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

SYMBOLIC-GEOMETRIC PLANNING

MAURÍCIO CECÍLIO MAGNAGUAGNO

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

Advisor: Prof. Felipe Meneguzzi

Porto Alegre
2020

REPLACE THIS PAGE
WITH THE PRESENTATION
TERM

PLANEJAMENTO SIMBÓLICO-GEOMÉTRICO

RESUMO

Planejadores clássicos, com ações descritas com precondições e efeitos, criam uma forma de operar em modelos puramente simbólicos para encontrar planos que alcançam os objetivos de um agente. Planos encontrados por planejadores clássicos geralmente ignoram detalhes geométricos necessários para resolver problemas de movimento, tais como pegar um objeto ou evitar passagens estreitas. Esses detalhes incluem as dimensões do robô e de objetos. Em contrapartida, planejadores de movimento consideram apenas detalhes físicos, não objetivos ou partes simbólicas do estado. Ambos planejadores simbólicos e de movimento são necessários para resolver certos problemas, mas relações entre dados simbólicos e geométricos devem ser compartilhadas para evitar replanejamento de grandes porções do espaço de busca. Para lidar com este problema, nós precisamos de um planejador simbólico-geométrico que compartilhe dados e restrinja valores possíveis conforme o planejamento avança, para minimizar uso de memória e tempo de planejamento. Diferentes algoritmos de planejamento para esse tipo de planejamento híbrido foram desenvolvidos com o objetivo de combinar planejamento simbólico e geométrico usando planejadores prontos ou novas implementações. A maior parte dos planejadores híbridos compartilha informações entre as partes usando um conjunto fixo de símbolos, os quais limitam a quantidade de informação que pode ser compartilhada. Gerando esses símbolos durante o planejamento para a parte simbólica e os relacionando com objetos externos complexos (*containers*, *structs*, instâncias) na parte geométrica é possível simplificar a descrição simbólica enquanto exploram-se estruturas complexas e funções já disponíveis por bibliotecas externas, como as usadas por simuladores. Esse trabalho traz como contribuições a definição de anexo semântico, um mecanismo para compartilhar informação entre as partes simbólica e externa/geométrica de um planejador de redes hierárquicas de tarefas (HTN), e uma tabela de símbolo para objeto para manter detalhes externos fora da parte simbólica, enquanto capaz de computar com tais objetos externos através de funções e anexos semânticos.

Palavras Chave: robótica, planejamento simbólico-geométrico.

SYMBOLIC-GEOMETRIC PLANNING

ABSTRACT

Classical planners, with actions described with preconditions and effects, create a way to operate on purely symbolic models in order to find plans to reach an agent's goals. Plans found by classical planners often lack the geometric details required to solve motion problems, such as grasping an object or avoiding narrow passages. Such details include robot dimensions and object properties. Conversely, motion planners consider only physical details, not symbolic goals or parts of the state. Both symbolic and motion planners are required to solve certain problems, but relations between symbolic and geometric data must be shared to avoid replanning large portions of the search-space. In order to tackle this problem, we need a symbolic-geometric planner to share data and limit possible values as planning progresses, to minimize memory usage and planning time. Different planning algorithms for this type of hybrid planning have been developed to address the problem of combining geometric and symbolic planning by using off-the-shelf planners or new implementations. Most hybrid planners share information between the parts using a fixed set of symbols, which limits the amount of information that can be shared. By generating such symbols during planning for the symbolic part and relating them to external complex objects (containers, structs, instances) in the geometric part, it is possible to simplify the symbolic description while exploiting complex structures and functions already available in external libraries, such as the ones used by simulators. The contributions of this work include the definition of semantic attachments, as a mechanism to share information between symbolic and external/geometrical parts for Hierarchical Task Network (HTN) planning, a symbol to object table to keep external details hidden from the symbolic part, while able to compute with external objects using functions and semantic attachments, and a precondition reordering algorithm to improve planning time.

Keywords: robotics, symbolic-geometric planning.

CONTENTS

1. INTRODUCTION

In domains such as assistive robotics, self-driving cars and space exploration, autonomy is now a more important issue than hardware, as hardware is readily available at low cost while software must be developed for each application. Non-autonomous robots lack the capacity to make long-term decisions, working successfully while following a script that defines a single behavior until something unforeseen happens and then fail without notice [2]. There is a need for robots to be autonomous agents [80], using goals or tasks as guiding forces, perceiving the world around themselves, reasoning with an internal model of such world and acting in order to increase their chances of success.

Means-end reasoning is a key skill in autonomous agents, which must handle a model of the world to find which actions they have available to achieve their goals before execution starts. Within Artificial Intelligence, Automated Planning is a subfield interested in computing this sequence of actions based on a description of the world. Planning can be done using different techniques [6, 47, 63], but it is easier to classify planners according to the description they receive before the planning process can start. Classical planners use a declarative description of the actions as preconditions and effects that require a discrete world in order to operate over a finite set of symbols, creating intermediate states as search progresses. As more actions and symbols are added to the description, more computing time is required to solve bigger problems, and even the best classical planner will not process a large number of possible combinations in a timely fashion. Hierarchical Task Networks (HTN) can solve this issue, using an hierarchical structure to guide the search process. HTNs are more expressive and complex to describe than classical planning descriptions. Both approaches are completely separated from the real world, yet both operate over symbols and numbers that abstract objects and resources of the real-world. As the description gets more complex, the time to extend and maintain it with more details increases. To make the description process easier and more reliable, one can exploit external libraries to solve common problems, such as geometric constraints. With such geometric library, one can ask for a set of valid positions and only ask again if a new constraint is found to limit the number of valid positions and not risk computing infinitely large sets. To avoid computing multiple sets some approaches use a preprocessing or filtering stage in order to minimize search time. Such optimizations tend to be domain specific, which require a domain expert, and are prone to error, as it is not trivial to visualize how both planner and external solver are sharing data at run-time due to the amount of data in each state [20]. An interface for the two approaches is needed to have a generic solution, otherwise the robot takes a long time to plan an action sequence, and never executes these actions in a world that is already different than the one it reasoned about.

Symbolic-Geometric planners try to address this issue, some use mixed approaches with two off-the-shelf planners [39], while others follow different approaches, combining planner and geometric solver into a single algorithm [18]. Other planners solve one side of the problem before the other, incorporating geometric details into a firstly symbolic plan, and replanning when such details cannot be added. [77] The main problem of these approaches is how to better share newfound information across the two planners.

This thesis contributes towards the *Pró-Alertas* project which aims to promote basic and applied research on issues related to the prevention, mitigation and management of natural disasters. Certain areas require routine or continuous verification to obtain data and evaluate how variations over time may lead to a natural disaster. Some of these areas are of difficult access and require a device to be installed for data gathering purposes to prevent future natural disasters. During natural disaster management such areas may have people in need of help. Robotic agents can be applied to reach the data gathering points to collect samples and/or apply sensors to obtain the information required and alert other agents, humans or robots, about the risk. Such robots may be too complex to operate and require certain autonomy to complete their tasks. Part of the robotic autonomy requires planning actions before execution to better handle the available resources, such as battery. Automated planners usually lack the ability to make external calls, with some having great constraints when they have the ability, which limits the capacity of the planner to deal with complex scenarios that require optimized solvers for sub-problems, and no user input during planning. We believe that with a more robust planning system it is possible to conciliate external solvers and expert knowledge with simulation to better test and prepare for dangerous situations. In such situations one may want to interfere with the planning process to either save time by removing unnecessary paths to be explored and to force critical paths to reach and gather data first, while generating a sequence of actions that matches an expected safe behavior from the robot to coexist with other agents.

Our research aims to address three issues. First, we want to design an easy-to-use domain description formalism that seamlessly integrates symbolic task problem description with the geometric constraints inherent to robotics. Second, we want to develop algorithms to solve problems in such formalism in a way that limits the number of objects explicitly reasoned about, by adding such objects on demand, and ultimately processing time. Third, we want to be able to work with symbols in a symbolic layer and with complex data structures and numeric/geometric functions in an external layer, with an intermediate layer connecting symbols to numeric details. By separating symbolic and geometric parts we can have an easier to understand search-space and more maintainable planning descriptions, while being able to exploit fast off-the-shelf geometric solvers.

To evaluate our approach we compare planning time and description differences between known classical planners with numeric support and their domains with our HTN planner with semantic attachments. We also explore how our approach could be used in other domains previously not explored due to the amount of numeric details or external structures required, now accessible through an intermediate layer.

1.1 Overview of the contributions

During the course of the PhD we published all the contributions detailed in this thesis. We also published contributions that are not directly related to this work, but which still support the work here related to HTN planning research. We published the following papers about HTN planning:

- Maurício Cecílio Magnaguagno, Felipe Meneguzzi. Method Composition through Operator Pattern Identification [54]. In International Conference on Automated Planning and Scheduling (ICAPS), Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), 2017;
- Felipe Meneguzzi, Maurício Cecílio Magnaguagno, Munindar P. Singh, Pankaj R. Telang, Neil Yorke-Smith. GoCo: Planning expressive commitment protocols [61]. In Autonomous Agents and Multi-Agent Systems, 2018;
- Maurício Cecílio Magnaguagno, Felipe Meneguzzi. HTN Planning with Semantic Attachments [56]. In International Conference on Automated Planning and Scheduling (ICAPS), Hierarchical Planning Workshop, 2019;
- Maurício Cecílio Magnaguagno, Felipe Meneguzzi. Semantic Attachments for HTN Planning [57]. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), 2020.

We also published the following work about planning tools and visualizations:

- Maurício Cecílio Magnaguagno, Ramon Fraga Pereira, Felipe Meneguzzi. DOVETAIL - An Abstraction for Classical Planning Using a Visual Metaphor [58]. In Proceedings of the 29th International Florida AI Research Society Conference (FLAIRS), 2016;
- Maurício Cecílio Magnaguagno, Ramon Fraga Pereira, Martin Duarte Móre, Felipe Meneguzzi. WEB PLANNER: A Tool to Develop Classical Planning Domains and Visualize Heuristic State-Space Search [59]. In International Conference on Automated Planning and Scheduling (ICAPS), Workshop on User Interfaces and Scheduling and Planning (UISP), 2017;
- Maurício Cecílio Magnaguagno, Ramon Fraga Pereira, Martin Duarte Móre, Felipe Meneguzzi. Develop, Visualize and Test Classical Planning descriptions in your browser. In International Conference on Automated Planning and Scheduling (ICAPS), System demonstration, 2019;
- Maurício Cecílio Magnaguagno, Ramon Fraga Pereira, Martin Duarte Móre, Felipe Meneguzzi. Knowledge Engineering Tools and Techniques for AI Planning - Web Planner: A Tool to Develop, Visualize, and Test Classical Planning Domains. Book/Chapter 11, to be released in 2020.

1.2 Overview of the Thesis

This thesis is divided into six chapters. Chapter 2 introduces the key background on classical planning, hierarchical planning, temporal planning, motion planning, and coroutines. Readers familiar with such formalisms may safely skip reading this chapter. Chapter 3 presents our symbolic-geometric planning approach, introducing an intermediate layer to support external predicates and functions to access structures and libraries outside the planner. External elements are mapped to symbols using a symbol-object table to obtain maintanable elements during planning, removing non-symbolic elements from the domain description. The chapter closes with common use-cases of these constructions. Chapter 4 evaluates our approach via complex examples and experiments using common scenarios and compare against known classical and HTN planners. Examples include common grid-based problems, and motion and temporal planning problems. Chapter 5 compares our work with other approaches that include numerical/geometrical data to planning and how they share constraints found either by the symbolic or non-symbolic parts with the rest of the system. Finally, Chapter 6 discusses what we achieved and future work.

2. BACKGROUND

This chapter introduces key theoretical background that underpins this work. In Section 2.1 we introduce classical planning, the basic formalism used by planners. Classical planners became very fast due to specialized structures [6] and heuristic functions [7] guiding search towards solutions with less steps, which made possible for planners to solve more detailed problems with more actions available. Section 2.2 introduces Hierarchical planning, a planning approach that focuses on specific actions at each time based on an hierarchical representation, built on top of the basic elements from classical planning. Section 2.3 introduces temporal planners, such planners are specialized in scheduling problems that arise when action duration is taken into consideration during planning Motion planning is shown in Section 2.4, such planners are specialized in geometric constraints. Motion planners focus on movement of the planning agent itself or objects that such agent can interact with, picking and placing such objects into satisfiable positions and orientations. Finally, Section 2.5 introduces coroutines and generators as multitasking approaches to yield and resume control between routines while keeping persistent state between calls.

2.1 Classical Planning

Devising a plan of action to achieve one's goals [69] is a fundamental capability in autonomous agents. Classical Planning is based on the idea of finding a sequence of actions that satisfies a predefined goal. There are two important aspects about the planning environment that affect which techniques can be employed to pursue a goal. The first aspect is determinism: when the outcome of transitions is always the same, the environment is considered deterministic, otherwise, the environment is non-deterministic, with a set of outcomes with a known or unknown probability for each transition. The second aspect to be considered is the observability: some environments are fully observable, such as most board-games, while others are partially observable, such as real-world scenarios, or completely non-observable from the agent's point of view, as a robot with sensor failure. The complexity of the environment impacts the description and the performance of the planning system.

2.1.1 Formalization

A classical planning problem instance is defined by initial and goal states that encode properties of the objects in the world at a particular time. In order to achieve the desired goal one must respect the rules of the domain, which limit which transitions are valid. Such transitions are the domain operators and are defined as preconditions and effects. Preconditions and effects use predicates and free variables that, when unified with the current objects, enumerate the possible actions to be performed. During the planning process, states are tested to check if they satisfy the preconditions of actions. The action effects can be applied if the preconditions are satisfied, creating a new possible state. Preconditions are satisfied when the constraint formula (usually a conjunction of predicates) is valid at the current state the action is being applied. The effects contain positive and negative sets of predicates to be added and removed by actions, respectively, changing object properties of the current state. Once a state that satisfies the goal is reached, the path taken is the plan or solution, a finite sequence of actions [65].

Classical planning formalisms comprise the following elements:

Definition 1 (Terms) *Terms are symbols that represent objects or variables. We call **O** the finite set of objects available.*

Predicates represent object relations that can be observed or changed during planning. Predicates can also be seen as constraints between terms. When all terms of a predicate are objects we call it a ground predicate, otherwise the predicate contain at least one variable term and is a lifted predicate. Ground predicates are obtained from lifted predicates through Unification, which replace variable terms with objects.

Definition 2 (Predicates) *Predicates are defined by a signature **name** applied to a sequence of N **terms**, represented by t_n, **p** = $\langle$**name(p), terms(t_1, ..., t_n)**$\rangle$. We call **F** the finite set of facts, comprised of all ground predicates.*

Definition 3 (Unification) *When a predicate **p** have all its variable terms, variables(**p**) = t $\in$ terms(**p**) $\wedge$ t $\notin$ **O**, replaced by **O** objects, we have unified a lifted predicate to a ground one. One can enumerate several substitutions using the cartesian product between variable terms and **O**, such that variables(**p**) x O = {(v, o) | v $\in$ variables(**p**) $\wedge$ o $\in$ O}. Then {(v_0, o_0), (v_0, o_1), ..., (v_n, o_n)} is the set of replacements of **p** to obtain ground predicates.*

A state is a finite set of ground predicates that describes a world configuration at a particular time. Partial states may be used to represent only what we are interested in, in a closed-world assumption where we have full observability. Partial states may also be used to represent only predicates whose state is certainly know, in an open-world assumption, where we may lack certainty about which predicates are true or false.

Definition 4 (State) *States are represented by **S** = $\langle p_1, ..., p_n\rangle$, a set of ground predicates.*

States can be modified respecting constraints that describe when a modification is APPLICABLE, and the modification itself, described by APPLY.

Definition 5 (Applicable) *A set of ground predicates is considered applicable when it is contained in the current state, described by the function* APPLICABLE$(set, State) : set \subseteq State$.

Definition 6 (Apply) *Apply is a function that removes and adds distinct effect sets of ground predicates to create a new State from the current State, described by the function* $\text{APPLY}(a, State)$: $(State -$ *eff*$^{-}$*(a))* $\cup$ *eff*$^{+}$*(a).*

Each set of predicates to be used in APPLICABLE and APPLY calls can be generalized, using variable terms, to make an Operator. More complex preconditions and effects consider expressions, quantifiers and conditions instead of just set operations to obtain more expressive Operators. The Operators can be Unified with **O** to obtain the full set of possible Actions.

Definition 7 (Operator) *Operators are represented by a 4-tuple* ***o*** *=* $\langle$***name(o), pre(o), eff(o), cost(o)***$\rangle$*:**name(o)** is the description or signature of **o**; **pre(o)** are the preconditions of **o**, a set of predicates that must be satisfied by the current state for action **o** to be applied; **eff(o)** are the effects of **o**; The effects contain positive and negative sets,* ***eff***$^{+}$***(o)*** *and* ***eff***$^{-}$***(o)***, *that add and remove predicates from the state, respectively; **cost(o)** represents the cost of applying this operator, usually one or zero.*

Definition 8 (Action) *Actions are instantiated/ground operators obtained from the Unification process of Definition 3. During the planning process, each action a that is* $\text{APPLICABLE}(pre(a), State_n)$ *can create a new reachable State,* $State_{n+1} \leftarrow \text{APPLY}(a, State_n)$. *The finite set of actions available is called **A**.*

Classical Planning is goal-driven, which requires the description of a Initial state and a Goal state to plan for. The planner is responsible for finding a Plan, a sequence of Actions from **A**, that when applied to the Initial state will reach the Goal state.

Definition 9 (Initial state) *The Initial state is a complete State, in a closed-world, represented by **I*** $\subseteq$ ***F***, *which is defined by a set of predicates that represent the current state of the environment.*

Definition 10 (Goal state) *Goal state is a partial State represented by **G*** $\subseteq$ ***F***, *which is defined by a set of predicates that we desire to achieve by successfully applying the actions available.*

Each Planning Instance is made of a generic domain **D** and a specific situation within this domain to be solved, described by **I** and **G**. The solution is a Plan. Not all Planning instances can be solved, as some **G** may be unreachable based on **I** and **A**, which results in planning failure.

Definition 11 (Domain) *Domain brings all problem independent elements together in the tuple **D** =* $\langle$***F, A***$\rangle$.

Definition 12 (Plan) *Plan is the solution concept of a planning problem and is represented by a sequence of actions that when applied in a specific order will modify **I** to **G** in **D**,* $\pi = \langle \boldsymbol{a}_1, ..., \boldsymbol{a}_n \rangle$. *An empty plan solves **G*** $\subseteq$ ***I***.

Definition 13 (Planning instance) *Planning instance represented by the 3-tuple **P** =* $\langle$***D, I, G***$\rangle$, *planners take as input **P** and return either* π *or failure.*

2.1.2 Heuristic Function

The order in which states are evaluated may generate visible difference in resources and time required to solve a planning instance. The function which controls the evaluation order is called heuristic function and gives priority to promising intermediate states using a metric between reached and goal states. Such metric may only consider the expected distance (number of actions to apply) to reach a goal state or also the traveled distance (number of actions applied). A perfect heuristic function is as complex as planning itself. To actually take advantage of a heuristic function one must solve a relaxed version of the problem, ignoring certain details. As different domains exhibit different characteristics it is almost impossible to find a heuristic function that is both fast and accurate for all domains. Domain independent heuristic functions are necessary components of fast search-based domain independent classical planners and aim to close the gap between domain independent planners and specialized solvers.

