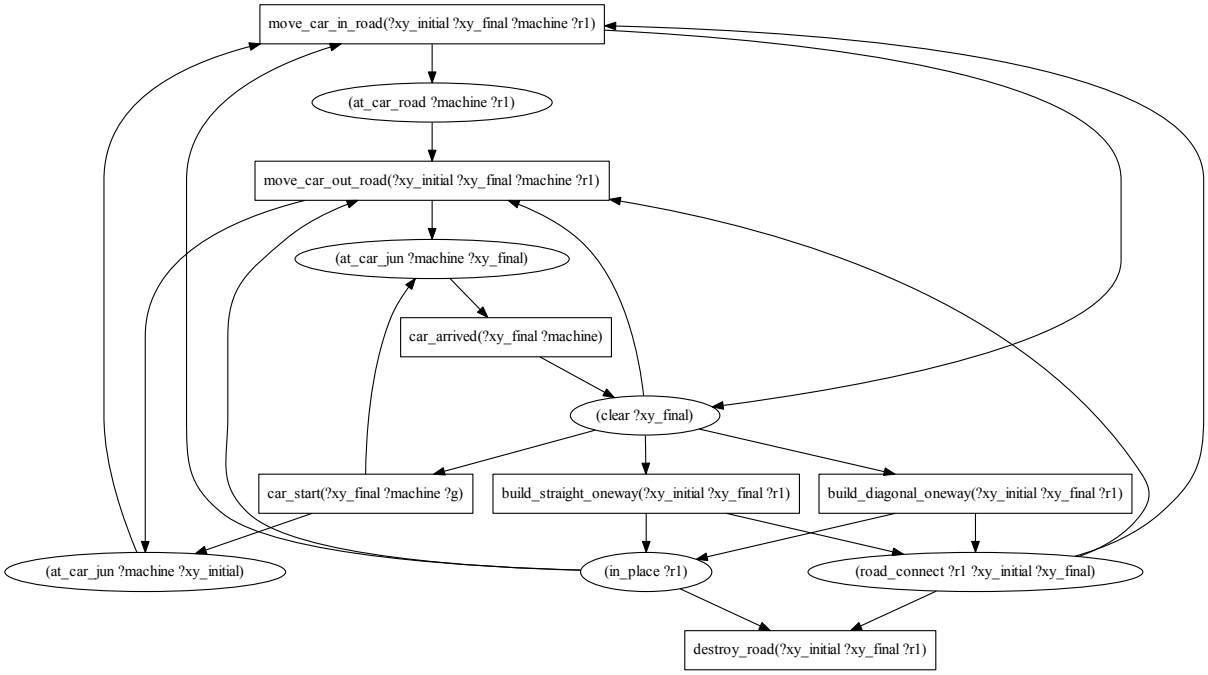

Figure 5.6 – Citycar operator patterns with 0 swaps and 25 dependencies.

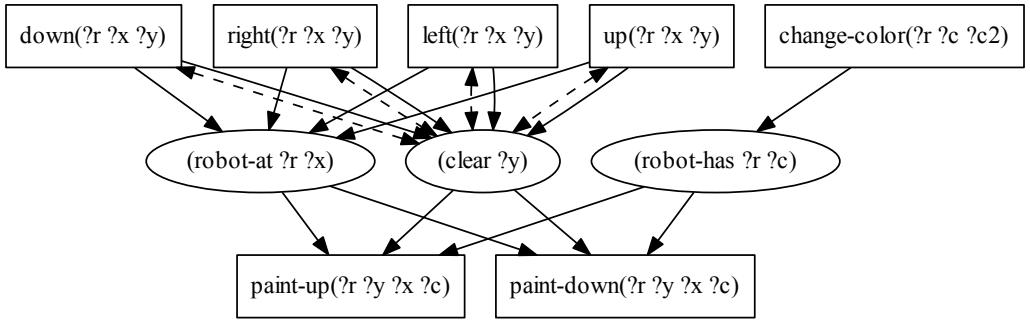

Figure 5.7 – Floortile operator patterns with 4 swaps and 18 dependencies.