2.1.3 Domain and Problem descriptions

In order to better describe and reuse environments, different languages were proposed, separating domain description from the implementation of planning algorithms. Stanford Research Institute Problem Solver (STRIPS) [25] is a planner recognized as being one of the first to propose a formalism to describe problem and domain, successors are referenced as STRIPS-like. *Planning Domain Definition Language* (PDDL) was created in 1998 with the goal of becoming a standard input for planners [60], to allow direct comparisons of efficiency between planning algorithms. With PDDL the *International Planning Competition* (IPC) became possible, in which different planning implementations compete in different tracks about resource usage and results obtained from the same proposed problems.

PDDL domains are usually limited to symbolic approaches, as very few planners support and are optimized for numeric problems. PDDL requires declaration of a finite set of objects, corresponding to the terms of Definition 1, for grounding purposes. The finite set of objects is a requirement to avoid a combinatorial explosion in the amount of possible actions, corresponding to Definition 8, and ground predicates, corresponding to Definition 2. Some objects may be the result of discretizations of the domain. The discretization process is used to convert continuous values into their discrete counterparts, just like exchanging size in metric units by small, medium and large. Limits between discrete parts are specific to each domain. A possible description of pathfinding is shown in Listing 2.1 as the Search domain, in which two predicates are used to represent where an agent is *at* and which positions are *adjacent*. The adjacencies remain constant as they are not part of any action effects while the position of a moving agent is updated by *move* actions. The problem in Listing 2.2 is a possible instance of the Search domain, in which one agent must traverse four positions to reach a goal state. Note that the domain designer does not need to care about the agent repeatedly moving between

```
(define (domain search) ; This is a comment
  (:requirements :strips :typing)
  (:predicates (at ?ag - agent ?p - position) (adjacent ?a ?b - position) )
  (:action move
    :parameters (?agent - agent ?from ?to - position)
    :precondition (and (at ?agent ?from) (adjacent ?from ?to) )
    :effect (and (not (at ?agent ?from)) (at ?agent ?to) ) ) )
```

Listing 2.1 – PDDL Search domain with single move action.

```
(define (problem pb1) (:domain search)
  (:objects ag1 - agent
            p0 p1 p2 p3 p4 - position)
  (:init (at ag1 p0)
         (adjacent p0 p1) (adjacent p1 p0)
         (adjacent p1 p2) (adjacent p2 p1)
         (adjacent p2 p3) (adjacent p3 p2)
         (adjacent p3 p4) (adjacent p4 p3) )
  (:goal (at ag1 p4) ) )
```

Listing 2.2 – PDDL Search problem with objects, initial and goal state.

adjacent positions, such as *p1* and *p2*, in an infinite loop. It is expected that the planner marks visited states or uses an approach that does not explore repeated states. Unlike basic search mechanisms, one can modify the problem description, or even the domain, to better represent the problem at hand, not having to deal with state representation. Different problems may result in the same description due to symbolic limitations that ignore geometric details using discretization, as seen when comparing the two maps of Figure 2.1 with our problem defined in Listing 2.2.

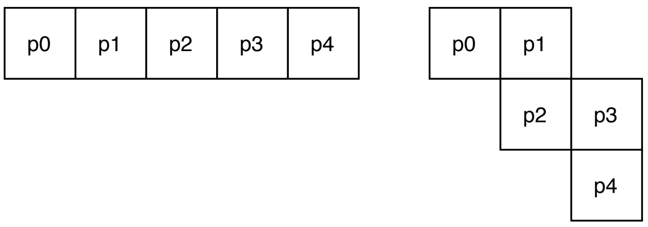

Figure 2.1 – Symbolic descriptions remove geometric details and may be interpreted in different ways. The same description may represent different maps. The domain designer becomes responsible for discretizing the environment in a readable and maintainable way.

A domain expert on the other hand may have a clue about which problems may appear and how to break such problems into parts that can be solved efficiently by common action sequences. In the next section, the flat structure of actions from classical planning domains is expanded to include methods that exploit the domain expert knowledge to better solve the problem by focusing on common subproblems of a domain.

2.2 Hierarchical Planning

With domain knowledge one knows which action sequences are frequently used to solve subproblems in specific domains [68]. Such sequences can be seen as recipes that, based on ingredients available and expert preferences, solve tasks in different ways. To exploit such domain knowledge about problem decomposition we shift from goal states to tasks. The goal state is implicitly achieved by the plan obtained from the tasks, just as a cake from a recipe. The ingredients available act as decision points, that can be used to create an hierarchy of decisions, while preferences appear as the order in which such decisions are considered. This planning formalism is called Hierarchical Task Network (HTN).

2.2.1 Formalization

The problem for hierarchical planning is defined with initial state and tasks. Each task corresponds to a starting node in an hierarchy, which comprises two types of nodes: primitive tasks that map directly to an operator; and non-primitive tasks that select a method that decomposes to subtasks. The process is repeated until only applicable primitive tasks remain, with states satisfying preconditions in the same way as classical planning.

HTN planning expands the elements of classical planning with:

Definition 14 (Task) *A task is represented by a signature* ***name(task)*** *applied to a sequence of N* ***terms*** *that act as parameters, forwarding ground values to be used by the task,* ***task*** *=* $\langle$***name(task), terms(***t_1***, ..., ***t_n***)***$\rangle$*.*

A set of tasks to be decomposed by an instance is called **T**. During each step of the planning process the first task is removed from **T** by SHIFT, and mapped by name to an operator (primitive task), equivalent to the one from Definition 7, or method (non-primitive/abstract task).

Definition 15 (Method) *A method is a 3-tuple* ***m*** *=* $\langle$***name(m), pre(m), tasks(m), constr(m)***$\rangle$*, where:* ***name(m)*** *is the description or signature of* ***m****;* ***pre(m)*** *are the preconditions of* ***m****;* ***tasks(m)*** *are the subtasks of* ***m****, replacing the original task for new tasks;* ***constr(m)*** *are the ordering constraints imposed to the subtasks of* ***m****. Each ordering constraint describes the relation between two subtasks, e.g.* $t_i \prec t_j$*. In methods where not all subtasks are ordered, the planner is free to find an ordering that achieves the plan. The finite set of methods available is called* ***M****.*

During the HTN planning process, each possible DECOMPOSITION of **m** is found by searching which methods match the current task, **name(t)** = **name(m)** for t $\in$ SHIFT(**T**) $\wedge$ **m** $\in$ **M**. In this work we will limit to total-order decomposition, **constr(m)** = $t_{n-1} \prec t_n$ for n = |**tasks(m)**| and $n \geq 2$, which simplifies SHIFT(**T**) to consider only the first task. Partial ordering requires bookkeeping of the ordering constraints to interleave tasks, and more complex precondition descriptions, as tasks can be accomplished in many ways not described by total ordering. The preconditions of the first task to be decomposed, **pre(**SHIFT(**T**)), have their variable terms unified to objects based on the current state, and external functions terms replaced by objects returned from the function evaluation.

Definition 16 (External functions) *External functions are defined by a signature **name** applied to a sequence of N **terms**, represented by t_n, **e** = ⟨**name(e), terms(t_1, ..., t_n)**⟩, and replaced by a single object obtained from the function evaluation, before or during planning if terms are ground or lifted, respectively. The finite set of external functions is called **E**.*

External functions are not usually included in HTN formalizations, but are commonly found in HTN planner implementations. We thus formally include **E**, that allow the HTN planner to invoke external code to create new object terms, as the ones from Definition 1, for the problem instance during search (e.g. to represent numbers and numeric operations). In effect, the presence of such functions makes the states in HTN planning potentially infinite, since such functions can introduce arbitrary new objects.

Definition 17 (HTN domain) *The classical planning domain, with **F** facts and **A** actions, is extended with **M** methods and **E** external functions to make the HTN domain. The HTN domain is represented by **D** = ⟨**F, A, M, E**⟩.*

Definition 18 (HTN plan) *HTN plan represented by a sequence of actions that when applied in a specific order will modify **I** to an implicit **G** defined by **T**, π = ⟨$\boldsymbol{a}_1$, ..., $\boldsymbol{a}_n$⟩.*

Definition 19 (HTN planning instance) *HTN planning instance represented by the 3-tuple **P** = ⟨**D, I, T**⟩ and returns π or failure.*

2.2.2 Decomposition-based Planning algorithm

We illustrate the algorithm for Total-order Forward Decomposition (TFD) [36, chapter 11] in Algorithm 1 as one possible implementation of an HTN. TFD decomposes tasks using a recursive approach that selects one task at a time with constraint orderings satisfied, with SHIFT (Line 3), and check if the current task maps to a primitive or non-primitive task, an operator or method respectively. Tasks that map to an operator (Lines 4-7) modify the current state, while tasks that map to a method (Lines 8-12) are replaced by its subtasks. The process is repeated until it is not possible to decompose the current task, either by reaching only primitive tasks or failure, which forces the algorithm to backtrack, trying another applicable unification or decomposition on a previous level.

If no tasks are left to be decomposed the algorithm returns an empty plan as the base of recursion (Line 2). This style of planning is able to describe more than STRIPS, with a built-in operator heuristic function tailored to the domain and designer preferences [53], with all the methods required beforehand, which consumes a domain expert time. With domain knowledge the HTN domain description is more complex than the classical planning description, as recursive tasks can be described. Recursive tasks appear when a non-primitive task expands a branch that contain itself in the subtasks, this is useful when there is a need to apply several times the same set of operators until a stop condition is met, like walking until the destination is reached.

Algorithm 1 Total-order Forward Decomposition planner

```
1: function TFD(S, T, D)
2:     if T = $\varnothing$ then return empty $\pi$
3:     t ← SHIFT(T)
4:     if t is a primitive task
5:         for $t_{applicable} \in$ APPLICABLE(pre(t), S) do
6:             $\pi \leftarrow$ TFD(APPLY($t_{applicable}$, S), T, D)
7:             if $\pi \neq failure$ then return $t_{applicable} \cdot \pi$
8:     else if t is a non-primitive task
9:         for m $\in$ DECOMPOSITION(t, D) do
10:            for tasks(m) $\in$ APPLICABLE(pre(m), S) do
11:                $\pi \leftarrow$ TFD(S, tasks(m) $\cdot$ T, D)
12:                if $\pi \neq failure$ then return $\pi$
13:    return failure
```

2.2.3 Domain and Problem descriptions

Since both PDDL and SHOP [63] input are based on LISP representations and developed around the same time they share style, but are incompatible. PDDL is more verbose with each field being named, while SHOP descriptions require the user to know what each unnamed field must describe. The problem file is almost identical to PDDL, the main differences are the lack of an explicit set of **O** objects from Definition 1, and a Task list **T** from Definition 14, instead of a Goal state, as in Definition 10. Operators represent the same as the classical operators, which correspond to Definition 7, elements that when unified, using the Unification from Definition 3, can be used to APPLY effects, using the function from Definition 6, to the current State when APPLICABLE, according to Definition 5. SHOP operators have a name, a set of parameters, preconditions, negative and positive effects, and optional cost with default value of one. Methods, have a name, parameters, preconditions and subtasks, as seen in Definition 15. Methods can be decomposed in different ways and have an optional label for each case. Subtask constraints are simplified to either totally ordered or unordered. Unordered tasks may interleave subtasks from different tasks to minimize plan size, but require more preconditions, compared to totally-ordered tasks, to consider states generated by other tasks due to interleaving. The domain designer is responsible for possible repeated states and infinite loops, as recursive tasks may revisit states indefinitely. One must describe the domain in a way that such cases never happen, or add extra information to the state and domain description, e.g. by adding preconditions to not explore visited configurations, which are marked using invisible operators (used during planning, but removed from the final plan). The successors SHOP2 [64], JSHOP and JSHOP2 [46] extended the input description language to support task interleaving, backtracking tries, and protections (predicate guards to avoid task interference)

The PDDL domain from Listing 2.1 can be modeled as an HTN domain. The HTN Search domain show in Listing 2.3 illustrates how one uses domain knowledge to guide search in an HTN planner. The key method driving the HTN for a plan consists of the *forward* method as the main task. Figure 2.2 contains the decomposition of the *forward* task to a base or recursive *forward* method. The base precondition tests if the agent is already at the goal position, which requires no further subtask decomposition. Otherwise, the recursive method can be applied with the agent moving to an adjacent position, marking the previous position as visited and decomposing one more step of recursive before freeing visited positions with *unvisit*. The *visit* and *unvisit* invisible operators are used to avoid an infinite loop where an agent moves around the same positions, forcing the HTN to backtrack and try new adjacent positions. Visited positions are cleared once the goal position is reached by the base *forward* method, so an agent is able to reuse these positions by later decompositions. Both methods achieve the implicit destination. Note that adding operators to a PDDL domain would increase the number of actions to consider during classical planning, while an HTN planner ignores primitive and non-primitive tasks outside the decomposition process. This strict behavior also makes HTNs domains harder to design and test, as one may have to add or modify many methods to include support to a new operator, or risk having a planner that cannot handle certain situations.

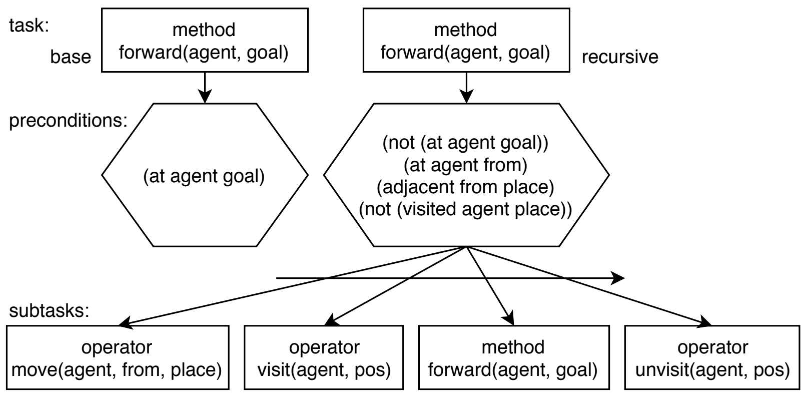

Figure 2.2 – In the HTN Search domain a recursive task is used to reach positions in a grid-based scenario. Two decompositions are available to *forward*, one without subtasks when the goal position is reached, and the other with a single movement before recursion.

The problem shown in Listing 2.4 has an equivalent initial state and an implicit goal state described by the main task *(forward ag1 p4)*. Many pathfinding solutions could be implemented to replace the *forward* search, other implementations include backward and bidirectional search [1], which expand positions from the goal position to the current position before moving, or from both directions until the paths cross, respectively.

All HTN descriptions in this thesis use JSHOP or JSHOP-like descriptions as no HTN standard description language currently exists, and JSHOP does not allow empty operator preconditions

[1]Available at *github.com/Maumagnaguagno/HyperTensioN/blob/master/examples/search/search.jshop*

to be omitted as SHOP. An example of a *call* to compute *distance* using an external function is shown in Listing 2.5.

```
(defdomain search (
  (:operator (!move ?agent ?from ?to)
    ( (at ?agent ?from) (adjacent ?from ?to) ) ; Preconditions
    ( (at ?agent ?from) )                      ; Del effects
    ( (at ?agent ?to) ) )                      ; Add effects
  (:operator (!!visit ?agent ?pos)
    ()                                         ; Preconditions
    ()                                         ; Del effects
    ( (visited ?agent ?pos) ) )                ; Add effects
  (:operator (!!unvisit ?agent ?pos)
    ()                                         ; Preconditions
    ( (visited ?agent ?pos) )                  ; Del effects
    () )                                       ; Add effects
  (:method (forward ?agent ?goal)          ; Decomposable by base or recursive
    base                                   ; Unique label used for debugging
    ( (at ?agent ?goal) )                  ; Preconditions
    ()                                     ; Empty subtasks
    recursive                              ; Unique label used for debugging
    (                                      ; Preconditions
      (not (at ?agent ?goal))
      (at ?agent ?from)
      (adjacent ?from ?place)
      (not (visited ?agent ?place)) )
    (                                      ; Subtasks
      (!move ?agent ?from ?place)
      (!!visit ?agent ?from)
      (forward ?agent ?goal)               ; Recursive subtask
      (!!unvisit ?agent ?from) ) ) )
```

Listing 2.3 – JSHOP Search domain with three operators and a recursive method.

```
(defproblem pb1 search
  ( ; Initial state
    (at ag1 p0)
    (adjacent p0 p1) (adjacent p1 p0)
    (adjacent p1 p2) (adjacent p2 p1)
    (adjacent p2 p3) (adjacent p3 p2)
    (adjacent p3 p4) (adjacent p4 p3) )
  ( (forward ag1 p4) ) ) ; Task list
```

Listing 2.4 – JSHOP Search problem with initial state and a single task.

```
(call distance ?x ?y ?gx ?gy)
```

Listing 2.5 – JSHOP Call statement for distance with two points described by four terms.

2.3 Temporal Planning

Temporal planning is used to describe even more complex problems, where actions have duration and a start time point. Preconditions are turned into conditions, which limit how the state must be before, during or after an action takes place. The same happen with effects, that now can happen at the start or end of an action. Now the planner needs to consider not only which action sequence will solve the problem, but also when each action must start and finish for their preconditions and effects to match. Temporal planning also adds events and processes as temporal constructions to describe a domain. Events are instantaneous changes applied to the environment once its preconditions are satisfied, such as a candle removing its light from the environment once its fuel ends, without the need of an action. Processes are continuous changes applied for as long as its conditions are satisfied, such as a candle consuming its fuel, gravity pulling objects, and the speed of an object changing its position. Events and processes are analogous to instantaneous and durative actions, respectively, taken by the environment itself and executed once their conditions are met.

To deal with time constructions the planner must handle a more expressive description and use a scheduler as flexible as the chosen description to control the beginning and end of each action, in this case durative actions. The STRIPS extension added by Temporal Graphplan (TGP) [74] requires all preconditions of an action to hold true during its execution, while PDDL 2.1 [28] expands the concept with *at start*, *at end* and *over all* keywords to describe when each predicate is expected/set to be true, as seen in Figure 2.3, allowing for more flexibility. Note that these keywords can be combined, using ***(and** (at start (p)) (at end (p)) (over all (p)))* as a precondition will force the predicate *(p)* to hold true for the entire duration of an action. Durative actions can also take advantage of time as a variable to describe continuous numeric effects, as the consumption of fuel decreasing for the amount of time an engine is running, as in the Durative-move domain of Listing 2.6. Not all planners are temporally expressive, able to consider the concurrency described in the domain correctly and optimally solve problems. Planners such as TGP are temporally simple and can still solve temporal domains that have no concurrency [14]. Other limitations can also impact what can be expressed in temporal domains. PDDL still lacks support for trigonometric functions, which would make simple movements trivial to describe using sine and cosine, as such functions have non-linear behaviors and are complex to be evaluated during planning. Domain descriptions are limited to functions provided by the language specification and planner implementation.

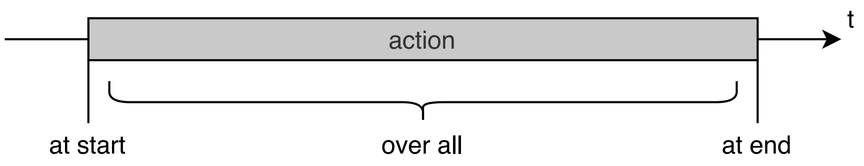

Figure 2.3 – PDDL 2.1 time constructions applied to actions is limited to *at start*, *at end*, *over all*.

```
(define (domain durative-move)
  (:requirements :fluents :durative-actions)
  (:predicates (moving ?vehicle) )
  (:functions (position ?vehicle) (fuelLevel ?vehicle) )
  (:durative-action move
    :parameters (?vehicle)
    :duration (= ?duration 100)
    :condition (over all (>= (fuelLevel ?vehicle) 0))
    :effect (and
      (at start (moving ?vehicle))
      (increase (position ?vehicle) (* #t 5))
      ; (increase (positionX ?vehicle) (* (* #t 5) (cos (angle))))
      ; (increase (positionY ?vehicle) (* (* #t 5) (sin (angle))))
      (decrease (fuelLevel ?vehicle) (* #t 10))
      (at end (not (moving ?vehicle))) ) )
  ... ) ; Other actions omitted
```

Listing 2.6 – Durative action example using PDDL 2.1 temporal elements to describe the fuel consumption and position changes of a vehicle as it moves using *#t* to represent the action duration.

HTNs can also incorporate temporal elements in their domain descriptions. PDDL 2.1 durative actions can be converted to SHOP2 HTN methods by following nine steps [38]. Such steps are:

1. Create a new method definition with the same signature as the <*act*> durative action.
2. Add all *at start* conditions as preconditions to the new method for early failure detection.
3. Handle action duration: Add three additional variables, *?start*, *?end*, and *?duration* to the method definition, and additional preconditions binding them:
   (a) *(time ?start)*
   (b) *(assign ?duration <original-PDDL-duration>)*
   (c) *(assign ?end (+ ?duration ?start))*
4. Create a *!start-<act>* operator, adding the *?start* variable to its arguments.
5. Create an *!end-<act>* operator, adding the *?end* variable to its arguments
6. The task network for the new method will be *(:ordered !start-<act> !end-<act>)*.
7. Add all *at end* conditions as preconditions to the *!end-<act>* operator.
8. *over all* conditions are enforced as follows:
   (a) Add the *over all* conditions as preconditions to the method <*act*>, so that they will hold at the start of the interval.
   (b) For each *over all* condition, the *!start-<act>* operator must add a protection, marking predicates as immutable.

   (c) Protections added by the *!start-<act>* operator must be removed by the *!end-<act>* operator.

9. Manage the time fluent:

   (a) Add *(time ?start)* precondition to the *!start-<act>* operator.

   (b) Update the time fluent at the *!end-<act>* operator. To do this, first check that the time for the *!start-<act>* operator has not passed adding the following preconditions: *(time ?t)* and *(<= ?t ?end)*. Finally, add the effect *(time ?end)*.

However, it is not possible to model concurrency with continuous effects based on a time variable that is always incremented without extra structures being added to the underlying HTN planner to contain the status of each predicate at specific time points. PDDL+ [27] events and processes could be used to take advantage of the more focused search of HTN to minimize the computational cost of the many interferences between effects, requiring external structures to handle time intervals and constraints, however no HTN planner ported such concepts from PDDL+. The PDDL+ domain from Listing 2.7 contains position, speed and acceleration functions to hold numeric data about the state and processes to update these values based on a time variable *#t* when the car is running. The goal in this domain is to safely stop near a certain position. The acceleration is propagated through processes to a velocity function *v*, and *v* is propagated to a displacement function *d*. The planner is responsible for choosing when and for how long each action is applied to reach the goal *d*.

```
(define (domain car_linear_mt_sc)
  (:predicates (engine_running) (engine_stopped) )
  (:functions (d) (v) (a) (max_acceleration) (min_acceleration) (max_speed) )
  (:constraint speed_limit
    :condition (and (>= (v) (* -1.0 (max_speed))) (<= (v) (max_speed))) )
  (:process displacement :precondition (engine_running)
                         :effect (increase (d) (* #t (v))) )
  (:process moving       :precondition (engine_running)
                         :effect (increase (v) (* #t (a))) )
  (:action accelerate
    :precondition (and (< (a) (max_acceleration)) (engine_running) )
    :effect (increase (a) 1.0) )
  (:action stop_car
    :precondition (and (> (v) -0.1) (< (v) 0.1) (= (a) 0.0) (engine_running) )
    :effect (and (assign (v) 0.0) (engine_stopped) (not (engine_running)) ) )
  (:action start_car
    :precondition (engine_stopped)
    :effect (and (engine_running) (not (engine_stopped)) ) )
  (:action decelerate
    :precondition (and (> (a) (min_acceleration)) (engine_running) )
    :effect (decrease (a) 1.0) ) )
```

Listing 2.7 – Car Linear domain with PDDL+ processes to apply acceleration to velocity and velocity to displacement changes.

2.4 Motion Planning

Motion Planning is used in robotics to obtain certain poses (physical configurations) from a sequence of discrete movements considering physical and temporal constraints, to evaluate obstacles in movement [51]. Continuous values are used to solve the several geometric constraints during planning, unlike classical planning where symbols play the main role. The constraints are required by the robot to avoid overlaps between its parts and other objects, which would translate into collisions, how to pick and place objects correctly, and apply safe speeds to the actuators. Since motion planners have applications in robotics, it is necessary to deal with unforeseen events, due to limited range sensors and other agents also modifying the world, which force the robot to explore different paths after learning new space constraints.

In order to approach this problem we can start with a relaxed version of the *basic motion planning problem* [51]. In this relaxed version we assume the environment to be static with a single rigid body moving object, our robot, which means no other objects are moving and the collection of particles of each object have fixed position relative to one another. The velocity and position constraints related to the object are ignored. The n-dimensional space where the robot moves is called the workspace. In this same workspace there are rigid body obstacles, therefore creating a subset of workspace called obstacle region. Assuming that we know the current positions of the object and obstacles the problem consists of moving the object to its goal position, without collisions. This problem can also be referred to as the *path planning problem*, or as a specific case for two dimensions called the *piano movers problem*, as the movers cannot lift the object and are constrained to move in a plane [72]. Approaches to motion planning include: roadmaps, cell decomposition, sampling-based methods and potential fields. In the next subsections these approaches are described.

2.4.1 Roadmaps

Roadmaps [73] use a network of curves on free space to represent possible paths for the robot to take. Once generated only such standardized paths are considered. Path planning is simplified to finding a connection between the start and goal positions to the roadmap and a path within the roadmap that connects these two points. Different roadmaps can be generated with different methods such as the *visibility graph* that use obstacle vertices to create possible paths, seen in Figure 2.4, or a retraction that use a function to slice the scenario into regions, such as the ones from a *Voronoi diagram* [5], and use the limits between them as the roadmap as shown in Figure 2.5.

The visibility graph is a non-directed graph whose nodes are the start and goal positions as well as the obstacle vertices. The links of such graph are the straight line segments that do not intersect with any obstacle. Obstacles must be simplified to polygons with a limited number of vertices to simplify computation to generate and query the graph during planning.

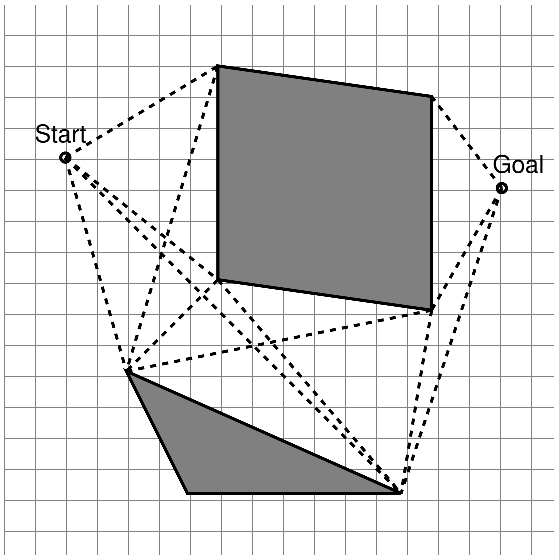

Figure 2.4 – Visibility graph with two polygon obstacles between start and goal points.

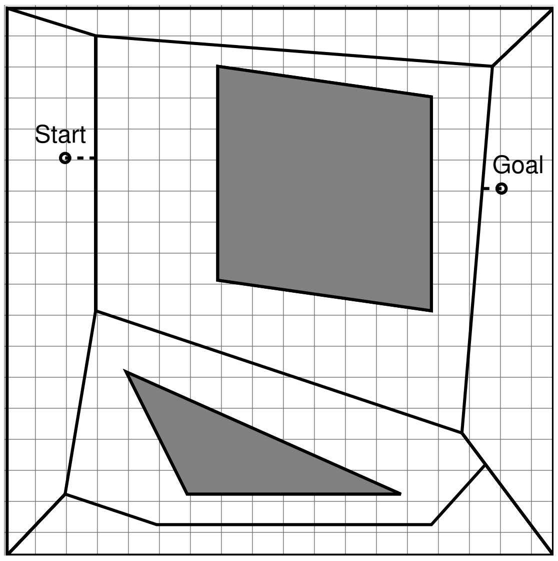

Figure 2.5 – A Voronoi-based roadmap connecting start and goal nodes to the paths obtained.

2.4.2 Cell Decomposition

Cell decomposition methods slice the environment into cells to calculate the path between the cell that contain the start position to the goal position [72]. Such methods can exactly match the collision free region with each cell shape being adapted by details of the environment and robot, or approximate using predefined shapes. The adaptive cell decomposition approach recursively partitions the environment in quadrants, refining the areas near obstacles, reaching an exact solution with enough subdivisions without generating many discrete areas for empty regions. The cell data is stored in a quadtree [26], a tree structure with up to four leaf children nodes (NE, NW, SW and SE) to every node. The approximate cell approach, on the other hand, is a simpler solution that uses the same cell dimensions to discretize the entire environment, simplifying partition at the cost of a more complex search (as more cells are generated in this method), specially in empty regions. Both resulting cell decompositions are shown in Figures 2.6 and 2.7.

2.4.3 Sampling-based methods

An explicit representation of the environment is required to apply roadmaps or cell decomposition approaches. As the configuration space becomes more complex these approaches become too costly to be used. To avoid an explicit representation one can sample the environment with different strategies to generate a connectivity representation of the free space. This approach can deal with any number of environment dimensions and obstacles of any shape. Two subdivisions exist: Probabilistic Roadmaps (PRMs) and single-query planners.

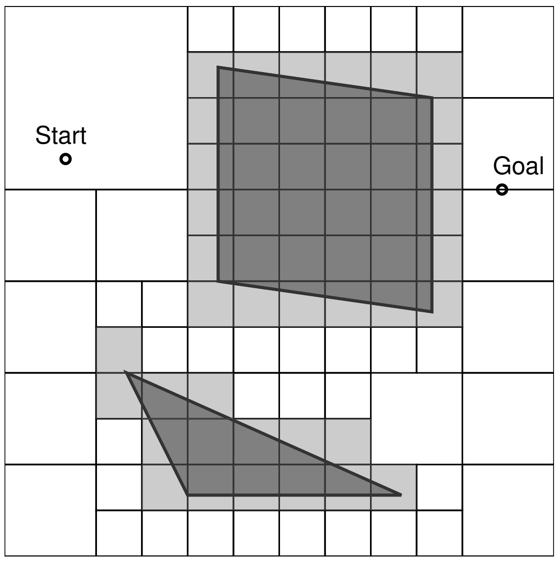

Figure 2.6 – Adaptive cell decomposition with collision free regions in white and blocked regions in gray. Cell size is adapted to minimize cell quantity and better describe environment details.

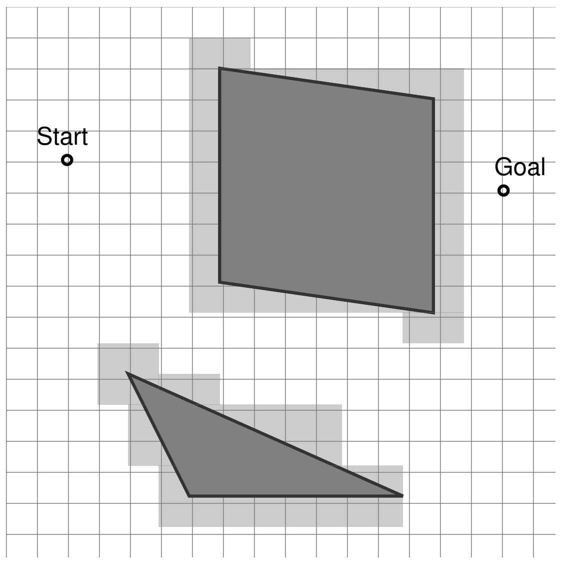

Figure 2.7 – Approximate cell decomposition with collision free regions in white and blocked regions in gray. All cells share the same shape.

Probabilistic Roadmaps (PRMs) [48] generate a roadmap by sampling the environment with a probability distribution instead of an obstacle-based approach. PRM generation is made of learning and query phases. In the learning phase, random samples based on a distribution are obtained and checked for collision with the obstacles. If a sampled path does not collide and can be connected to a at least one nearby previous node according to some metric, such as Euclidean distance, usually by a straight line, the node and vertices are added to the roadmap. The process repeats until enough nodes or vertices are added to the roadmap. The query phase attempts to find a path from the initial to the goal state via the roadmap. The process can either obtain a path from a search algorithm using the roadmap, fail and retry after improving the connectivity from a second round of learning or using a specialized strategy, or simply return failure. PRMs will always find a solution given enough nodes, but may fail to find passages in narrow paths using few nodes. Figure 2.8 shows an example of path generated through PRMs such that the learning phase generated too few nodes. To generate enough nodes one can either distribute nodes closer to obstacles (Obstacle-based PRM [1]) or based on a Voronoi diagram of the environment [42].

Unlike other techniques that aim to obtain a reusable structure for multiple motion planning problems in the same environment, single-query planners aim to solve only one problem. Their main advantage is to represent only connectivity related to the problem. Unlike roadmaps, where a node is added when it does not collide with an obstacle, nodes in single-query planners are only added if they can be connected to the current structure, a tree that starts from the start node, which removes the need for an additional search phase. The most used single-query planner approach is the Rapidly-exploring Random Tree (RRT) [52], which samples in the goal node direction using a fixed step size closer to goal nodes. After a certain number of nodes have been added to the structure, the process can try to connect nodes to the goal node instead of new random sampled ones. To speed up the process, a bidirectional approach can be used, growing random trees from the start and goal nodes and connecting each other after an expansion phase.

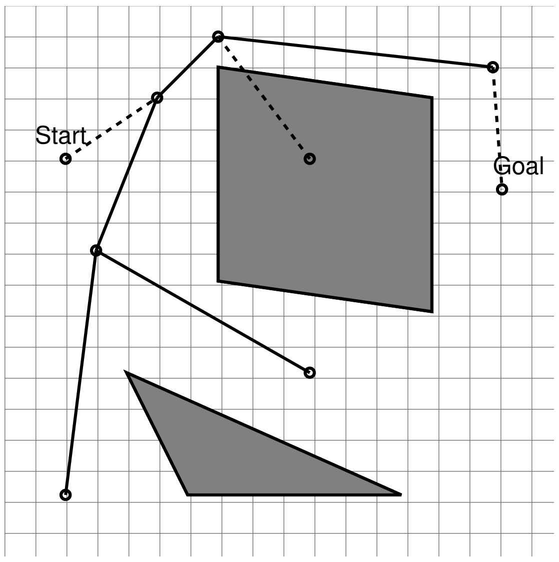

Figure 2.8 – A Probabilistic Roadmap with few nodes, but able to connect start and goal nodes to the roadmap. One sample could not be added to the roadmap as it collides with an obstacle.

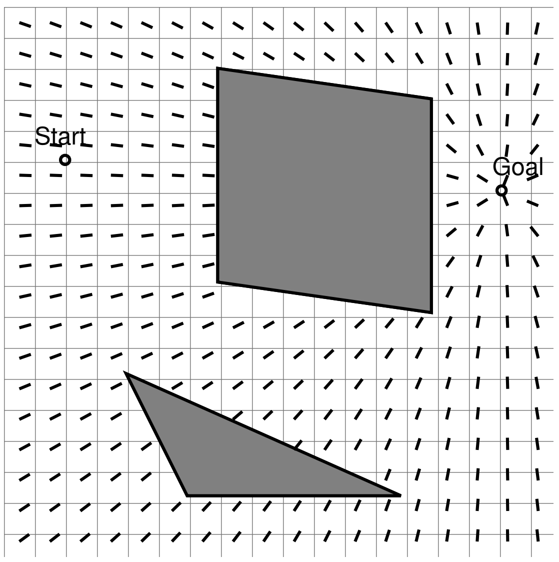

Figure 2.9 – A simple potential field function generating a trap as the direction from start to goal is perpendicular with an obstacle wall.

2.4.4 Potential Fields

Potential fields [3] discretize the environment to a fine regular grid of configurations to guide the search. The robot is simplified to a particle that is attracted and repulsed by artificial potential fields, the goal and the obstacles respectively. The forces applied to each position in the grid move the robot in a promising direction towards the goal. This method can be very efficient when compared with other methods, but can also get the robot trapped in a dead-end without careful design of the potential field function to avoid such scenarios. A simple potential field of our example environment is shown in Figure 2.9.

2.5 Coroutines

Coroutine is a cooperative multitasking approach that gives the programmer full control about their execution, as routines are not preempted by a scheduler, but explicitly called from other routines, with persistent state between calls [62]. Generators are limited coroutines that always pass control back to the caller while yielding a value, and thus are usually used to implement iterators.

Every time a *yield* instruction is executed the generator state is saved to the stack, suspended, and resumed once the generator is called again.

Figure 2.10 shows how the main program, the caller, and a coroutine, the callee, interact. The main program starts calling the coroutine, which initializes its internal state and passes control to it. The active coroutine computes the value to be yielded to the caller, but saves its state before yielding and being suspended. The caller, now with the first value yielded, can operate on this value and destroy the coroutine, or call the coroutine again to obtain a new value from the coroutine. This process is repeated until the caller destroys the coroutine, using explicit functions, letting the coroutine out of scope, or the coroutine failing to return another possible value, which is expected behavior for coroutines that read files in blocks or compute elements of finite sets. Since coroutines do not need to store previously yielded values in memory, the caller is expected to either consume or store the values obtained from the coroutine, which removes memory leaking problems for long term programs. The flexibility of coroutines allows it to be the caller to other coroutines.

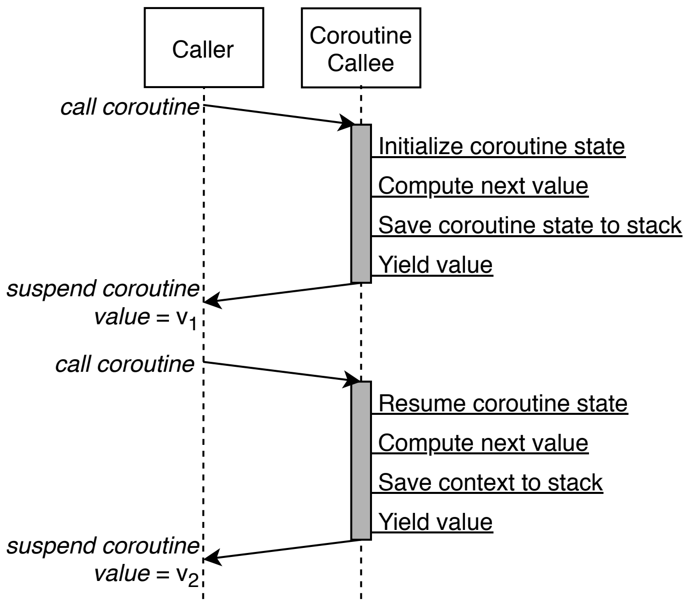

Figure 2.10 – Basic main program (caller) and coroutine (callee) interaction to generate two possible values in two calls.

3. SYMBOLIC-GEOMETRIC PLANNING

In this chapter we describe our approach to symbolic-geometric planning. Symbolic planners are fast at solving domains that can be discretized into symbols, as seen in Section 2.1.3, when used with performant heuristic functions, while retaining the domain characteristics we are planning for. However, if the discretization process is not possible, either by requiring tedious human work or by not obtaining enough detail with an automated process, one must use a planner that also supports numeric features. Numeric features usually require a less readable description, and more computing power and memory due to the increased number of states numeric variables can reach.