In this domain we see several movement actions modifying the robot position: *up*, *down*, *left* and *right*. Each action have a different direction restriction, but all require the new position to be *clear*, which may require several executions to reach the desired position, defined by the predicate *at-robot*, as shown by the Figure 5.7. The movement is one sub-problem of this domain, requiring a position swap with the 4 direction based actions to be solved. The other sub-problem of this domain is the color used by the robot to paint each floor tile. Since the color required is know based on the goal state it is simple to assume that the *change-color* action will change the current color of the robot with the next one required. With the correct color selected and positioned adjacent the floor tile to be painted the robot can use *paint-up* or *paint-down* to obtain the desired color for each floor tile. Since this domain have two different ways to paint any tile the automated solution saw no problem

```
(:method (swap_clear_until_robot-at ?y)
  swap_clear_base_robot-at
  ( (robot-at ?y) ) nil
)
(:method (swap_clear_until_robot-at ?y)
  swap_clear_until_robot-at_using_up
  (
    ( clear ?current ) (not ( clear ?y))
    (up ?current ?intermediate ) (not ( visited_clear ?intermediate ))
  )
  (
    (!up ?current ?intermediate )
    ( !!visit_clear ?current )
    ( swap_clear_until_robot-at ?y)
    ( !!unvisit_clear ?current )
  )
)
(:method (swap_clear_until_robot-at ?y)
  swap_clear_until_robot-at_using_down
  (
    ( clear ?current ) (not ( clear ?y))
    (down ?current ?intermediate ) (not ( visited_clear ?intermediate ))
  )
  (
    (!down ?current ?intermediate )
    ( !!visit_clear ?current )
    ( swap_clear_until_robot-at ?y)
    ( !!unvisit_clear ?current )
  )
)
```

Listing 5.8 – A few HTN methods obtained for Floortile using JSHOP description.

in using only one. It selects the desired color, moves and paint down. Errors may occur when asked to paint the upper row, which requires the agent to paint up. Although simple for a human to see this limitation during debugging we believe this situation also occurs when humans are describing the HTN, an excessive use of an action. The base, up and down cases of the position swap are in Listing 5.8

5.4.5 Goldminer

In the Goldminer domain one or more miners must pick gold in a grid scenario and drop it in a deposit location. The domain contains three operators: *pick* gold, *drop* gold and *move* miner. Most goals of this domain define that all gold objects must be at the deposit location. Figure 5.8 shows the relationship between the operators to compose tasks. The Goldminer domain generate five important tasks. The travel task, based on the swap operator *move*. The picking task, based on the dependency between move and pick, since the miner must be at the same location as the gold. The picking task uses the swap previously defined before applying pick. The dropping task, analogous to the picking task. The pick to drop task, which is based on the dependency between pick and drop, since the miner

must have the gold before dropping it. Such task uses the picking task followed to a drop task, since the miner must move, pick, move and drop gold objects. The final task is to decide the free-variable miner from the pick to drop task during run-time. This method will try several miners until one of them successively finishes the task. This is required because we do not check if there is at least one miner that reaches all gold positions, since the grid may contain obstacles.

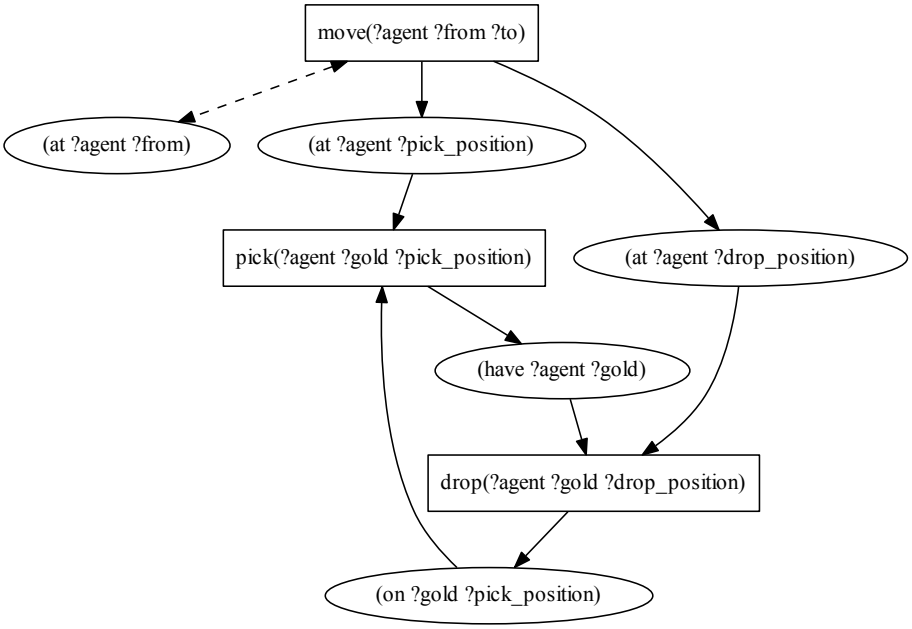

Figure 5.8 – Goldminer operator patterns with 1 swap and 4 dependencies.