Also, numeric features come with rounding errors due to the internal representation of floating point numbers. Unlike symbolic approaches where symbols are compared for equality during precondition evaluation, one must compare numeric values with a delta to consider such errors. If the planner is the one responsible for the comparison it will do so for every numeric element implicitly, which impacts planning time, otherwise the user is responsible for explicitly defining when one must consider rounding errors, which impacts description time and maintenance. For complex object instances (from object-oriented programming) such as polygons that are made of many point instances, the number of comparisons in the description is prone to error. The comparison itself, usually, is not the goal of the planning description, in this case the same polygon can be defined by many orderings of the same points (the triangle *ABC* is the same as *BCA*) and within floating point error, point B is almost equal to B', as in the example from Figure 3.1. Such comparisons are part of subproblems that are better solved by external and specialized libraries, such as the Computational Geometry Algorithms Library (CGAL) [23] or Boost.Geometry [35].

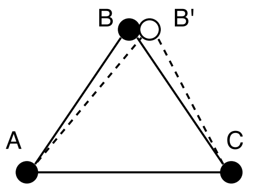

Figure 3.1 – The points ABC and BCA represent the same polygon, and considering floating point errors one could even consider the B point the same as B'. This makes the comparison of continuous values more complex than purely symbolic comparison common on planning descriptions.

Some HTN planners, such as JSHOP [46], are able to take advantage of specialized libraries using external function calls, as the ones from Definition 16, to do such comparisons. Function calls can also be used to access external solvers to handle complex numeric and object operations. Such external functions can solve complex motion planning problems found during symbolic planning. However, this approach is limited, function calls are expected to be deterministic and return a single value, in this case a motion plan. If a domain designer wants to try multiple values until one succeeds, the return value must be a list of values. Generating a complete list of possible values is more costly than computing a single value, as the first one could be enough to find a feasible plan. Instead of the complete list approach, it is more consistent with planning descriptions to use the equivalent of an external predicate, computed on demand during planning and not stored in the state structure as the Predicates of Definition 2. Such external predicates are called semantic attachments.

3.1 Semantic attachments

The semantic attachment term was coined by Weyhrauch [79] to describe the attachment of an interpretation to a predicate symbol using an external procedure. Semantic attachments have already been used for classical planning [20] to unify values just like predicates, and their main advantage is that new users do not need to discern between them and common predicates. While common predicates are stored in a state structure, semantic attachments are computed at runtime and can be implemented as generators/coroutines, as seen in Section 2.5. Unification and APPLICABLE, as seen in Definition 3 and Definition 5, can be merged and implemented as a generator that yields ground instances of an operator or method with preconditions satisfied by the current state. Such generator can be used to obtain other unifications once the HTN planner backtracks, until no more unifications can be obtained, forcing the planner to return to the previous decomposition level.

In Figure 3.2 the process of unification of a free variable from a method precondition using a coroutine is illustrated. As the value V_1 for *?var* does not satisfy other preconditions or forces the HTN to backtrack, the coroutine is resumed to obtain V_2, a new unification for ?var, and HTN decomposition is resumed. The inequality *?var* $\neq V_1$ is kept true by the coroutine state, that should not try to repeat the same assignment for *?var*, in this case based on pointers and indexes of which values have already been yielded. Two distinct solutions can be applied when multiple variables are unified by the same coroutine, the first solution is to generate the cartesian product and assign variables based on each line of the product. The second solution is to choose values that better match other parameters, that are grounded, and the current state. The second solution is more complex, requiring domain expert knowledge to take full advantage of which values and ordering to try.

Definition 20 (Semantic Attachments) *Semantic Attachments are defined by a signature* ***name*** *applied to a sequence of N* ***terms****, represented by* t_n*,* ***sa*** = $\langle$***name(sa), terms(***t_1***, ...,*** t_n***)***$\rangle$ *that maps to an equivalent coroutine implemented externally. We call* ***SAs*** *the finite set of semantic attachments available during planning.*

The **SAs** are coroutines implemented externally and exposed to the HTN domain description, extending the HTN domain previously defined in Definition 17. With a state that is not only declarative, with parts being procedurally computed, it is possible to minimize memory usage and delegate complex state-based operations to external methods otherwise incompatible with planning.

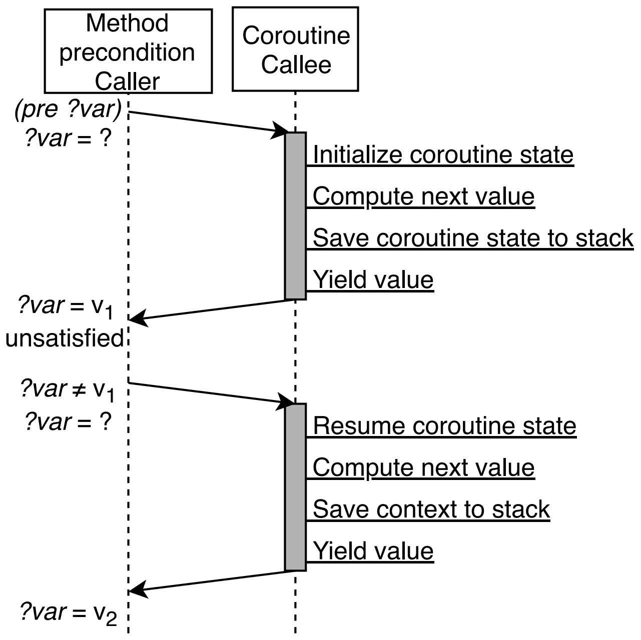

Figure 3.2 – Method precondition example computing and assigning possible values V_1 and V_2 to *?var* variable through a coroutine, one at a time.

3.2 Symbolic anchors for external values

Since we are exploiting external libraries through function calls and semantic attachments, we can hide or abstract away the numeric parts of planning in a layer between the planner and the libraries. Our three-layered architecture of Figure 3.3 was inspired by the work of Lavindra and Meneguzzi [16]. In the symbolic layer we manipulate an anchor symbol, corresponding to a Term of Definition 1, such as *polygon1*, while in the external layer we manipulate *Polygon instance with N points* as an object or struct based on what the selected external library specifies. The intermediate layer acts as the foreign function interface between the other two layers, and can be replaced or modified to accommodate other external libraries without modifications to the symbolic description. The intermediate layer is domain dependent, however its functions and semantic attachments can more easily be reused in other domains, due to their limited scope, than actions and methods, corresponding to the Definitions 8 and 15. This way we avoid complex representations in the symbolic layer and incomplete representations in the external layer.

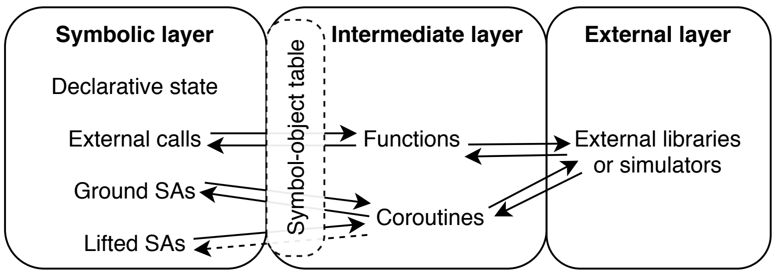

Figure 3.3 – Symbolic and external layers share information through an intermediate layer that maps representations and calls between them.

Object instances created in the external layer are compared with previously exposed object instances. Object instances that do not match previously exposed objects are mapped to new anchor symbols and stored in the symbol-object table. Object instances that match a previously exposed object map to the same previously attributed symbol. Object matching can be implemented through equality or equivalence, when certain features are not equal but within an error margin, according to the domain. This process makes symbol comparison work in the symbolic layer even for symbols related to complex objects/structs, solving the ABC equals B'CA triangle problem, for example. Symbols generated in branches that later backtrack due to failure are kept in the symbol-object table, which facilitates debugging, as the user can see readable identifiers repeating instead of numeric values.

Definition 21 (Symbol-object table) *Our symbol-object table is defined by N key-value pairs, represented by* ***SO-table*** *=* $\langle\langle$***key(k***$_1$***), value(v***$_1$***)***$\rangle$***, ...,*** $\langle$***key(k***$_n$***), value(v***$_n$***)***$\rangle\rangle$*.*

The symbol-object table is accessed through two functions: QUERY and INSERT. The QUERY function is used to find pairs based on anchor symbols as keys, used to convert symbols into usable object instances by external calls, as functions and semantic attachments. The INSERT function is used to find or create symbolic anchors to external object instances. Once operations are finished in the external layer the INSERT function expose resulting objects as symbols through free variables of semantic attachments, or as the return value of external function calls.

Definition 22 (Query) *The query function is represented by* QUERY*(****SO-table****, symbol-*k_n*) =* $\langle$*symbol-*k_n*, object-*$v_n$$\rangle$ *if symbol-*k_n $\in$ ***SO-table***$_n$*, else* $\varnothing$*, where n represents indexes of the* ***SO-table****.*

Definition 23 (Insert) *The insert function is represented by* INSERT*(****SO-table****, object-*v_n*) = symbol-*k_n *if object-*v_n $\in$ ***SO-table***$_n$*, else new-symbol-*k_{n+1}*. The insert function also have the side effect of adding a tuple to the* ***SO-table*** *for any new object,* ***SO-table***$_{n+1}$ *=* $\langle$*new-symbol-*k_{n+1}*, object-*$v_{n+1}$$\rangle$*.*

3.3 Domain and Problem description

We decided to follow the style of the JSHOP [46] description, keeping most language elements intact, while adding elements to represent our approach. The main structure of domain and problem, as well as methods, remain the same. Each semantic attachment predicate signature is described as the Predicate of Definition 2, as seen in Listing 3.1, and must be present in the domain file for the planner compiler to know which predicates are externally evaluated. Signatures are not uncommon in planning, as PDDL also requires predicate signatures to be explicitly defined. Each semantic attachment used in the domain must have only one signature and its name must match the actual implementation. The semantic attachment parameters are included to increase maintainability of the description, as only the arity of the semantic attachment is tested before planning starts. The user must supply at most the same quantity of parameters as described in the signature, while parameters that are not supplied will be initialized with default values described in the semantic attachment implementation. Not finding the implementation, not matching the semantic attachment arity, or missing parameters without default values results in error. The problem description contains the same JSHOP elements. We created a new file extension to discern between the original and the extended language files, UJSHOP, with semantic attachments. The implementation of external functions and semantic attachments is part of the intermediate layer.

```
(defdomain example (
  ; Signatures of semantic attachments in use must be explicitly defined
  (:attachments (semantic-attachment-name ?param1 ?param2) )
  (:operator ...)
  (:method ...) )
```

Listing 3.1 – UJSHOP example domain with ADJACENT SA signature with two terms.

3.4 Intermediate layer

The symbol-object table, external functions and semantic attachments are in the intermediate layer, which is defined in a separate file from domain and problem. The planner loads such file if one is available at the same folder as the domain and problem files, otherwise continues normally. Listing 3.2 contains a complete example of an external description. The symbol-object table generation is currently not automated and the user is responsible for choosing a structure to generate symbols that better represents the objects being handled by the external layer. If this is not the case a generic Hash table (Lines 5-6), with a function to convert object to symbol (Lines 8-14), can be copied to this file. The *symbol* function in this example is equivalent to the *insert* function from Definition 23, as this function name may vary. The symbol turned into **SO-table** key is currently generated as a counter prefixed by a constant string, but the user could generate specific symbols for complex objects using the equivalent of *to string* methods defined by the object class. In this example the *query* function from Definition 22 is the actual Hash table access method and requires no extra function definition. If the user prefers, the symbol-object table may start with predefined symbols, which may aid debugging later, in this case two points are defined as *start* and *goal*.

Functions and semantic attachments start by converting symbols to objects using the table, when not doing purely symbolic operations, as seen in the external function *distance* (Lines 16-20). Note that *distance* expects two position symbols that already exist in the table, if such symbols are not found we could raise an exception or return an infinity symbol as the resulting distance. If a new symbol is returned by a call or unified by a free variable in a semantic attachment, it is the user responsibility to store this unique symbol in the table to reuse later, but there is no mechanism to enforce it, so the infinity symbol from the previous example could not be stored in the table. The semantic attachment ADJACENT (Lines 22-37), may operate as a ground predicate precondition, which can be true or false (Line 35), or lifted predicate precondition, unifying the free variable *?pos2* with any position in the Moore neighborhood, eight cells around a central cell in a grid.

```
 1 module External
 2   extend self
 3
 4   WIDTH = HEIGHT = 5
 5   @symbol_object = {'start' => Point.new(0,0), 'goal'  => Point.new(4,4)}
 6   @symbol_counter = 0
 7
 8   def symbol(object)
 9     symbol = @symbol_object.key(object)
10     return symbol if symbol
11     symbol = "symbol#{@symbol_counter␣+=␣1}"
12     @symbol_object[symbol] = object
13     return symbol
14   end
15
16   def distance(pos1, pos2)
17     pos1 = @symbol_object[pos1]
18     pos2 = @symbol_object[pos2]
19     return Math.hypot(pos1.x - pos2.x, pos1.y - pos2.y).to_s
20   end
21
22   def adjacent(pos1, pos2)
23     pos1 = @symbol_object[pos1]
24     if pos2.empty?
25       [[-1,-1], [0,-1], [1,-1], [-1,0], [1,0], [-1,1], [0,1], [1,1]].each {|x,y|
26         nx = pos1.x + x
27         ny = pos1.y + y
28         if 0 <= nx and nx < WIDTH and 0 <= ny and ny < HEIGHT
29           pos2.replace(symbol(Point.new(nx, ny)))
30           yield
31         end
32       }
33     else
34       pos2 = @symbol_object[pos2]
35       return (pos1.x - pos2.x).abs <= 1 and (pos1.y - pos2.y).abs <= 1
36     end
37   end
38 end
```

Listing 3.2 – Intermediate layer with symbol-object table, *distance* function and ADJACENT SA.

Instead of adding adjacent positions to the initial state we can compute them during planning, which makes the symbolic description of the initial state simpler. If the domain ever changes to a different map discretization, such as hexagon grids or irregular polygons, we only need to change the behavior of ADJACENT. Domain constants may be exposed to the intermediate layer from the beginning, such as width and height. Backtracking does not affect any external structure state, including the symbol-object table, therefore increasesing in size as more objects are exposed to the symbolic layer if the user does not explicitly remove them. Instead of creating an analogous state in the external layer we believe it is currently simpler to cache costly computations for later use instead of adding a mechanism to keep states consistent across layers. To better deal with cases where information found during search may be useful even after backtracking, such as newfound constraints or visited positions, we can use the external layer to store it.

3.5 Optimize the description by reordering preconditions

If we restrict our operators to STRIPS [25], i.e. no disjunctions in preconditions and effects, we can reorder the preconditions during the compilation process to improve execution time, removing the burden of the domain designer to optimize a mostly declarative description by hand, based on how free variables are used as terms. Each free variable creates a dependency between the first predicate or semantic attachment that contains such variable as a term and the next predicates or semantic attachments with the same term. The first predicate or semantic attachment is responsible for grounding such variable while the next predicates or semantic attachments only verify if the previously ground value matches the current state. Predicates have priority over semantic attachments to ground free variables, as the possible values are obtained from the current finite state, while semantic attachments may cover a possibly infinite number of values. Algorithm 2 shows how preconditions are filtered in distinct sets to later be reordered for improved performance.

Consider the abstract method example of Listing 3.3, with two semantic attachments among preconditions, *sa1* and *sa2*. This method is compiled to Algorithm 3 with both semantic attachments evaluated after common predicates, while function calls happen before or after each semantic attachment, based on which variables are ground at that point. In Line 4 the free variables *fv1* and *fv3* have a ground value that can only be read and not modified by other predicates or semantic attachments. In Line 7 every variable is ground and the second function call, difference, can be evaluated.

Each semantic attachment is responsible for unifying all remaining free variables with valid values before resuming and, preferably, define a stop condition, otherwise the HTN process will keep backtracking and evaluating the semantic attachment seeking new values and never returning failure. Due to the implementation support of arbitrary-precision arithmetic and accessing data from real-world streams of data/events (which are always new and potentially infinite) a valid value may never be found. Thus, we expect the domain designer to implement mechanisms that limit the maximum number of times a semantic attachment evaluates a call (i.e. to have finite stop conditions). This maximum number of tries can be implemented as a counter in the internal state of a semantic attachment to avoid repeated evaluation of the same values. Note that the number of side-effects in both

Algorithm 2 Filter preconditions during compilation process based on free variables used as terms of predicates and semantic attachments.

```
1: function FILTERPRECONDITIONS(pre)
2:     $P_{ground} \leftarrow \{p \mid p \in pre \wedge (p \in \mathbf{F} \vee (p \in \mathbf{E} \wedge \forall t \in \mathbf{terms}(p)\ t = object))\}$
3:     $P_{lifted} \leftarrow \{p \mid p \in pre \wedge (p \in Predicate \vee p \in \mathbf{E}) \wedge \exists t \in \mathbf{terms}(p)\ t \neq object\}$
4:     $P_{sa} \leftarrow \{p \mid p \in pre \wedge p \in \mathbf{SA}\}$
5:     return $\langle P_{ground}, P_{lifted}, P_{sa} \rangle$
```

```
(:attachments (sa1 ?a ?b) (sa2 ?a ?b) )
(:method (m ?t1 ?t2)
  label
  (                               ; Preconditions
    (call != ?t1 ?t2)             ;   no dependencies
    (call != ?fv1 ?fv2)           ;   ?fv1 and ?fv2 dependencies
    (sa1 ?t1 ?fv1)                ;   no dependencies, ground ?fv1
    (pre1 ?t1 ?t2)                ;   no dependencies
    (sa2 ?fv1 ?fv2)               ;   ?fv1 dependency, ground ?fv2
    (pre2 ?fv3 ?fv1)) )           ;   ?fv1 dependency, ground ?fv3
  ((subtask ?t1 ?t2 ?fv1 ?fv2)) ) ; Subtask
```

Listing 3.3 – Abstract method with semantic attachments among preconditions.

Algorithm 3 Compiled preconditions from Listing 3.3, reordered to optimize execution time.

```
1: function M(S, $t1, t2$)
2:   if t1 $\neq$ t2
3:     for each fv1, fv3; $\{\langle$pre1, t1, t2$\rangle$, $\langle$pre2, fv3, fv1$\rangle\} \subseteq$ S do
4:       for each sa1(t1, fv1) do
5:         free variable fv2
6:         for each sa2(fv1, fv2) do
7:           if fv1 $\neq$ fv2 then yield [$\langle$subtask, t1, t2, fv1, fv2$\rangle$]
```

external functions calls and semantic attachments increases the complexity of correctness proofs and the ability to inspect and debug domain descriptions.

Unlike Algorithm 1, we move the generic APPLICABLE of Definition 5, from the HTN algorithm to custom unifiers implemented by the compilation process directly into the operator and method functions, with preconditions reordered based on the return of Algorithm 2. This is an important modification that allows more complex preconditions to be evaluated. By moving such routine to the method itself we can have custom implementations, including generator-based ones. The original Algorithm 1 does not define how the APPLICABLE set of free variable assignment can be implemented. The equivalent non-custom version of our approach is defined by Algorithm 4. The symbol-object table is an argument of the new APPLICABLE routine to be used by external function calls and semantic attachments after ground preconditions are satisfied and lifted preconditions used to evaluate some of the free variables present in the preconditions.