5.4.6 Grid

An agent must reach a goal position in a 2D grid space. The agent is able to move, pick keys and unlock doors. The cells in the grid may be open, closed or not connected to the adjacent cells. In order to move the destination cell must be adjacent and open. Closed cells act like doors that require specific keys to be open with.

The Grid domain have three tasks, inside the boxes of Figure 5.9. The first task is to move in the grid, which requires zero or more applications of the action *move* in order to achieve a certain cell, defined by the predicate *at-robot*. The second task is the picking, the agent moves towards an specific key location before picking it. The third task is the unlocking, the agent with the key moves to an specific door that requires that key to be unlocked. The process is repeated until sufficient doors are opened, creating a path of open cells between the robot and the goal cell. We can achieve most of the HTN expected to solve such domain. The problem found is how to discover that multiple times the agent will have to pick and unlock doors only to obtain access to other keys that contribute towards the goal position. The current HTN obtained provides a single journey: move zero or more times to

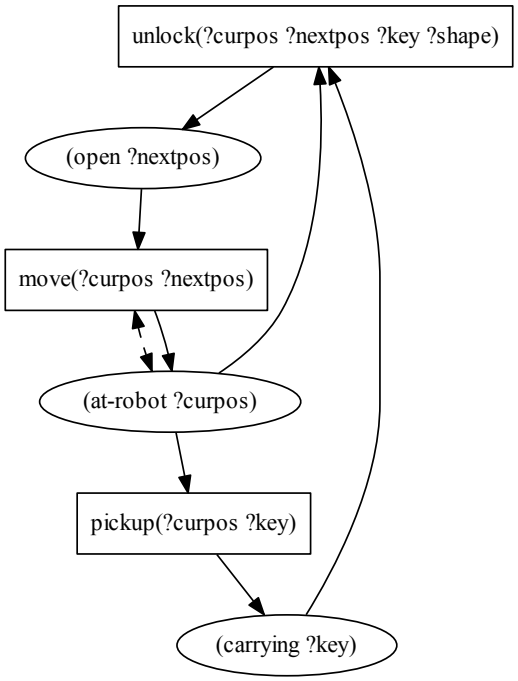

Figure 5.9 – Grid operator patterns with 1 swap and 4 dependencies.

```
1  (:method (dependency_pickup_before_unlock ?curpos ?key ?nextpos ?shape)
2     dependency_pickup_before_unlock_unsatisfied
3     (
4        (place ?curpos) (place ?nextpos) (conn ?curpos ?nextpos)
5        (shape ?shape) (lock-shape ?nextpos ?shape)
6        (key ?key) (key-shape ?key ?shape) (not (carrying ?key))
7     )
8     (
9        (dependency_swap_at-robot_until_at-robot_before_pickup ?curpos ?key)
10       (dependency_swap_at-robot_until_at-robot_before_unlock ?curpos ?nextpos ?key ?shape)
11    )
12 )
```

Listing 5.9 – Single journey HTN method obtained for Grid using JSHOP description.

obtain key if the agent does not have it and move zero or more times in order to unlock a door with a key being carried. The single journey method is in Listing 5.9.

5.4.7 Parking

This domain involves cars parked on a street with N curb locations, where cars can be double-parked but not triple-parked. The goal is to find a configuration of parked cars that satisfy the triple-parked limitation, by driving the cars between the curb locations.