Algorithm 4 Semantic attachment-enabled APPLICABLE tests ground, lifted and coroutine based preconditions to replace free variables with suitable values.

```
1: function APPLICABLE(pre, S, SO-table)
2:   $P_{ground}$, $P_{lifted}$, $P_{sa}$ $\leftarrow$ FILTERPRECONDITIONS(pre)
3:   if $P_{ground} \not\subseteq$ S then return
4:   for $pre_{lifted}$ with free variables $fvs_{lifted} \in P_{lifted}$ do
5:     for each $fvs_{lifted}$ match with name($pre_{lifted}$) $\in$ S do
6:       for $pre_{sa}$ with free variables $fvs_{sas} \in P_{sa}$ do
7:         for EXTERNAL($pre_{sa}$, $fvs_{sas}$, S, SO-table) do
8:           yield
```

3.6 Framework

In order to bring together the symbolic domain and problem descriptions, external functions and semantic attachments, and the reordering optimizations seen in the previous section with the HTN planner, we need a framework. The framework is responsible for the compilation process. The compilation process between description and planning is taken by *Hype*, a framework that parses the planning description to an intermediate representation and compile [1] to an equivalent Ruby code that connects with *HyperTensioN U* [2], our HTN planner whose features are described in this chapter. The parser or compiler can be replaced for equivalent modules to support other input or output formats, respectively, as long as they are compatible with the intermediate representation. Optimizations can be applied to the intermediate representation if necessary, such as the preconditions reordering from Section 3.5. External functions, semantic attachments and the symbol-object table are defined in an external file in the compiler target language, in this case Ruby, outside the compilation process and only loaded during planning, making $\mathbf{D} = \langle \mathbf{F}, \mathbf{A}, \mathbf{M}, \mathbf{E}, \mathbf{SAs}, \mathbf{SO\text{-}table} \rangle$. The process and its elements are better seen in Figure 3.4. The process is based on another project of ours [55], *HyperTensioN* [3], forked to accommodate new features using a new intermediate representation.

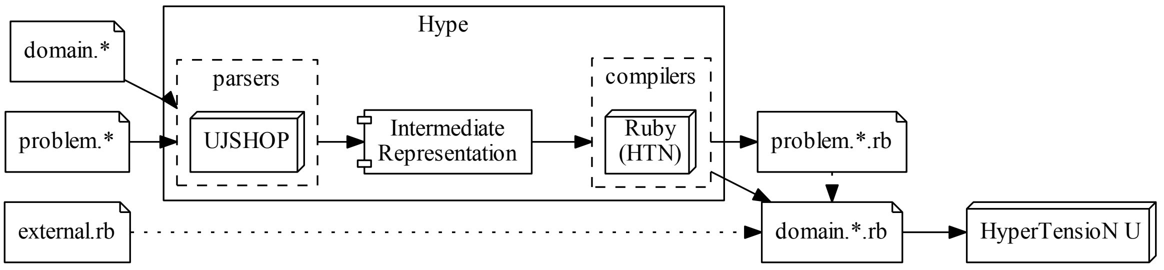

Figure 3.4 – *HyperTensioN U* framework parses symbolic description and compiles to the target language before connecting with external definitions (intermediate layer) and HTN planner.

Once the planning description is compiled and linked with the planner, it is possible to decompose the tasks. To decompose such tasks into a plan we adapt Algorithm 1 to support coroutine-based semantic attachments. Semantic attachments increment the concept of the APPLICABLE function as seen in Definition 5, as they expand preconditions to either test or unify free variables relying on coroutines defined externally, and not only the current State. To improve execution time the preconditions may be reordered by the compilation process: ground function calls are tested earlier than other preconditions, then free variables from predicates are unified and, one by one, their unifications are used by lifted predicates and function calls. Semantic attachments, which may be several, are

[1]The compiler here is also considered a source-to-source compiler or *transpiler*, as code is only translated from one language to another, not to machine code.

[2]Available at *github.com/Maumagnaguagno/HyperTensioN_U*

[3]Available at *github.com/Maumagnaguagno/HyperTensioN*

then applied to either generate more unifications for variables that are still free or just test if they are satisfied by the current state. Finally, the remaining predicates and function calls that require at least one free variable to be grounded by semantic attachment can be tested as ground preconditions. For each unification the method subtasks are decomposed or operator effects applied. If done in a different order we are relying too much in a partially declarative language, in which the user have no hint that order may impact unification time, generating multiple values where ground preconditions could safely return earlier, or generating infinite values from a semantic attachment that unifies before the common unification engine, which uses values from the finite current state.

3.7 Semantic Attachments and Symbol-Object table use-cases

In the following subsections three different use-cases are presented. The first case describes the usage of the symbol-object table to hide numeric details from the description. The second use-case describes the usage of semantic attachments to iterate over a numeric interval, an operation required by geometric and temporal problems that must consider dimensions. The final use-case contains a semantic attachment being used to either unify free variables or compute the truth value of a ground external predicate.

3.7.1 Discrete distance between objects

A common problem when moving in dynamic and continuous environments is to check for object collisions, as agents and objects do not move across tiles in a grid. One solution is to calculate the distance between the centroid of both objects and verify if this value is in a safe margin before considering which action to take. To avoid the many geometric elements involved in this process we can map centroid position symbols to coordinate instances and only check the symbol returned from the symbol-object table, ignoring specific numeric details and comparing a symbol to verify if objects are near enough to collide. This process is illustrated in Figure 3.5, in which p_0 and p_1 are centroid position symbols that match symbols S_0 and S_1 in the symbol-object table, which maps their value to point objects O_0 and O_1. Such internal objects are used to compute *distance* and return a symbolic distance in situations where the actual numeric value is unnecessary. The implementation resembles Algorithm 5, returning near or far symbols based on the distance, but never adding such symbols to the symbol-object table.

Algorithm 5 Distance function can use the symbol-object table to access position symbols and hide away numeric computation and return value.

```
1: function DISTANCE($p0, p1$)
2:     $o0 \leftarrow$ QUERY(SO-table, $p0$)
3:     $o1 \leftarrow$ QUERY(SO-table, $p1$)
4:     return SYMBOL(HYPOT(X($o0$) - X($o1$), Y($o0$) - Y($o1$)))    ▷ Map values to near or far symbols.
```

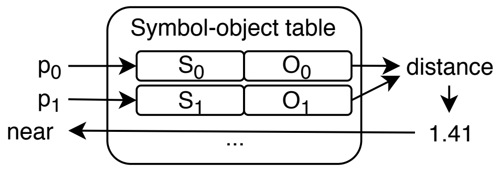

Figure 3.5 – The Symbol-object table maps symbols to object-oriented programming instances to hide procedural logic from the symbolic layer.

3.7.2 An iterator for HTN

In order to find a correct number to match a spatial or temporal constraint one may want to describe the relevant interval and precision to limit the number of possibilities without having to discretely add each value to the state. Planning descriptions usually do not contain information about numeric intervals and precision, and if there is a way to add such information it is through the planner itself, as global definitions applied to all numeric functions, i.e. as the timestep, mantissa and exponent digits of DiNo [67]. The STEP semantic attachment described in Algorithm 6 addresses this problem, unifying t with one number at a time inside the given interval with an ϵ step. Listing 3.4 contains the STEP semantic attachment being used to generate possible deadlines two timesteps from the current timestep *now*. The INSERT function from Definition 23 is responsible for generating and adding a readable symbol to *t*, which is the *?deadline* variable in our domain.

Algorithm 6 The STEP semantic attachment replaces the pointer of *t* with a numeric symbol before resuming control to the HTN.

```
1: function STEP($t, min = 0, max = \infty, \epsilon = 1$)
2:     for i ← $min$ to $max$, step $\epsilon$ do
3:         $t$ ← INSERT(SO-table, i)
4:         yield                                        ▷ Resume HTN
```

```
(defdomain timestep-example (
  (:attachments (step ?t ?min ?max ?step))
  ; ...
  (:method (m ?now)
    ( (step ?deadline (call + ?now 2)) ) ; Precondition
    (                                    ; Subtasks
      (!start         ?deadline   )
      (!move  (call + ?deadline 1))
      (!stop  (call + ?deadline 2)) ) ) )
```

Listing 3.4 – STEP semantic attachment works as an iterator in the *m* method preconditions to generate possible deadlines for subtasks. It uses only the free variable *?t* and *?min* value terms, *?max* and *?step* implicitly fallback to default values externally defined.

3.7.3 Lazy adjacency evaluation

To avoid having complex effects in the move operators one must not update adjacencies between planning objects during the planning process. Instead one must update only the object position, deleting the old position and adding the new position to the state. Such positions come from a partitioned space, previously defined by the user. The positions and their adjacencies are either used to generate ground operators or stored as part of the state. To avoid both solutions, one could implement adjacency as a coroutine while hiding numeric properties of objects, such as position. Algorithm 7 is equivalent to the ADJACENT semantic attachment present in Listing 3.2, with the main two cases for planning descriptions. In the first case both symbols are ground, and the coroutine resumes when both objects are adjacent, doing nothing otherwise, failing the precondition. In the second case $s2$, the second symbol, is free to be unified using $s1$, the first symbol, and a set of directions D to yield new positions to replace $s2$ with a valid position, one at a time. Thus, this coroutine either checks whether $s2$ is adjacent to $s1$ or tries to find positions adjacent to $s1$ binding the corresponding values to $s2$ if such values exists.

A very common use-case of adjacency in HTN planning is illustrated by Listing 3.5. A *forward* method is used to traverse a grid-like map where one agent tries to move around until a goal position is reached, backtracking when no new position can be accessed. This replacement is trivial, as one formula can be used to express adjacent positions, but leaves one open question: what if we add the goal as a parameter of the ADJACENT SA? In Chapter 4 we investigate how much information a domain designer is capable of exposing to a semantic attachment and how much it impacts the domain description complexity and planning time.

Algorithm 7 The ADJACENT semantic attachment can either check if two symbols map to adjacent positions or generate new positions and their symbols to unify $s2$.

```
1: D ← {⟨-1,-1⟩,⟨0,-1⟩,⟨1,-1⟩,⟨-1,0⟩,⟨1,0⟩,⟨-1,1⟩,⟨0,1⟩,⟨1,1⟩}
2: function ADJACENT($s1, s2$)
3:     $s1$ ← QUERY(SO-table, $s1$)
4:     if $s2$ is ground
5:         $s2$ ← QUERY(SO-table, $s2$)
6:         if $|x(s1) - x(s2)| \leq 1 \wedge |y(s1) - y(s2)| \leq 1$ then yield
7:     else if $s2$ is free
8:         for each ⟨x, y⟩ ∈ D do
9:             nx ← x + x($s1$);    ny ← y + y($s1$)
10:            if $0 \leq nx < WIDTH \wedge 0 \leq ny < HEIGHT$
11:                $s2$ ← INSERT(SO-table, ⟨nx, ny⟩)
12:                yield
```

```
(defdomain adjacency-example (
  (:attachments (adjacent ?position1 ?position2))
  ; ...
  (:method (forward ?agent ?goal)
    goal
    ( (at ?agent ?goal) )            ; Precondition
    ()                               ; Empty subtasks
    keep-moving
    (                                ; Preconditions
      (not (at ?agent ?goal))
      (at ?agent ?here)
      (adjacent ?here ?there)        ; or (adjacent ?here ?there ?goal)
      (not (visited ?agent ?there))
      (not (blocked ?there)) )
    (                                ; Subtasks
      (!move ?agent ?here there)
      (!!visit ?agent ?here)
      (keep-moving)
      (!!unvisit ?agent ?here) ) ) )
```

Listing 3.5 – The ADJACENT semantic attachment replaces the many adjacent predicates from the declarative state, making problem descriptions smaller and more readable, while a smaller state structure saves memory.

4. EXAMPLES AND EXPERIMENTS

In the following sections we explore how different problems can be described to be solved by our approach. We conducted empirical tests of our HTN planner in a machine with Dual 6-core Xeon CPUs @2GHz / 48GB memory, repeating experiments three times to obtain an average. The results show a substantial speedup over the original classical descriptions from DiNo [67] and ENHSP [70] with more complex descriptions, while being competitive against Metric-FF [41].

4.1 Adjacency / Initial state and method ordering impact

In this section we improve the efficiency of HTN planning in traditional scenarios, just by modifying the order in which planning elements are defined and considered during planning. The order in which elements are defined in a planning description may affect which tasks are decomposed first and the resulting plan. Using a grid-based scenario as an example, a domain designer may opt for a directional or non-directional domain description. Both approaches present the same problem: the order of elements in the symbolic description influences planning time and solution.

A directional domain description is made of specialized methods for each direction, which may force exploration of many unnecessary states or go directly to the goal position according to different maps. Consider the example scenario of Figure 4.1 with an agent initially in the center and a goal position in the upper-right corner. The methods static ordering dictates which directions are explored first. For example, the direction ordering up, left, right, down would force exploration of almost the entire scenario, save bottom-right tiles, while the ordering down, up, left, right would make the agent explore the entire scenario This static method ordering is useful in domains where certain directions are more promising or yield better plans, like side-scrolling games.

Other domain designers may opt for a non-directional domain description that just moves to adjacent positions not visited, resulting in a more compact domain description, as previously seen in Listing 2.3. The impact of method ordering is replaced by the adjacent description contained in the initial state, as tiles are compared following the internal state structure ordering, generated from the initial state description.

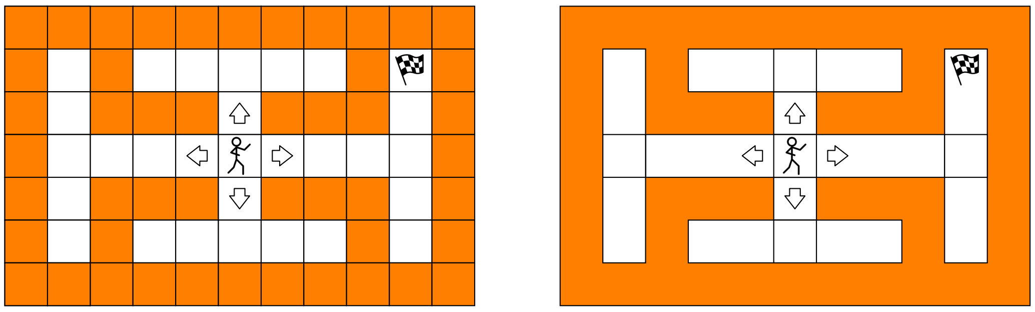

Figure 4.1 – Two discretizations of the same maze-like scenario. The discretization and order in which methods select a direction directly impact the amount of tiles explored during HTN planning.

```
(:method (forward-direction ?agent ?goal)
  base
  ((at ?agent ?goal)) ; Precondition
  () ; Empty subtasks

  recursion-right
  ((at ?agent ?from) (right ?from ?place) (not (visited ?agent ?place)))
  ( ; Subtasks
    (!move ?agent ?from ?place)
    (!!visit ?agent ?from)
    (forward ?agent ?goal)
    (!!unvisit ?agent ?from) )
  recursion-up
  ... )
```

Listing 4.1 – Prioritize directions in a certain order using custom method decompositions instead of the generic adjacent movement.

The solution to this problem is to add goal information to avoid expanding tiles in any direction. One could optimize the direction selection using external functions to compute which direction is more promising. In order to move towards the goal the adjacent positions must be ordered from nearest to furthest to the goal before decomposition continues. To calculate the distance between two points one must use numeric values instead of discrete positions, e.g. *(at agent 1 1)* instead of *(at agent top-left-corner)*. Adding such details to the domain description is nontrivial, as not all HTN planners support priority/sorting constructions and the lack of a description standard force domain designers to know internal details about their HTN planner and description language to implement such constructions. JSHOP2 [46] allows prioritization using the *sort-by* construction, as seen in Listing 4.2, which binds values to a variable based on a specified comparison function. However it is not clear if JSHOP2 currently allows external functions to be used together with the prioritization construction, as no examples or documentation cover this aspect.

A domain designer may also opt for a different domain description while trying to improve planning time, using a new discretization of the same scenario by changing the granularity of the description. By clustering corridor tiles into single tiles one can effectively traverse many intermediate states in a single action, however it requires a preprocessing stage and make numeric approaches with direction prioritization harder to work with. Compare the previous ordered and sorted solutions with the symbol-object table and semantic attachment solution presented in Listing 4.3, in which both direction prioritization and readable names are used through the semantic attachment and symbol-object table, respectively, without the inclusion of another specific construction. The performance difference between the non-directional domain with and without semantic attachments is shown in Figure 4.2 using 20 automatically generated 15x15 maze problems, displayed in Figure 4.3, with unique tile names instead of numeric coordinate values. Our agent starts at the top-left position and must reach the bottom-right position in all problems. In scenarios where the agent enters a wrong path with many dead-ends we can see a peak in the planning time, as in the problem 17 without SA and the problem 8 with SA. Problems with many dead-ends repeatedly reach the same position, as the depth-first search style of HTN combined with a local *visited* predicate is not enough to mark such positions between decompositions, requiring a backtracking-persistent cache to avoid such repetitions.

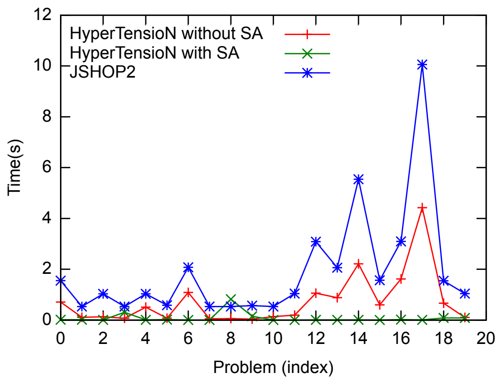

Figure 4.2 – Time in seconds to find a single plan for Maze problems.

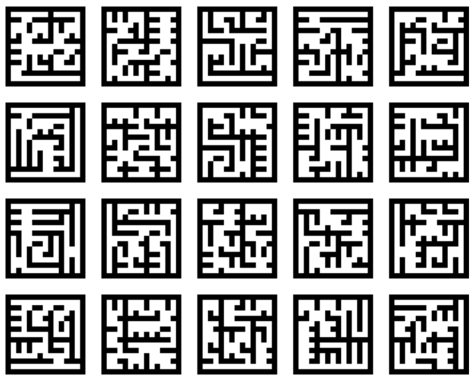

Figure 4.3 – Maze problems generated through recursive division, adding walls with passages.

```
(:method (forward-direction ?agent ?goal)
  base
  ((at ?agent ?goal)) ; Precondition
  () ; Empty subtasks
  recursion
  ( ; Preconditions
    (:sort-by ?distance < (and
      (at ?agent ?from)
      (adjacent ?from ?place)
      (not (visited ?agent ?place))
      ; Not clear if assigned variables with sort-by are supported
      ; (assign ?distance (call distance ?placex ?placey ?goalx ?goaly))
      ; Requires precomputed distances in the initial state
      (distance ?place ?goal ?distance) )) )
  ( ; Subtasks
    (!move ?agent ?from ?place)
    (!!visit ?agent ?from)
    (forward ?agent ?goal)
    (!!unvisit ?agent ?from) ) )
```

Listing 4.2 – Sort-by mechanism available in JSHOP binds variable to values according to the comparison selected.

```
(:attachments (adjacent ?pos ?near ?goal))
(:method (forward ?agent ?goal)
  base
  ...
  recursion
  ( ; Preconditions
    (at ?agent ?from)
    ; Equivalent to (adjacent ?from ?place) with ?goal priority
    (adjacent ?from ?place ?goal)
    (not (visited ?agent ?place)) )
  ( ... ) ) ; Same subtasks of previous forward method
```

Listing 4.3 – Improve domain readability with symbols instead of coordinates and domain performance with a more expressive semantic attachment that considers the goal position.

4.2 Plant Watering / Gardening

In the Plant Watering domain [29] one or more agents move in a 2D grid-based scenario to reach taps to obtain water and pour water in plants spread across the grid. Each agent can carry up to a certain amount of water and each plant requires a certain amount of water to be poured, as seen in Figure 4.4. Many state variables can be represented as numeric fluents, such as the coordinates of each agent, tap and plant, the amount of water to be poured and being carried by each agent, and the limits of how much water can be carried and the size of the grid. There are two common problems in this scenario, the first is pathfinding, to reach either a tap or a plant, the second is the top level strategy. To avoid considering multiple paths in the decomposition process one can try to move straight to the goal position in scenarios without obstacles, which is the case of this domain and simplifies the forward method. Otherwise positions closer to the goal are considered first. To achieve this straightforward movement we modify the ADJACENT semantic attachment to also consider the goal position, using Algorithm 8. A top level strategy may consider which plant is closer to a tap or closer to an agent, how much water an agent can carry and so on. A simpler top level strategy verifies how much water must be poured to a plant, move to any tap, obtain water, move to the previously selected plant and pour all the water loaded. This process is repeated until every plant has enough water poured. The forward method is shown in Listing 4.4 and compiled to Algorithm 9. We compare with the fastest satisficing configuration of ENHSP (`sat`) and Metric-FF 2.1 (`standard-FF`) in Figure 4.5, which shows that our approach is faster with execution times near 0.01s (ignoring interpreter loading time), or competitive with Metric-FF around 0.11s (considering interpreter loading time), with all three planners obtaining non-step-optimal plans.

Algorithm 8 In this goal-driven ADJACENT semantic attachment the positions are coordinate pairs, and two variables must be unified to a closer to the goal position in an obstacle-free scenario.

```
1: function ADJACENT($x, y, nx, ny, gx, gy$)  ▷ Or ADJACENT($xy, nxy, gxy$) for symbolic positions
2:     ▷ $\langle x, y\rangle \leftarrow$ QUERY(SO-table, $xy$)
3:     ▷ $\langle gx, gy\rangle \leftarrow$ QUERY(SO-table, $gxy$)
4:     ▷ COMPARE returns -1, 0, 1 for $<, =, >$, respectively
5:     ▷ $nxy \leftarrow$ INSERT(SO-table, $\langle x$ + COMPARE($gx$, $x$), $y$ + COMPARE($gy$, $y$)$\rangle$)
6:     $nx \leftarrow x$ + COMPARE($gx$, $x$)
7:     $ny \leftarrow y$ + COMPARE($gy$, $y$)
8:     yield
```

Algorithm 9 Compiled output of the Plant Watering HTN domain excerpt from Listing 4.4.

```
1: function FORWARD($a, gx, gy$)
2:     if x($a$) = $gx \land$ y($a$) = $gy$ then yield $\varnothing$
3:     else
4:         free variables nx, ny
5:         for each ADJACENT(x($a$), y($a$), nx, ny, $gx$, $gy$) do
6:             yield [$\langle$move, $a$, nx, ny$\rangle$, $\langle$forward, $a$, $gx$, $gy\rangle$]
```

```
(:attachments (adjacent ?x ?y ?nx ?ny ?gx ?gy))
(:method (forward ?a ?gx ?gy)
  base
  ( ; Preconditions
    (call = (call function (x ?a)) ?gx)
    (call = (call function (y ?a)) ?gy) )
  () ; Empty subtasks
  keep_moving
  ( ; Preconditions
    (adjacent (call function (x ?a)) (call function (y ?a)) ?nx ?ny ?gx ?gy) )
  ( ; Subtasks
    (!move ?a ?nx ?ny)
    (forward ?a ?gx ?gy) ) )
```

Listing 4.4 – Excerpt of the Plant Watering HTN domain, the ADJACENT semantic attachment is described separately.

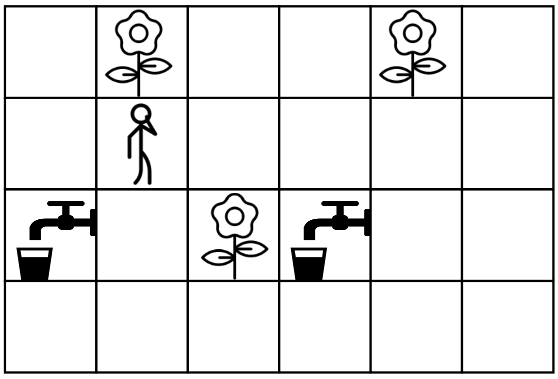

Figure 4.4 – Plant watering problem with one agent, four flowers and two taps in 6x4 grid-based scenario.

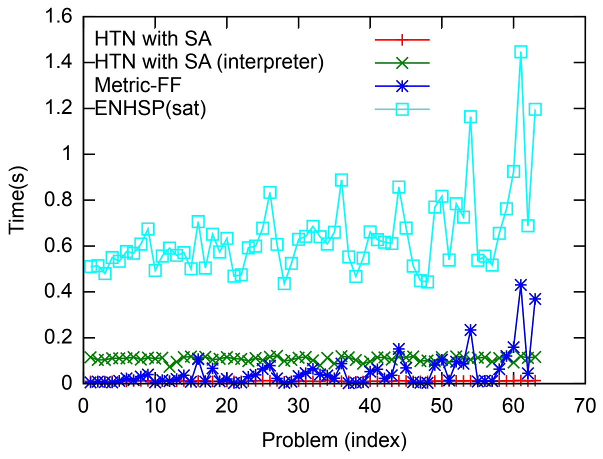

Figure 4.5 – Time in seconds to solve Plant Watering problems.

4.3 Sokoban

Preprocessing is an important step in many domains. In this section we explore preprocessing as a regular HTN task of the domain using the tight integration between symbolic and intermediate layer. The intermediate layer is used to modify the state with preprocessed data, generate possible values and store data about previous failed HTN branches to improve planning time.

In the Sokoban game an agent must push one or more boxes in a warehouse to their storage locations. Each location is a cell on a grid, either clear or occupied by an agent, box, or wall, as seen in Figure 4.6. Boxes can get stuck into non-storage locations or behind other boxes, which creates many possible deadlocks (locations that when occupied by a box create dead end states) within the game, forcing it to be reset. To avoid such situations the actions taken by the agent must be carefully planned ahead, which makes Sokoban a challenging puzzle from a search perspective. Sokoban is very trivial in terms of description, with only move and push operators. The agent can always move to an adjacent clear location or push an adjacent box towards a direction in which there is a clear location, moving the agent to the box location and the box to the previously clear location.

The domain designer has two key challenges to solve in their description: how to describe adjacent and pushable in terms of predicates. Adjacency can be described in the initial state for a NxM grid as *(adjacent cell1 cell2)*, which generates many predicates and will require more predicates as N or M increases. Pushable can be described in the same way, *(pushable cell1 cell2 cell3)*. These two predicates greatly increase the number of predicates in the initial state, while also requiring maintenance to support bigger maps. Such process can be automated by a preprocessor, as map dimensions are limited, to generate the complete initial state. HTNs can move such preprocessing stage to happen during planning, which can save time for domains where a Sokoban-like subproblem may never have to be solved for a specific task to be completed, which never requires adjacency and pushable to be computed. Going one step further, we can preprocess deadlocks and avoid many actions that lead to no solution. Listing 4.5 contains our Sokoban HTN domain with the original operators, one invisible operator to verify repeated states and two methods, one to preprocess deadlocks and the other to move/push agent/box recursively until all boxes are in storage locations. This domain description takes advantage of accessing the symbolic state from external parts to simplify description complexity. The external elements ADJACENT, PUSHABLE and *boxes_stored* read *clear* cells and box locations from the current state to either perform a unification or test externally, while *find_deadlocks* function adds deadlock predicates to the state. The *new_state* function accesses a persistent memory outside the planning engine, which is a separate structure that holds state information even after backtracking occurrences. This memory is useful to avoid dead end states repeatedly in many branches, as current deadlock preprocessing does not consider deadlocks that involve other boxes, only the map layout. The *new_state* function either adds the current new state to the structure and returns true, or verifies that the current state is not new and returns false, which makes the precondition fail.

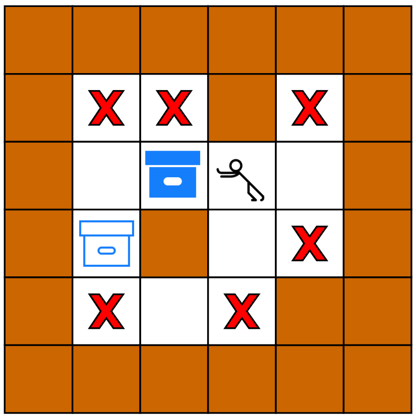

Figure 4.6 – Sokoban game with walls restricting the actions of an agent that must move boxes to their storage location. Boxes that reach deadlock locations, represented by X, cannot be moved again.

```
(defdomain sokoban (
  (:attachments (adjacent ?from ?to) (pushable ?from ?intermediate ?to) )

  (:operator (!move ?from ?to)
    ()                              ; Empty preconditions
    ( (player ?from) (clear ?to) )  ; Del effects
    ( (player ?to) (clear ?from) ) ) ; Add effects

  (:operator (!push ?from ?intermediate ?to)
    ()                                                    ; Empty preconditions
    ( (player ?from) (box ?intermediate) (clear ?to) )    ; Del effects
    ( (player ?intermediate) (box ?to) (clear ?from) ) ) ; Add effects

  (:operator (!!visit ?player) (call new_state ?player) () ())

  (:method (solve-processed ?player)
    no_box_outside_storage
    (call boxes_stored) ; Precondition
    ()                  ; Empty subtasks
    push
    ( (pushable ?player ?box ?clear) (not (deadlock ?clear)) ) ; Preconditions
    ( (!push ?player ?box ?clear) (!!visit ?box) (solve-processed ?box) )
    move
    (adjacent ?player ?clear)                                        ; Precondition
    ( (!!visit ?clear) (!move ?player ?clear) (solve2 ?clear) ) ) ; Subtasks

  (:method (solve ?player)
    preprocess
    (call find_deadlocks)                                 ; Precondition
    ( (!!visit ?player) (solve-processed ?player) ) ) )) ; Subtasks
```

Listing 4.5 – Sokoban domain with ADJACENT and PUSHABLE semantic attachments, and *new_state*, *boxes_stored* and *find_deadlocks* external functions used to generate or test external state features.

4.4 Real-Time Strategy In-range Problem

Some problems have clear solutions that are either hard to describe in finer detail, due to planning description limitations, and hard for the planner to derive from the high level operators and methods alone. In this section we explore how to describe one specific problem that requires numeric value support, storing and sorting values to reach an optimal solution.

Real-Time Strategy (RTS) games are a subgenre of strategy games in which teams fight for resources in a 2D scenario. Units of each team gather resources to obtain buildings or improvements in a real-time environment. The basic problem of RTS games is the number of possible actions that can be made at any moment due to the vast number of available units (agents) and specialized actions for each unit while reacting to actions from the opponent. Players spend hours training to find useful strategies to win in certain scenarios. Due to their complexity, RTS games are also subject of AI research [66]. Many AI approaches can be used to play RTS games, but a naive search-based classical planner is not the preferred option, as it will suffer from a predicate combinatorial explosion caused by the many agents and positions that can be unified in this domain, which combined with long plans will take enough time for the opponent to improve and attack without resistance. The strategic nature of the game matches HTN solutions better than classical planning [49], as strategies focus on certain action sequences and limit the number of states the planner can reach to only promising ones. The ranged attack is one specific problem of interest within RTS games. Unlike common attacks that require two units to be in adjacent positions to perform an attack action, a ranged unit can attack within a certain range. To exploit such feature an attacking unit prefers to avoid doing any extra steps closer to the target with the ranged unit and attacking as far as possible. To describe such strategy the map is simplified to a grid with three relevant positions: the position of the target enemy, the position of the attacker and the position in range that we want to discover. The goal is to minimize the number of movements required by the attacker to be in range. Consider Figure 4.7, where one green ranged unit have the task to attack a red enemy unit. In this implementation the range is a circle of radius three. Considering each free position to be a valid candidate, the first thing to do is to check if the distance between a position and the target is smaller than the range, in this example we assume three cells. To simplify this process, we can filter positions that are inside the smallest square that limits this circle, avoiding several distance calculations on large grids. This square is clipped by the grid limits, which must also be taken into account. Now we can test each free position within the square that is also inside the circle, store each one with their respective distance to the target in a list and sort it based on the distance between the attacker and the position, to minimize movement. The first elements of such list after this process (for a mostly free grid) are positions that are over the circle border, being not only closer to the attacker, but as far away as possible from the target unit.

This solution is simpler in text than in planning description, as filtering and sorting are procedural features that are either not supported by planners or hard to describe, especially when combined, as previously seen in Section 4.1 with sorting. An implementation of the *go-attack* method using our approach is shown in Listing 4.6, and the IN_RANGE semantic attachment is shown in Listing 4.7. The *go-attack* method have two cases, one where the attacker is already in range, and another where it moves with *forward* before being able to decompose to an *attack* method that repeatedly apply attack actions. The IN-RANGE semantic attachment follows the same process previously described. Objects are not stored in a table in this implementation as no position symbols are used, only numeric values as *X* and *Y* coordinates.

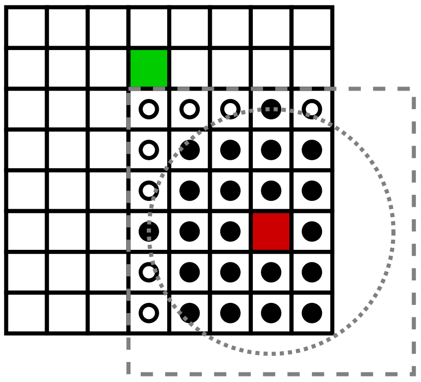

Figure 4.7 – Discrete RTS ranged attack, a green ranged unit must select a position that is within the range radius to attack the red enemy unit while trying to minimize movements to reach such position.

```
(:attachments (in_range ?x ?y ?ax ?ay ?tx ?ty ?range ?width ?height))
(:method (go-attack ?attacker ?target)
  in-range
  (
    (range ?attacker ?range)
    (at ?attacker ?ax ?ay)
    (at ?target ?tx ?ty)
    (call <= (call ^ (call + (call ^ (call - ?ax ?tx) 2)
                              (call ^ (call - ?ay ?ty) 2)) 0.5) ?range) )
  ( (attack ?unit ?target) )
  not-in-range
  (
    (grid ?width ?height)
    (range ?attacker ?range)
    (at ?attacker ?ax ?ay)
    (at ?target ?tx ?ty)
    (in_range ?x ?y ?ax ?ay ?tx ?ty ?range ?width ?height)
    (free ?x ?y) )
  ( (forward ?unit ?x ?y) (attack ?unit ?target) ) )
```

Listing 4.6 – Go-attack method with IN_RANGE semantic attachment.

```
def in_range(x, y, ax, ay, tx, ty, range, width, height)
  ax = ax.to_i
  ay = ay.to_i
  tx = tx.to_i
  ty = ty.to_i
  range = range.to_i
  width = witdh.to_i
  height = height.to_i
  minx = tx > range ? tx - range : 0
  miny = ty > range ? ty - range : 0
  maxx = tx + range
  maxy = ty + range
  maxx = witdh  - 1 if maxx > witdh - 1
  maxy = height - 1 if maxy > height - 1
  list = []
  minx.upto(maxx) {|tox|
    miny.upto(maxy) {|toy|
      list << [tox, toy] if Math.hypot(tx - tox, ty - toy) <= range
    }
  }
  list.sort_by! {|i,j| Math.hypot(ax - i, ay - j)}.each {|i,j|
    x.replace(i.to_f.to_s)
    y.replace(j.to_f.to_s)
    yield
  }
end
```

Listing 4.7 – IN-RANGE semantic attachment externally defined.

4.5 Motion Planning

Most planning problems are limited by the symbolic approach to deal only with discrete representations. Maps are usually based on a grid, where positions are cells that can either be occupied or available, similar to board games and puzzles, such as Sokoban. Not only a person or process must slice the scenario in cells before planning starts, this may limit planning to simplified solutions that require actual numeric definitions in realistic applications. To plan within a realistic scenario one must deal with the continuous and geometric properties of it, usually 2D or 3D. Since planners and geometric libraries are usually developed separately, one must either recreate geometric functions for the planner or adapt a library to avoid rework. This process is not trivial: not only rounding errors, but object comparisons, could lead to failures during planning or incorrect generation of a (non-satisficing) plan. With our approach, one can reuse the same symbolic description for most motion planning subproblems using a combination of preprocessing, as the one used in Sokoban from Section 4.3, and ADJACENT semantic attachment, from Section 3.7.3. The actual motion planning is implemented in the intermediate layer. In the following subsections we discuss how motion planning techniques could be integrated in our symbolic-geometric planning approach.

4.5.1 Roadmap and Cell decomposition

The visibility graph from Roadmaps, from Section 2.4.1, and cells from Cell decomposition, from Section 2.4.2, can be generated by a function call and stored in the intermediate layer, while adjacency between the regions can be implemented through the ADJACENT semantic attachment to generate new locations to move. By choosing to not preprocess the map into roadmap or cells, one will have to generate map information during each semantic attachment call, possibly caching information obtained during each semantic attachment call for large maps. By adding the goal position as a parameter to the ADJACENT semantic attachment it is possible to give priority to portions of the map between current and goal points. The resulting HTN domain is very similar to the Plant Watering domain of Section 4.2, however map properties are not discretized beforehand, letting the planner decide when, how and which parts of the map to discretize as needed. For cell decomposition approaches, the cell size can also be controlled during planning using a *?cell-size* parameter in the semantic attachment, Listing 4.8, to deal with objects with different proportions or safety margin distances.

In this configuration the continuous map must be described in the initial state or loaded from an external source. Continuous map descriptions require polygons and polyhedra details, which are harder to maintain using planning descriptions. The map description can also be loaded from an external source. An example of such problem description is shown in Listing 4.9, with problem specific details described in the problem file, while a common map is loaded from an external source.

```
(:attachments (adjacent ?from-cell ?to-cell ?goal-cell ?cell-size))
```

Listing 4.8 – ADJACENT semantic attachment for cell decomposition.

```
(defproblem pb1 search
  ( ; Initial state
    (at robot start)
    (position start 8 5) ; Define position coordinates
    (position goal 6 8)
    (polygon  1 1  2 1  2 2  1 2) ; Add square obstacle
    ; ... Add many obstacles here ...
    (load-map map.json) ) ; Add map file information during compilation
  ( (forward robot goal) ) ) ; Task list
```

Listing 4.9 – Motion planning problem loading external continuous map properties from external file.

4.5.2 Sampling and Potential Fields

Sampling and Potential Fields, described in Sections 2.4.3 and 2.4.4, respectively, do not need preprocessing, as they sample or compute forces based on their current position in the continuous space, differently from the roadmaps or cells that discretize space independent from agent position. Sampling combined with semantic attachments can yield the steps of a path obtained from the user specified number of samples, which are connected when within the given range, with every position sample being converted by the symbol-object table to a more readable symbol. The first call to the semantic attachment generates the actual sampling, yielding the first sample to go to while caching the path in the intermediate layer to be consumed on subsequent calls on the same area and range. The semantic attachment implementation can try to find a path more than once, adding more samples every round or changing sampling strategy to avoid returning a failure. SAMPLE semantic attachment is shown in Listing 4.10.

Potential Fields are expected to return one position for each call, which makes them suitable to be implemented by functions. However, by replacing the potential field function with a semantic attachment to generate new positions we can also add new forces at local minima dead-ends found during planning to repulse the agent, making possible to also consider the new forces when back-tracking occurs and trying a different path. Extra domain knowledge provided by the HTN can also be used to avoid narrow passages and consider dynamic obstacles.

```
(:attachments (sample ?from ?to ?goal ?minx ?maxx ?miny ?maxy ?samples ?range ?
    tries))
(:method (sampling ?agent ?goal)
  base
  ; Preconditions
  ( (at ?agent ?from) (call < (call distance ?from ?goal) 5) )
  () ; Empty subtasks
  keep_moving
  ; Preconditions
  ( (at ?agent ?from) (width ?w) (height ?h)
    (sample ?from ?to ?goal 0 ?w 0 ?h 1000 5 3) )
  ; Subtasks
  ( (!move ?agent ?from ?to) (sampling ?agent ?goal) ) )
```

Listing 4.10 – Sampling positions with a semantic attachment.