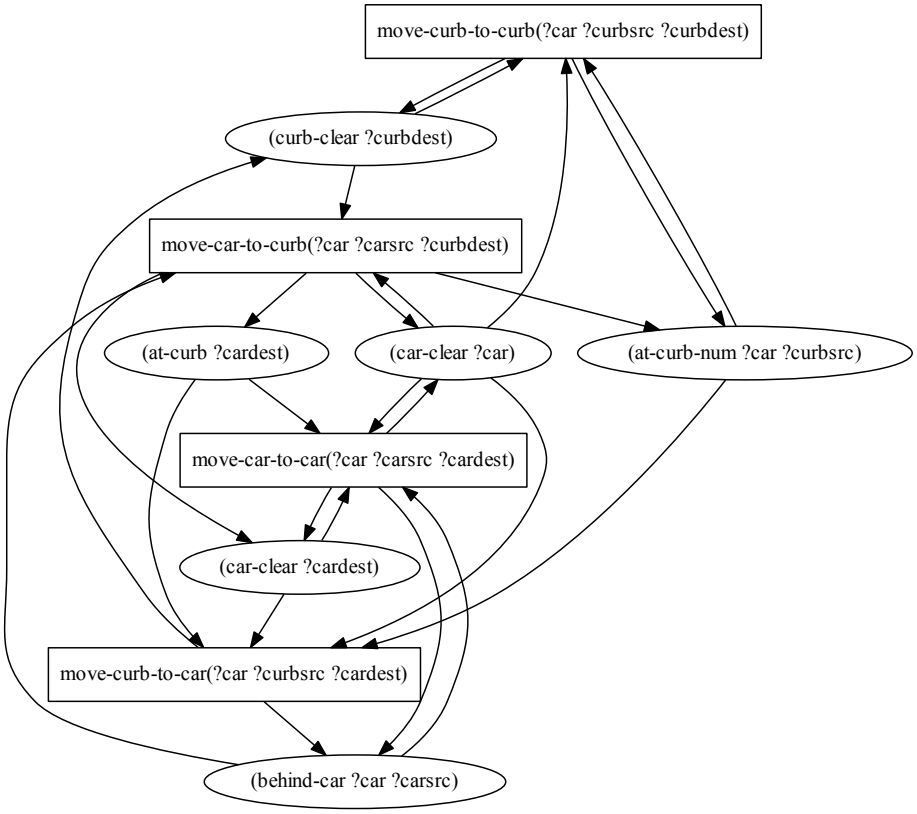

Figure 5.10 – Parking operator patterns with 0 swaps and 20 dependencies.

```
1 (:method (unify_dependency_move−car−to−car_before_move−curb−to−curb ?car ?curbdest)
2   unify_dependency_move−car−to−car_before_move−curb−to−curb
3   ( (car ?car) (car ?carsrc) (car ?cardest) (curb ?curbsrc) (curb ?curbdest) )
4   ( (dependency_move−car−to−car_before_move−curb−to−curb ?car ?carsrc ?cardest ?curbsrc ?curbdest) )
5 )
```

Listing 5.10 – Unification HTN method obtained for Parking using JSHOP description.

Parking domain have several operators with symmetric dependency, which suggests that effects of one action can be undone by another. It is not a simple task to break into sub-problems since most predicates also act as preconditions of other dependency operators. Such predicates act as bridges, connecting the actions in the same hierarchy. As previous domains demonstrated the main problem is the starting point, as no action works as a setup to start the process as the *hire-diver* in CaveDiving. The program decides to start with *move-curb-to-curb* or with *move-car-to-car* based on the goal *behind-car* or *at-curb-num* it wants to achieve, but it is clear that other actions can also be used to achieve such goals based on Figure 5.10. Since the predicates of the goal state are not enough to decide the values of some variables for the Parking domain an explicit Free-variable method is created to decide satisfiable values, as seen in Listing 5.10.