4.5.3 Dubin's path / Bitangent search

For an agent to move in a continuous space it is common practice to simplify the environment to simpler geometric shapes for faster collision evaluation [45]. One possible simplification is to find the smallest circle or sphere that contains each 2D or 3D obstacle, respectively, and use this simpler obstacle shape to evaluate paths. In this context the optimal path is the one with the shortest lines between initial position and goal, considering bitangent lines between each simplified obstacle plus the amount of arc traversed on their borders. This path is also know as Dubin's path [22]. One possible approach for a satisficing plan is to move straight to the goal or to the closest obstacle to the goal and repeat the process. Such movement requires a visible destination, without any other obstacles between current and target positions. A second consideration is the entrance and exit rotation direction, as clock or counter-clockwise, to avoid cusped edges. Cusped edges are not part of optimal realistic paths, as the moving agent would have to turn around over a single point instead of changing its direction a few degrees to either side. For the problem defined in Figure 4.8 the possible paths from point *i* to *g* are ACG, ADH, BEG and BFH, other paths contain unnecessary movements or cusped edges, such as the AEH path, with two cusped edges.

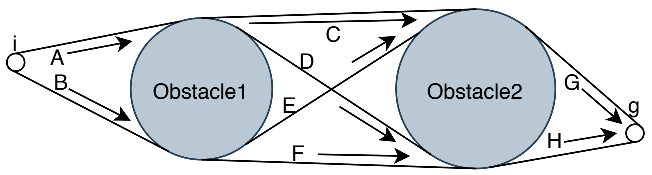

Figure 4.8 – Possible bitangent paths from *i* to *g* with two circular obstacles.

Two possible approaches can be taken to solve the search over circular obstacles using bitangents. One is to rely on an external solver to compute the continuous movement actions, a motion planner, which could happen after or during HTN decomposition has taken place. If computed after HTN decomposition, one must replace certain dummy actions of the HTN plan with actual movement actions and replan in case of failure. If computed during HTN decomposition one must be able to either access the entire solver or parts of it.

The first approach delegates the entire search to an external implementation to compute the path. The external implementation can be implemented as a function, and used by the HTN as the SEARCH-CIRCULAR function implemented in Listing 4.11. A *forward* method calls the external solver when the agent is not at the goal already. The SEARCH-CIRCULAR function can return or externally store the resulting steps of the path plan. The plan found by an external solver can be stored in the intermediate layer as a list to be consumed by later function calls. The *apply-plan* method recursively applies movement actions until no more positions, retrieved by the *plan-position* function based on their index, are stored in the external list of size *plan-size*. The HTN planner can try to use this path plan, but it is impossible to ask for a new plan if it fails, as functions return a value once. The external functions could be modified to return many or all possible path plans. This is not only complex to describe, but also computing intensive, as many paths are unnecessary for most cases. The external solver should only compute bitangents as required, as bitangent precomputation takes a significant amount of time for scenarios with many obstacles.

```
(:method (forward ?agent ?goal)                     ; Call external solver
  base
  ( (at ?agent ?goal) )                             ; Preconditions
  ()                                                ; Empty subtasks
  search
  (                                                 ; Preconditions
    (at ?agent ?start)                              ; Plan stored outside HTN
    (call search-circular ?agent ?start ?goal) )    ; External solver call
  ( (apply-plan ?agent ?start 0 (call plan-size)) )); Subtask

(:method (apply-plan ?agent ?from ?index ?size)     ; Consume external plan
  index-equals-size
  ( (call = ?index ?size) )                         ; Preconditions
  ()                                                ; Empty subtasks
  get-next-action
  ( (assign ?to (call plan-position ?index)) )      ; Preconditions
  (                                                 ; Subtasks (move and repeat)
    (!move ?agent ?from ?to)
    (apply-plan ?agent ?to (call + ?index 1) ?size) ) )
```

Listing 4.11 – Search over circular obstacles using bitangents is done entirely by external function and resulting plan steps stored in intermediate layer are consumed by the HTN.

The second approach interleaves HTN and external calls to better deal with backtracking, which requires external calls to not return a value once, but new values as needed. In cases like this we can use semantic attachments to compute values on-demand. An implementation of the second approach is presented in Listing 4.12. The second approach relies on parts of the external solver to describe continuous search to the HTN planner, namely a VISIBLE function and a CLOSEST SA. The VISIBLE function returns true if from a point on a circle there is a straight line to the goal without obstacles, and false otherwise. The CLOSEST semantic attachment generates unifications from a circle with an entrance direction to a point in another circle with an exit direction, new points closer to the goal are generated first. Our initial task maps to the *forward-attachments* method, which decomposes to a search *loop* in one of two directions, *clockwise* or *counter-clockwise*. The *loop* method is very similar to a discrete search on a grid, the only difference is the *direction* and use of CLOSEST instead of ADJACENT SA. Both *loop* method cases are used to move around circular obstacles using bitangents, keeping the entrance and exit direction to avoid cusped edges between the circles. Instead of generating all bitangents between start, goal and obstacles before search, we can exploit knowledge of the goal position to generate bitangents that are closer to the goal, performing a greedy search. The CLOSEST semantic attachment unifies three variables: *?out_circle* with the target circle, *?to* with a point on the target circle, and *?out_dir* with the exit rotation direction. Differently from the external solver, one can deal with failure at any moment, while being able to modify behavior with the same external parts, such as the initial direction the search starts with in the *loop* method, or to limit or prioritize overtakes to the left or right of moving obstacles. Another advantage over the original solution is the ability to ask for N plans, which forces the HTN to backtrack after each plan is found and explore a different path until the number of plans found equals N, or the HTN planner fails to backtrack and obtain more plans.

```
(:attachments (closest ?circle ?to ?out_circle ?in_dir ?out_dir ?goal))
(:method (forward-attachments ?agent ?goal)
  clockwise
  ((at ?agent ?start))                               ; Precondition
  ((loop ?agent ?start ?start clock ?goal))          ; Subtask
  counter-clockwise
  ((at ?agent ?start))                               ; Precondition
  ((loop ?agent ?start ?start counter ?goal))        ; Subtask
)

(:method (loop ?agent ?from ?circle ?in_dir ?goal)
  base
  ((call visible ?from ?circle ?goal))               ; Precondition
  ((!move ?agent ?from ?goal))                       ; Subtask
  recursion
  (                                                  ; Preconditions
    (closest ?circle ?to ?out_circle ?in_dir ?out_dir ?goal)
    (not (visited ?agent ?to)) )
  (                                                  ; Subtasks
    (!move ?agent ?from ?to)
    (!!visit ?agent ?from)
    (loop ?agent ?to ?out_circle ?out_dir ?goal)
    (!!unvisit ?agent ?from) ) )
```

Listing 4.12 – Search over circular obstacles using bitangents is done by the HTN using the CLOSEST semantic attachment to generate each step.

4.6 Temporal Planning

Although there are steps to reproduce classical planning time constraints in HTN, as seen in Section 2.3, there are no approaches to include a time variable and exploit HTN planning to test multiple durations. Based on the ITERATOR semantic attachment, presented in Section 3.7.2, we exploit durative actions in HTN descriptions.

4.6.1 Generator

In the Generator domain, first described by Howey and Long [44], one can generate energy based on the amount of fuel available in each generator's tank. Two actions are available in this domain: *generate* and *refuel*. These two actions are durative actions, as their effects are based on a time variable that matches the duration of the action. Both actions act on the fuel level of each generator tank and are constrained by the capacity of fuel each generator tank can store.

Here we use the linear version of the Generator domain, based on the one from DiNo [1], shown in Listing 4.13, where the amount of fuel being added or removed from each generator tank matches the behavior of a linear function. The linear version is easier to compute, debug and compare than the original version, as more planners are able to support linear function descriptions than otherwise.

```
(define (domain generator2)
(:requirements :fluents :durative-actions :duration-inequalities :adl :typing)
(:types generator tank)
(:predicates (refueling ?g - generator) (generator-ran) (available ?t - tank))
(:functions (fuelLevel ?g - generator) (capacity ?g - generator) )
(:durative-action generate
 :parameters (?g - generator)
 :duration (= ?duration 1000)
 :condition (over all (>= (fuelLevel ?g) 0))
 :effect (and (decrease (fuelLevel ?g) (* #t 1))
              (at end (generator-ran))))
(:durative-action refuel
 :parameters (?g - generator ?t - tank)
 :duration  (= ?duration 10)
 :condition (and (at start (available ?t))
                 (over all (< (fuelLevel ?g) (capacity ?g))))
 :effect (and (at start (refueling ?g))
              (increase (fuelLevel ?g) (* #t 2))
              (at start (not (available ?t)))
              (at end (not (refueling ?g))) )) )
```

Listing 4.13 – Linear Generator domain in PDDL from DiNo.

The Generator domain can be converted to UJSHOP, as durative actions are split in two actions, start and finish, following the steps from Goldman [38], presented in Section 2.3. The PDDL 2.1 functions used in the durative-actions are simulated in the intermediate layer to consider timespans. The user is responsible for describing how processes affect the value of functions queried at specific points in time. Function constraints are simulated using an extension to JSHOP protections, by describing named expressions as axioms and adding or removing protections over such axioms during planning. A function queried at a time *t* must never break any protected axioms, which are evaluated replacing their time variable with *t*, as this is the same as failing an action precondition and will force backtracking. The intermediate layer complex parts related to functions and processes are contained in the basic library, making such constructions trivial to be supported by new domains. The user must only describe the STEP semantic attachment and *identity* and *double* functions, used by the generate and refuel processes to decrease fuel at the same rate as it generates or to add fuel twice as fast as the action duration. This is equivalent to the PDDL 2.1 expressions, externally defined as UJSHOP was not extended to support specific temporal constructions. The final UJSHOP description of the Generator domain is shown in Listing 4.14.

The domain description is much longer than the classical one, but all mechanisms are visible and their intention is clear. On the other side, DiNo's documentation supplies the user with specific

[1] https://github.com/KCL-Planning/DiNo/blob/master/ex/linear-generator/generator.pddl

```
(defdomain generator (
  (:attachments (step ?t ?min ?max ?step))
  (:- (enough-fuel ?g ?t) (call >= (call function (fuelLevel ?g) ?t) 1000))
  (:- (minimum-fuel ?g ?t) (call >= (call function (fuelLevel ?g) ?t) 0))
  (:- (maximum-fuel ?g ?t)
    (call <= (call function (fuelLevel ?g) ?t) (call function (capacity ?g))))
  (:operator (!generate-start ?g ?start ?finish)
    ( (minimum-fuel ?g ?start) )
    ()
    ( (protect-axiom minimum-fuel ?g)
      (call process decrease (fuelLevel ?g) identity ?start ?finish) ) )
  (:operator (!generate-finish ?g ?start ?finish)
    ( (minimum-fuel ?g ?finish) )
    ( (protect-axiom minimum-fuel ?g) )
    ( (generator-ran) ) )
   ; ... Certain operators and methods were omitted due to space
  (:method (multiple-refuel ?g ?start)
    enough-fuel
    ( (enough-fuel ?g ?start) )
    ()
    add-fuel
    ( (tank ?t)
      (available ?t)
      (assign ?finish (call + ?start 10)) )
    ( (!refuel-start  ?g ?t ?start ?finish)
      (!refuel-finish ?g ?t ?start ?finish)
      (multiple-refuel ?g ?finish) ) )
  (:method (refuel-and-generate ?g)
    ()
    ( (multiple-refuel ?g 0) (generate ?g) ) ) ))
```

Listing 4.14 – Linear Generator domain in UJSHOP.

execution values to set the PDDL parser time quantum (a time step), real scale and real fraction digits (value resolution), and such values are not trivial to guess. Executing both DiNo, with the supplied values, and our approach over the problems from DiNo's repository, we obtain the planning times of Figure 4.9. Our approach does get slower as more tanks are available in the more complex problems, but the entire batch of problems takes less time than the simplest problem being solved by DiNo (e.g Problem 20 takes 0.0053s). We ignore the setup time required by both planners in this comparison, as both require file generation, compilation or interpreter loading.

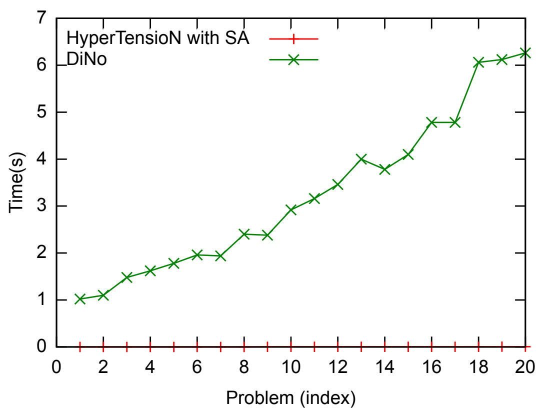

Figure 4.9 – Time in seconds to solve Linear Generator problems.

4.6.2 Car Linear

In the Car Linear domain [8] the goal is to control the acceleration of a car, which is constrained to a minimum and maximum speed, without external forces applied, only moving through one axis to reach its destination, and requiring low speed to safely stop. The idea is to propagate process effects to state functions, in this case acceleration to velocity and velocity to position, while being constrained to an acceptable speed and acceleration. The planner must decide when and for how long to increase or decrease acceleration, therefore becoming a temporal planning problem. We use the STEP semantic attachment to iterate over the time variable and propagate temporal effects and constraints, i.e. speed at time *t*. The strategy implemented in the Car Linear HTN domain, Listing 4.15, is to briefly accelerate to obtain speed and later to stop, leaving the semantic attachment responsible to find the amount of time required to obtain the required displacement, as shown in Figure 4.10. In this domain all state properties are functions, i.e. numeric values, with the speed limit constraint in place during the entire planning time. Functions and speed constraint are initialized by the initial state, as seen in Listing 4.16, with a single *forward* task with the destination minimum and maximum limits. Note that the destination is not a single value to deal with rounding errors that may happen during planning with non-integer numerical values and different time steps.

We compare the execution time of our approach with ENHSP with `aibr`, ENHSP main configuration for planning with autonomous processes, in Table 4.1. There is no comparison with a native HTN approach, as one would have to add a discrete finite set of time predicates (e.g. ⟨*time 0*⟩) to the initial state description to be selected as time points during planning.

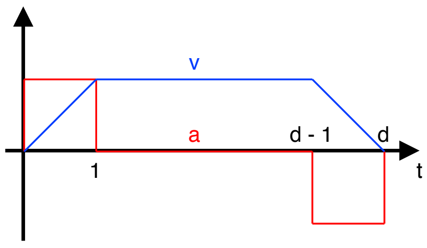

Figure 4.10 – In the Car Linear domain the car acceleration *a* can be controlled to achieve an acceptable velocity *v* to obtain a displacement *d* and reach the destination, within an error margin, in less time while respecting the domain constraints.

Problem	1	2	3	4	5	6	7	8	9
ENHSP (aibr)	0.484	0.432	0.411	0.443	0.461	0.474	0.465	0.436	63.585
HTN with SA	0.016	0.019	0.014	0.016	0.018	0.019	0.017	0.018	01.402

Table 4.1 – Time in seconds to solve Car Linear problems.

```
(defdomain car_linear (
  (:attachments (step ?t ?min ?max ?step))
  (:- (speed_limit ?time) ; Constraint to axiom, protected since initial state
    (and (assign ?vt (call function v ?time))
         (call >= ?vt (call - 0 (call function max_speed)))
         (call <= ?vt (call function max_speed)) ) )
  (:operator (!start_car ?start ?finish)
    ((not (engine_running)))
    ()
    ( (engine_running)
      (call process increase d displacement ?start ?finish)
      (call process increase v moving_custom ?start ?finish) ) )
  (:operator (!stop_car ?time)
    ( (engine_running)
      (assign ?vt (call function v ?time))
      (call > ?vt -0.1) (call < ?vt  0.1) (call = (call function a ?time) 0) )
    ((engine_running))
    ((call event assign v 0 ?time)) )
  (:operator (!accelerate ?time)
    ( (engine_running)
      (call < (call function a ?time) (call function max_acceleration)) )
    ()
    ((call event increase a 1 ?time)) )
  (:operator (!decelerate ?time)
    ( (engine_running)
      (call > (call function a ?time) (call function min_acceleration)) )
    ()
    ((call event decrease a 1 ?time)) )
  (:operator (!!test_destination ?min_destination ?max_destination ?time)
    ( (assign ?d (call function d ?time))
      (call >= ?d ?min_destination) (call <= ?d ?max_destination) ) () () )
  (:method (forward ?min_destination ?max_destination)
    base
    ()
    ( (!!test_destination ?min_destination ?max_destination 0) )
    keep_moving
    ( (step ?deadline 3) )
    ( (!start_car 0 ?deadline)
      (!accelerate 0)
      (!decelerate 1)
      (!decelerate (call - ?deadline 1))
      (!accelerate ?deadline)
      (!stop_car ?deadline)
      (!!test_destination ?min_destination ?max_destination ?deadline) ) ) ))
```

Listing 4.15 – Car Linear domain in UJSHOP.

```
(defproblem pb0 car_linear
  ( ; Initial state
    (function d 0) (function v 0) (function a 0)
    (function min_acceleration -10) (function max_acceleration 10)
    (function max_speed 10)
    (protect_axiom speed_limit) )
  ( (forward 999.5 1010.5) ) ) ; Task list
```

Listing 4.16 – Car Linear problem in UJSHOP.

4.7 Rules and Behaviors

Certain domains, such as games and driving vehicles, force agents to act within certain rules to match what is an expected behaviour by the other agents, otherwise the planning process becomes extremely complex or impossible, decreasing the chances of agents obtaining their goals. Certain rules are deterministic and clear, while others may be subjective to what one agent may perceive about the environment or their interpretation about the rule description, forcing them to behave in unexpected ways in corner cases not clearly specified.

Humans can deal with ambiguous or incomplete rules in a day-to-day basis using experience and analyzing the behavior of others to complete missing information about how to act in the current situation. However, the majority of robots lack the sensors, processing power or models to do the same as humans in such scenarios. The problem becomes more visible as robots share spaces recently occupied only by humans, as autonomous cars, drones and boats must adhere to preexisting rulesets not made for them to coexist with human drivers. One such ruleset are COLREGS [13], which stands for COLlision REGulationS, which focus on navigation rules to be followed by ships and vessels to avoid collision hazards with static and moving objects. The main idea behind COLREGS is to have a set of common rules to be used internationally, although some rules may be different according to region or context, such as competitive sailing at sea. The regulatory content covers many aspects of sailing, such as visibility, which are hard to measure and predict, while other aspects better match rules to plan with, such as overtaking.

With motion and temporal planning it is possible to describe how trajectory and speed can be controlled by a plan to carry out such maneuvers while complying with the COLREGS, maintaining a behavior that does not jeopardize other vessels. The aspects more interesting for planning are:

- **Overtake**: catch up with and pass another vessel in the same direction, sometimes competing for space;
- **Head-on and crossing situations**: consider other vessel direction and speed to avoid collision;
- **Never stop in narrow channels**: consider position to gather and process data, simplify collision avoidance with other vessel;
- **Follow traffic schemes (lanes)**: simplify collision avoidance by using a preset direction and speed in some regions.
- **Minimize speed to gather more data or avoid hazard**: consider position and speed during planning; and
- **Consider current, wind and depth during maneuvers**: physical properties must be considered as they impact the planning process.