5.4.8 Transport

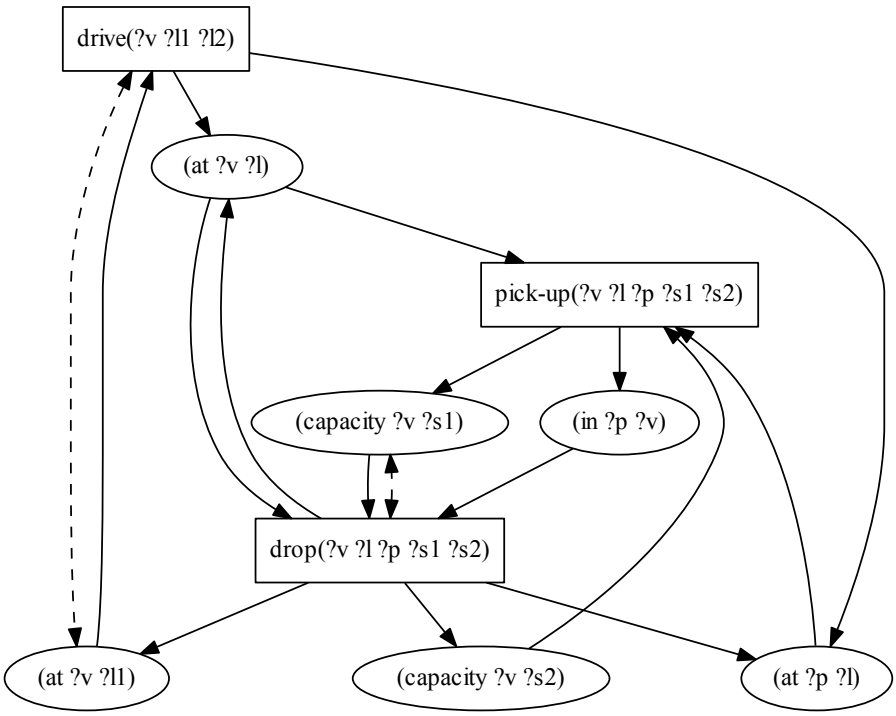

Figure 5.11 – Transport operator patterns with 2 swaps and 9 dependencies.

A variation of the logistics domain, the goal is to move packages between places using a set of vehicles. In the Transport domain we can see the position of the vehicles and the objects playing the main role in Figure 5.11. The *drive* action is identified as a swap operator, which satisfies the position required to apply the action *pick-up*. Items that have been picked can be dropped, which can be used to control the capacity of vehicle. This domain can be seen as two sub-problems, the first as the drive to location task, and the second as to pick and drop items. The HTN ignores pick-up while trying to solve *capacity* or *at* swaps, can be fixed by a human with small effort.

5.4.9 VisitAll

An implementation of the Traveling Salesman Problem (TSP), where an agent must visit all cities/places without repeating previous roads. This is a simple domain, with one operator. The complexity lies in the problem description, not in the domain, and a single swap is enough to represent how to solve it, Figure 5.12. Every location to be visited will call this task, which will apply zero or more moves to reach all locations. The ordering the locations must be visited to avoid repetition is the complex problem. We can achieve full HTN support as expected for this domain, since it is

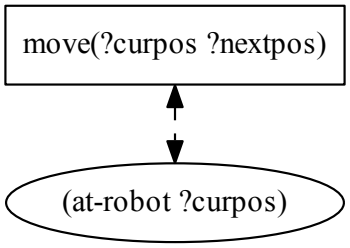

Figure 5.12 – VisitAll operator patterns with 1 swap and 0 dependencies.

one of the simplest cases of the swap operator pattern. The HTN generated is in Listing 5.11, with two extra operators to avoid repetition and two swaps to solve the two goals present for problems of this domain. The first goal is to have visited every location, this goal is related to the *swap_at-robot_until_visited* task. The second goal is to be *at* a certain location, this goal is related to the task *swap_at-robot_until_at-robot*.

```
(defdomain grid-visit-all(
  (:operator (!move ?curpos ?nextpos)
    (
      (place ?curpos) (place ?nextpos)
      (at-robot ?curpos) (connected ?curpos ?nextpos)
    )
    ( (at-robot ?curpos) )
    ( (at-robot ?nextpos) ( visited ?nextpos) )
  )
  (:operator ( !!visit_at-robot ?curpos)
    nil nil ( ( visited_at-robot ?curpos) )
  )
  (:operator ( !!unvisit_at-robot ?curpos)
    nil ( ( visited_at-robot ?curpos) ) nil
  )
  (:method (swap_at-robot_until_at-robot ?curpos)
    swap_at-robot_base_at-robot
    ( (at-robot ?curpos) ) nil
  )
  (:method (swap_at-robot_until_at-robot ?curpos)
    swap_at-robot_until_at-robot_using_move
    (
      (at-robot ?current ) (not (at-robot ?curpos))
      (connected ?current ?intermediate )
      (not ( visited_at-robot ?intermediate ))
    )
    (
      (!move ?current ?intermediate )
      ( !!visit_at-robot ?current )
      (swap_at-robot_until_at-robot ?curpos)
      ( !!unvisit_at-robot ?current )
    )
  )
  (:method ( swap_at-robot_until_visited ?curpos)
    swap_at-robot_base_visited
    ( ( visited ?curpos) )
    nil
  )
  (:method ( swap_at-robot_until_visited ?curpos)
    swap_at-robot_until_visited_using_move
    (
      (at-robot ?current ) (not (at-robot ?curpos))
      (connected ?current ?intermediate )
      (not ( visited_at-robot ?intermediate ))
    )
    (
      (!move ?current ?intermediate )
      ( !!visit_at-robot ?current )
      ( swap_at-robot_until_visited ?curpos)
      ( !!unvisit_at-robot ?current )
    )
  )
))
```

Listing 5.11 – HTN obtained for VisitAll using JSHOP description.

6. RELATED WORK

We start with hierarchical planning algorithms that differ from task to state as goal. After that we explore works that have the same goal as ours, to obtain a HTN description from a simpler classical planning description. The difference lies in the plan traces and annotations used to help the system to obtain the HTN description. Some of them focus on clustering common sequences of operators that repeat accross the given examples A recent survey about planning and machine learning is also mentioned, as several planners are including a learning component to increase domain knowledge and how domain knowledge still poses as a hard problem to be found correctly.

6.1 Goal Decomposition with Landmarks

Goal Decomposition with Landmarks(GoDeL) [20] is a hierarchical planning system that can search with partial domain knowledge. That means the classical operators are enough to find a plan, but with more knowledge available the plan can be found faster. When enough knowledge is described (complete set of methods) the performance is expected to be the same as HTN. When no knowledge is described (only operators) the performance is expected to be the same as classical planners. This algorithm, thus, allows a designer to incrementally improve the planning description, instead of requiring that all domain knowledge to be specified before planning.

6.2 The Landmarks

The landmarks are sequences of ground predicates that must be true in every solution, therefore they can be seen as intermediate goals impossible to avoid. GoDeL uses a landmark system from the LAMA planner to find such intermediate goals. The LAMA planner [18] is a classical planner with heuristic forward search. It is built upon the Fast Downward planning system [8]. The core idea is to infer landmarks to be used as heuristic. The system is based on three different modules, a translator that deals with the PDDL description, a knowledge compilation module that takes care of which structures are required for the problem and the search engine that operates on those structures. The modules are different programs sequentially called to solve the planning problem. Landmark identification is useful for planning as it can achieve both the facts and the order they must be true, although it is important to remember this is a heuristic, more time is required to find more landmarks.

6.3 HTN with goal states

Instead of describing the goal as a state or task the goal is represented as a network. The goal network represents a partially ordered set of goals. In case of a single goal the problem is the same

as in classical planning. The methods are used to provide guidance based on relevance, which means the effects match part of the goal. An example would be the decomposition that requires *clean(car)*, now we search for operators and methods that can lead to this goal, such as an operator or method *clear* that have *clean(X)* as an effect, otherwise we must explore using brute-force.

Experiments with different amounts of knowledge, were created to see how GoDeL would act. Three different sets of knowledge were created for Logistics, Blocks-World and Depots domains. The complete set of methods always performed better than the medium, and the medium better than the low set. The sub-goal inference is extremely important to save time, giving a hint of what effect to pursuit. The landmark system is the central point of future study, being able to exploit knowledge and infer sub-goals for methods not described.

Our work also tries to minimize the gap of classical and HTN planning. Both approaches start with a planning instance from a classical description. We want to automatize the method construction process while GoDeL tries to take advantage of the methods already given. Both solutions require an expert to optimize the output by adding the missing methods to reach a goal state in some cases.

6.4 HTN-MAKER

HTN-MAKER [10] implements a mechanism to extract methods from a STRIPS domain with plans and task definitions. The result is a sound but not complete solution, with some problems beyond what is possible to solve with the obtained HTN. Such unsolvable problems are expected to exist due to the lack of plan examples that cover all scenarios. The authors chose a structure to represent what the tasks must accomplish with preconditions and effects. The goal is to find the corresponding method for each given task based on several STRIPS problems and plan traces. The algorithm tries to cluster operators to generate methods, and then repeat the process considering the previously defined methods to generate high-level methods. The amount of generated methods and the infinitely recursive manner HTN can deal with decomposition make a challenge to find the smallest set of methods required. Methods can be pruned based on preconditions and recursive solutions are still considered a problem. The author cite the appropriate level of generality as a problem, as it is impossible to know how many methods are needed to solve anything for some domains.

Our work shares the same purpose of HTN-MAKER, receive a classical description and output a HTN description of the same domain. The only difference is what HTN-MAKER requires as extra inputs: a set of solved plans and task definitions. Both extra inputs are related to our approach. The set of solved plans are similar to our operator patterns, used to guide the search for useful methods. The task definitions guide the goal to task mechanism, selecting which methods are candidates to reach a predefined goal.

6.5 Macro-FF

Macro-FF [4] extends FF planner [9] with the use of common sequence of operators, which are called macros. Each macro acts as one operator that achieve the same effect of applying the entire sequence at once. A system analyzes a domain and a set of its solved problems in order to learn macros from. Once analyzed the system abstracts the information found to create an enhanced domain to solve the actual problems for the domain. Once generated, the macros are filtered to maintain only the most promising ones for later use. The paper proposes two different approaches, one simpler that requires static facts and is limited to STRIPS, and another more complex solution to handle more than the STRIPS subset using a different representation for the macros. The authors mention that future work may target HTNs, and very little work about learning HTNs has been conducted (until the publication of its paper [4] in 2005).

The common sequences of Macro-FF are related to the operator patterns we explored. The problem of macros is the lack of relations with other macros, which requires a method-like construction. Such macros can help the planner to achieve the goal state much faster, but require a lot of solved problems and a smart filter to actually guarantee such speed-up. We avoided solved problems for this reason, without a smart filter the planner would have to consider not only the original actions, but several macros when searching for applicable instances at the current state.

6.6 Machine Learning

Machine learning is being used in the planning context to discover domain knowledge, as macro-operators or hierarchical task networks. In fact a renewed interest in machine learning lead to the learning-based planners track at the IPC in 2008. There are two problems that can be solved with learning methods according to Jiménez et al. [12], discover the action model and the domain knowledge.

We are more interested in the domain knowledge. One of the open problems discussed in the review is how to effectively describe knowledge for automated planners. And this is one of the important points we want to address. Domain knowledge is still part of the domain, in order to separate both domain and common knowledge we need to understand which sub-problems exist and how they are related to domain knowledge. Only then we will be able to find an interesting description that suits our needs. We cannot ignore action model learning as some domains may have more than one way to represent the same transitions, with one being much easier to relate to a common knowledge to be used as our strategy. It is also important to note that any errors in the learning process may lead to an inefficient planner, the greedier the planning algorithm the bigger impact it will have.

6.7 Remodelling

Following Macro-FF we can also think about remodelling the domain, removing the original operators and replacing them with macro versions. Recently this idea was proposed by Alhossaini et al. [1], using Macro-FF simpler mechanism to generate macros with the help of a feature selection mechanism to replace the operators. This mechanism requires more study to achieve better control. The authors cite time as one drawback, as it takes a long time to process all the macros possible for several domains. They consider the current time, even with parallelization and offline execution, to be considerably large.

The removal of the original operators is not different from what we do, putting the operators behind tasks that map to methods that will use such operators when applicable. Operators that are not part of any subtask list and do not appear in the top-level tasks can be considered removed from the domain. HTN descriptions also ignore certain operators by focusing one task at a time, which could be considered a domain remodelling case. We avoid the long processing time since we have no solutions to mine data from, only the operators and predicates used by them. Most domains have a small number of operators and predicates, at least when compared with the actions of enough solutions to cover several cases, which makes our processing time very small.

7. CONCLUSION

In this dissertation, we have developed a mechanism to automatically generate HTN tasks and methods from unannotated classical planning domains. The main goal was to identify and use operator patterns to create common methods related to the goals of the specific planning problems. The pattern graphs obtained even in cases in which it was not possible to obtain a full HTN description helped to visualize how the operators relate to each other in each domain. Such a mechanism has an impact on the efficiency of and ease with which HTN planning domains are designed. While our tool fails to generate the full HTN methods for some domains, it provides a description that requires only minor modifications in order to solve the classical problems in an HTN fashion. Thus, some exceptional domain elements require an expert to define their relation with the other operators manually describing their usage. Some methods generated by our tool are not currently used by the top level methods, nevertheless, they can be useful to a domain designer, who may want to take advantage of them in cases our approach did not connected them to a top-level task.

We focus most of our work on the operators and which patterns they match in order to relate them to a correct method composition structure and for the second part we face the tasks and their relations. The tasks play the main role in HTN, and details about the goal to task conversion process are the main focus of future research. Our approach only searches permutations of the top level of tasks without interleaving, which already solves some of the problems encountered during our tests.

More research is required to identify potential solutions in which it may be possible to take advantage of unordered subtasks. We must also consider that one task may solve several goal predicates when using certain objects for each free-variable, which means those objects should be preferred over others to minimize the plan length and speed-up search. Since we reuse preconditions when composing methods from bottom-up, we may end up with several repeated tests of the same predicate, even when we know that no operator has been applied and we are using a total-order decomposition. We maintained those preconditions in order to obtain a solution even when using interleaved solutions in the future, although a human or an automatic process could generate a valid smaller/compressed version of our generated domains.

BIBLIOGRAPHY

[1] Alhossaini, M. A.; Beck, J. C. “Instance-specific remodelling of planning domains by adding macros and removing operators”. In: SARA, 2013.

[2] Blum, A. L.; Furst, M. L. “Fast planning through planning graph analysis”, *Artificial intelligence*, vol. 90–1, 1997, pp. 281–300.

[3] Bonnet, B.; Geffner, H. “HSP: Heuristic Search Planner”, 1998.

[4] Botea, A.; Enzenberger, M.; Müller, M.; Schaeffer, J. “Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators”, *Journal of Artificial Intelligence Research*, vol. 24, 2005, pp. 581–621.

[5] Fikes, R. E.; Nilsson, N. J. “STRIPS: A new approach to the application of theorem proving to problem solving”, *Artificial intelligence*, vol. 2–3, 1971, pp. 189–208.

[6] Fox, M.; Long, D. “Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning”. In: Proceedings of International Joint Conference on Artificial Intelligence, 2001, pp. 445–452.

[7] Ghallab, M.; Nau, D.; Traverso, P. “Automated planning: theory & practice”. Elsevier, 2004.

[8] Helmert, M. “The fast downward planning system”, *Journal of Artificial Intelligence Research*, vol. 26, 2006, pp. 191–246.

[9] Hoffmann, J. “FF: The fast-forward planning system”, *AI magazine*, vol. 22–3, 2001, pp. 57.

[10] Hogg, C.; Munoz-Avila, H. “Learning hierarchical task networks from plan traces”. In: Proceedings of the ICAPS-07 Workshop on AI Planning and Learning, 2007.

[11] Ilghami, O.; Nau, D. S. “A general approach to synthesize problem-specific planners”, Technical Report, DTIC Document, 2003.

[12] Jiménez, S.; De la Rosa, T.; Fernández, S.; Fernández, F.; Borrajo, D. “A review of machine learning for automated planning”, *The Knowledge Engineering Review*, vol. 27–04, 2012, pp. 433–467.

[13] Lawler, E. L. “The traveling salesman problem: a guided tour of combinatorial optimization”, *Wiley-Interscience series in discrete mathematics*, 1985.

[14] Lekavỳ, M.; Návrat, P. “Expressivity of STRIPS-like and HTN-like planning”. In: *Agent and Multi-Agent Systems: Technologies and Applications*, Springer, 2007, pp. 121–130.

[15] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; Wilkins, D. “PDDL-the planning domain definition language”, 1998.

[16] Nau, D.; Cao, Y.; Lotem, A.; Muñoz-Avila, H. “SHOP: Simple hierarchical ordered planner”. In: Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2, 1999, pp. 968–973.

[17] Pattison, D.; Long, D. “Domain Independent Goal Recognition”. In: STAIRS, Ågotnes, T. (Editor), 2010, pp. 238–250.

[18] Richter, S.; Westphal, M. “The LAMA planner: Guiding cost-based anytime planning with landmarks”, *Journal of Artificial Intelligence Research*, vol. 39–1, 2010, pp. 127–177.

[19] Russell, S.; Norvig, P.; Intelligence, A. “A modern approach”, *Artificial Intelligence. Prentice-Hall, Englewood Cliffs*, vol. 25, 1995.

[20] Shivashankar, V.; Alford, R.; Kuter, U.; Nau, D. “The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning”. In: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, 2013, pp. 2380–2386.