Such aspects may generate conflicting behaviors during execution, which must be consistent with the common sense of a human to be effective [4] in an environment shared by autonomous and manually controlled vessels. Some of these aspects are more interesting for offline planning, simulated before the event, to plan data gathering on a routine scanning from a lake or river. Other aspects are more interesting to be done online, taking advantage of all the sensors on board of the robotic vessel while constrained to limited computing power within the vessel.

By combining the previous implementations of the Bitangent search domain, from Section 4.5.3, and the Car Linear domain, from Section 4.6.2, it is possible to obtain a new domain in which we can plan in the continuous space while reasoning about temporal events and processes. With motion and temporal planning aspects combined it is possible to describe the COLREGS aspects previously selected. We can have a *goto* method to reach a goal position with a certain orientation angle. If the goal position is nearby only the orientation must be adjusted. Otherwise the planner decomposes the *forward-attachments* method to select which overtaking rotation direction to start with, clockwise or counter-clockwise. The *loop* method will try to move to a visible goal position, using the same strategy of the Car Linear domain, or keep moving towards the goal selecting visible obstacles that are closer to the goal and repeating this process until the goal is reached or failure is returned. Part of this domain is shown in Listing 4.17.

Head-on trajectories can use an expanded obstacle radius in the vessel coming towards our agent, forcing it to move away from the collision route faster, as seen in Figure 4.11. For safety reasons avoiding head-on vessels is usually done in only one rotation direction, starboard (counter-clockwise), which removes one case from the *forward-attachment* and will require an extra flag to set which directions are allowed in the CLOSEST semantic attachment. Adding the *?time* variable to the CLOSEST semantic attachment will also enable moving obstacles, vessels or not, to have their position queried at specific times based on their expected routes during planning time. An extra case could be considered to change speed to minimize turns and safely deal with crossing situations. Overtaking forces our agent to align with the other vessel to an angle in which it avoids a crossing situation while maintaining distance to increase safety, as the other vessel may not be aware of our presence and may change speed and direction without notice.

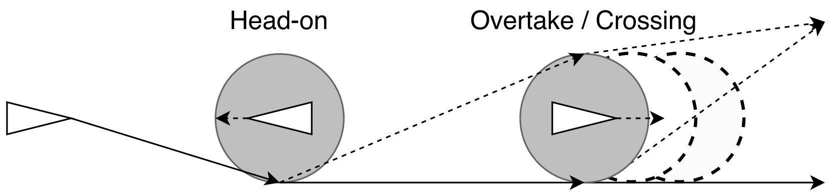

Figure 4.11 – Head-on and overtaking situations force certain behaviors to match the COLREGS ruleset. The head-on collision avoidance behavior will always force the agent to select the starboard side, while overtaking leave the agent free to select a direction to avoid crossing in front of the other vessel (dashed line) or move further ahead (straight line).

Terrain details can be considered by the CLOSEST semantic attachment. Terrain details include a preference for wider channels, delaying narrow channels alternatives as they require the agent to keep moving and not block the passage, which increases the description complexity with guard cases. If the terrain follows a traffic scheme there is no need to use the *goto* method, just add a sequence of landmark points as subgoals and a method that forces the agent to reach each landmark in the sequence. External forces, such as current and wind, influence the position function in the same way as the acceleration process, but are outside the control of any agent action. Such forces can be emulated by a local database of expected external forces at specific positions and times to better model this domain.

```
(:attachments (step ?t ?min ?max ?step)
              (closest ?circle ?to ?out_circle ?in_dir ?out_dir ?goal) )
(:method (goto ?robot ?goal ?angle ?time)
  base
  ( (assign ?pos (call function (pos ?robot) ?time))
    (assign ?goal_angle (call atan ?pos ?goal))
    (call = (call distance ?goal ?pos) near) )
  ( (turn ?robot ?goal_angle ?angle ?time) )
  last-move
  ( (assign ?pos (call function (pos ?robot) ?time))
    (call != (call distance ?goal ?pos) near)
    (assign ?goal_angle (call atan ?pos ?goal)) )
  ( (!!set ?robot ?pos ?time)
    (forward-attachments ?robot ?goal ?pos ?angle ?time) ) )
(:method (forward-attachments ?robot ?goal ?pos ?angle ?time)
  ;clockwise
  ;((at ?robot ?pos ?time))
  ;((loop ?robot ?start ?start clock ?goal ?pos ?angle ?time))
  counter-clockwise
  ((at ?robot ?pos ?time))
  ((loop ?robot ?start ?start counter ?goal ?pos ?angle ?time)) )
(:method (loop ?robot ?from ?circle ?in_dir ?goal ?pos ?angle ?time)
  base
  ( (call visible ?from ?circle ?goal)
    (assign ?current_angle (call function (a ?robot) ?time))
    (assign ?goal_angle (call atan ?pos ?goal))
    (step ?deadline (call + ?time 1)) )
  ( (turn ?robot ?current_angle ?goal_angle ?time)
    (!!start_movement ?robot ?time ?deadline)
    (!increment_velocity_x ?robot ?time)
    (!decrement_velocity_x ?robot ?deadline)
    (!!test_destination ?robot ?pos ?deadline)
    (turn ?robot ?goal_angle ?angle ?deadline) )
  recursion
  ( (step ?deadline (call + ?time 1))
    (closest ?circle ?to ?out_circle ?in_dir ?out_dir ?goal)
    (not (visited ?robot ?to))
    (assign ?goal_angle (call atan ?pos ?to)) )
  ( (turn ?robot ?current_angle ?goal_angle ?time)
    (!!start_movement ?robot ?time ?deadline)
    (!increment_velocity_x ?robot ?time)
    (!decrement_velocity_x ?robot ?deadline)
    (!!visit ?robot ?to)
    (loop ?robot ?to ?out_circle ?out_dir ?goal ?pos ?angle ?deadline)
    (!!unvisit ?robot ?to) ) )
```

Listing 4.17 – HTN domain combining Bitangent search and Car Linear strategy to plan in the continuous space while also considering temporal effects of the acceleration process.

5. RELATED WORK

We selected related work about planning to show how other researchers are dealing with the symbolic-geometric data and the large state space created by combinatorial explosion. We focus on symbolic-geometric planners as their application is closely related to robotics and the common problems of this domain.

Before we cite related work, it is interesting to categorize the different approaches symbolic-geometric planners may take. According to Schüller et al. [71] we can categorize planners based on how different strategies focus on low-level checks about geometric feasibility, which happen:

Before planning Preprocessing geometric data for the planning stage;

During planning Making calls between the symbolic and external layers;

After planning Obtaining many plans and filtering impossible ones;

After planning Obtaining a single plan, and replanning with new constraints after each failure.

We can also focus on which layer takes control, expanding the previous categories, according to Gharbi et al. [37]:

Symbolic layer calls external layer The symbolic layer is responsible for the search and asks the external layer about the geometric feasibility of actions (**during** and **after planning**);

External layer calls symbolic layer The external layer is responsible for the movements and asks the symbolic layer about which paths to take (**before planning**);

Sample in the compound space Search is controlled by both layers simultaneously (not defined by Schüller et al. [71]).

Different problems can be better solved by certain strategies provided by such categories, e. g. robots that do not change the environment geometry (only movement) or change in a limited way (pick and place objects) can reuse geometric data for longer periods, which makes preprocessing more suitable. In our work the **symbolic layer calls the external layer**, **during planning**, which makes the process faster than preprocessing a lot of data or filtering later, which may require replanning. Since the calls happen during planning and access external libraries, it is possible to create an interface to let experts interact with the process, mostly to select the order in which the planner will consider each possible value. Replanning with new constraints makes the process more tedious for a human to interact with and more time consuming, as new constraints may force the planner to start from scratch, being unable to exploit a cache of complex previous operations.

Considering deliberation and execution, de Silva and Meneguzzi [16] propose to add more layers to an agent than the three layers used in this work. A deliberation layer is used to control the symbolic planning layer providing goals or tasks. The symbolic planning layer is connected to the geometric planning layer by anchors (similar to our symbol-object table in the intermediate layer) that are used to share data between the two parts, discretizing objects that were discovered during execution that must be visible to the symbolic layer, such as obstacles. The anchors are defined as automatic mechanisms that expose such data to the other layer when available, as a **compound state**, unlike our approach that only creates new symbols through semantic attachments or function calls. The lower levels are close to the hardware, monitoring if performed actions generate the expected results captured by sensors.

In our work the upper (deliberation) and lower (robotic devices) levels from de Silva and Meneguzzi are ignored, as they are more suited for continuous execution, with deliberation only taking place before planning, whereas sensing and acting, from the lower levels, are more suited before and after planning, respectively. If acting and sensing must happen during planning, the intermediate or external layer could access low level libraries that control robotic devices to execute and monitor such functions, but the symbolic layer must be aware that backtracking may not be possible in certain situations, as the robot progresses and use resources. Deliberation could be used to either call the lower levels to plan and execute simultaneously, or only plan, which could be used to know in advance about possible challenges and failures based on known predicates that require no further exploration.

5.1 Semantic attachments for classical planning

Dornhege et al. [20] prefer to combine symbolic and geometric planners in a classical planner, a non-hierarchical architecture, using an extended version of PDDL to define geometric calls to a classical planner. They opted to integrate the planners tightly to avoid preprocessing and replanning, with the **symbolic planner calling the geometric planner** during the planning process. They use semantic attachments to integrate both parts, checking the truth value of specific predicates by a trajectory planner. Two types of semantic attachments are defined: *condition checker* for complex preconditions, and *effect applicator* to compute the numerical state variables in effects. The PDDL description is extended further to support *grounding* and *cost modules* [19], to externally unify variables and the cost of actions. No comments are made about the limitations of a static set of objects (literal symbols) defined by PDDL problems, which impacts the number of possible predicates that trigger calls to a geometric planner, and in a dynamic environment where dummy PDDL objects must be defined, as they may or may not be used, and how this approach could affect performance. In a later work [21] a cache was added to reuse values from external procedures applied to similar previous states. The work of Dornhege et al. was very inspiring to our approach, as the negative points of a classical planner do not exist for an HTN approach. The number of objects is not static by design, as the actions applicable at any given moment come from the task decomposition procedure and have their variables constrained by their preconditions. The number of numerical and external calls is greatly minimized by following the decomposition procedure, no extra computation is required for tasks that are never decomposed. The use of coroutines to explore other paths is more suitable for HTN, as only backtracking forces the planner to ask for a new value, which triggers the coroutine to yield such new value.

5.2 STRIPStream / PDDLStream

With STRIPStream, Garret et al. [30] tried to solve some of the issues with classical planners, extending the STRIPS language [25] to support the specification of blackbox generators. As generators create infinite streams of objects and static predicates, two planning algorithms are used to reduce problems to finite versions. The first is an incremental planner, which generates planning instances based on the streams, and retrying with more objects after each failed attempt. The second is a focused planner, that starts with dummy objects to be replaced by concrete values during planning, such values being limited to the region of interest, thus focusing the generation of objects on a predefined interval. PDDLStream [31], the successor of STRIPStream, adheres to the PDDL standard [60], adding a *stream.pddl* file to define the equivalent of our intermediate layer, with input/output parameters, domain and certified facts. Domain facts act as typing information, defining legal inputs, while certified facts declare properties guaranteed to be satisfied by all stream outputs. Two new planning algorithms are added to the system, binding and adaptive, to take more advantage of the previous plans. Binding propagates stream outputs to the next stream inputs to evaluate more at each step, and Adaptive maintains a queue of bindings to repeatedly consider. All planning algorithms operate in the same way, solving a sequence of finite planning problems of increasing size. The implementation of the generators uses PyBullet [1]. The goal application of this work is geometrical tasks done by a robot in a kitchen scenario, with actions like *pick*, *place* and *scoop*, never mentioning temporal planning constructions. STRIPStream and PDDLStream have the same problem with fixed sets of objects to represent streams of data from outside solvers as Dornhege et al.

5.3 Extend the description language

Other works also extended the description language to take advantage of external calls, such as the Object-oriented Planning Language from Hertle et al. [40], with type safe mechanisms and similar to the C++ language, Functional STRIPS from Ferrer-Mestres et al. [24], which supports state constraints, functions, and numerical variables to deal with symbolic and geometric properties. Gaschler et al. [34, 33] also implemented external calls in the symbolic layer with variables that can deal with uncertainty, able to represent their state as *known*, *unknown*, *incomplete* or *computed during run time* to obtain feasible plans. Later, they implemented geometric predicates to speed up the planning process [32].

[1]pybullet.org

5.4 Geometric Task Planner

HTN-GTP [18] uses a totally-ordered HTN as the symbolic part of the system, decomposing tasks as they are defined. The symbolic part tries to find a solution using a discrete space of candidates to grasp and place objects in order to solve pick and place tasks. The Geometric Task Planner (GTP) considers agent "effort" level, discrete grasps, placement positions and orientations. The authors cite that the main advantage of GTP lies in its ability to solve a variety of common tasks related to object visibility, grasp and reachability. Their work does not analyze what happens when geometric backtracking must be performed, modifications of the geometric parts while symbolic parts of the plan are kept intact, as this may force complete replanning if symbolic constraints must be changed to allow planning on a different branch of the state space. More recent work of their research group [15, 50, 37] does not consider geometric backtracking as the original [17], using only geometric alternatives represented by action instances in the symbolic level to explore other planning branches.

5.5 Symbolic versus Geometric top-level planner

Şucan and Kavraki [77] use *low-level checks after planning* on a set of symbolic plans, looking for a feasible geometrical solution to achieve a symbolic goal. Later they expanded the work to deal with uncertainty using Markov Decision Process to guide the search [78]. Srivastava et al. [76] also use the *low-level checks after planning* approach, but once a symbolic plan is found the actions are expanded to consider the missing geometric relations, that will trigger a state update and search for a new symbolic plan in case of failure. Their following work [75] considers geometric backtracking to go back to the last feasible action in case of failure and try another alternative. The alternatives are limited to a certain number, and the reason of failure is informed to the symbolic planner. Choi and Amir [12] propose to do the opposite, use an external geometric layer to guide the search. They use a motion planner to explore the world, such as the ones seen in Section 2.4, and use the generated graph to search for feasible actions, using sets of motion between the edges of the generated graph that modify the state of objects. Such actions are then used by a symbolic planner to search for a feasible symbolic plan.

5.6 aSyMov

aSyMov [10, 39, 11, 9] is a symbolic-geometric planner created with robotic tasks in mind, it shares constraints between symbolic and geometric parts of the planner using a **compound state**. To do so, aSyMov employs a hybrid approach, the symbolic problems are solved by Metric-FF [41] while the geometric problems, such as path and manipulation, are solved by Move3d [73]. At each planning step the data from both parts, symbolic and geometric, is considered. It uses accessibility lists to solve geometric constraints alongside a roadmap, a representation of the configuration space (the geometric state). The Probabilistic Roadmap Methods (PRMs) aSyMov relies on are considered an efficient approach for highly dimensional motion planning problems. aSyMov works as a forward search planner in the state space, exploring from the current state to the goal. When the goal is reached the plan found is **post-processed** to add the geometric information required. This means that the planner solves a purely symbolic version of the problem, ignoring geometric relations until a symbolic plan is achieved. When backtrack happens, the system is forced to acknowledge that constraints exist in the real-world and a heuristic estimator is used to select the closer-to-goal state from the reachable states before **replanning**. The system also decides if more exploration of possible roadmaps is required before selecting a state. The state is represented by three parts: a purely symbolic, an interface between symbolic and geometric constraints, such as positions, and a purely geometric part with the combinations that can be reached without collision. One of our first goals was to take advantage of unlimited symbol generation at run-time instead of having a limited number of symbols to connect the symbolic layer with the external layer. Creating such position objects during planning could improve the performance of the planner, while removing the user's burden to describe all position objects that may be used along the search process.

5.7 Comparison

The related work cited in this chapter is organized in Table 5.1, to be better compared with our work. We use the categories presented in the beginning of this chapter and a few more to compare each approach. The ability to call external procedures during planning (7), as functions or semantic attachments, to solve specific subproblems that cannot be represented as easily in the planning description language. When only symbolic or geometric state is kept in memory the planner will derive the other state representation from it (8 and 9), which may incur in some processing penalty to save memory and avoid a complex internal representation that is forced to keep both symbolic and geometric states consistent. Some planners may have a geometric solver that present several possibilities to continue planning (10), some may even be able to backtrack only the geometric part of the current solution to obtain a feasible plan (11). Each approach present several features according to their planning strategies, we selected the following for comparison:

1. Symbolic layer calls external/geometric layer;
2. External/Geometric layer calls the symbolic layer;
3. Sample in the compound state;
4. From a first symbolic plan find a geometrically feasible solution;
5. Ensure geometric feasibility while computing the symbolic plan;
6. Find a geometrically feasible plan among all symbolic plans;
7. Call external procedures;

8. Compute geometric states from symbolic states;
9. Create symbolic knowledge from geometry;
10. Geometric alternatives;
11. Geometric backtracking;

Table 5.1 – Related work sorted by research groups, our work is in the last row of the table.

Work	1	2	3	4	5	6	7	8	9	10	11
Dornhege et al. (2009) [20]	X				X		X		X		
Dornhege et al. (2012) [19]	X				X		X		X		
Hertle et al. (2012) [40]	X				X		X		X		
Dornhege et al. (2013) [21]	X				X		X		X		
Garret et al. (2017) [30]				X		X	X		X		
Garret et al. (2020) [31]				X		X	X		X		
Ferrer-mestres et al. (2015) [24]	X				X		X				
Gaschler et al. (2013) [34]	X				X		X				
Gaschler et al. (2013) [33]	X				X		X				
Gaschler et al. (2015) [32]	X				X		X				
de Silva et al. (2013) [17]	X				X		X		X	X	X
de Silva et al. (2013) [18]	X				X		X			X	
de Silva et al. (2014) [15]	X				X		X			X	
Lallement et al. (2014) [50]	X				X		X			X	
Gharbi et al. (2015) [37]	X				X		X			X	
Șucan and Kavraki (2011) [77]	X					X					
Șucan and Kavraki (2012) [78]	X					X					
Srivastava et al. (2013) [76]	X			X					X		
Srivastava et al. (2014) [75]	X			X				X	X	X	X
Choi and Amir (2009) [12]		X							X	X	
Cambon et al. (2003) [10]			X					X	X		
Gravot et al. (2003) [39]			X					X	X		
Cambon et al. (2004) [11]			X					X	X		
Cambon et al. (2009) [9]			X					X	X		
Our work	X				X		X		X	X	

Most systems seen in this chapter use a static description with a constant number of symbols to be used during planning, as PDDL, or address geometric details offline, precomputing as much as possible. This static view does not reflect the dynamic world we expect autonomous robots to work in. We believe it would be more efficient to create and destroy symbols as perceptions are received by the robot, or new environment constraints are found. We also believe that our approach of semantic attachments and symbol-object table adds a lot of flexibility while introducing few elements to a description language already understood by a domain designer. The semantic attachments can be introduced gradually to already working HTN domains, making the process easier to understand for new users, instead of requiring multiple setup steps to recreate a domain in a new language.

6. CONCLUSION

We developed a notion of semantic attachments for HTN planners that not only allows a domain expert to easily define external numerical functions for real-world domains, but also provides substantial improvements on planning speed over comparable classical planning approaches. The use of semantic attachments improves the planning speed as one can express a potentially infinite state representation with procedures that can be exploited by a strategy described as HTN tasks. As only semantic attachments present in the path decomposed during planning are evaluated, a smaller amount of time is required compared to approaches that precompute every possible value during operator grounding. Our description language is arguably more readable than the commonly used strategy of developing a domain specific planner with customized heuristics, or attaching procedures that must have all variables ground [36, chapter 11]. Specifically, we allow designers to easily define external functions in a way that is readable within the domain knowledge encoded in HTN methods at design time, and also dynamically generate symbolic representations of external values at planning time, which makes generated plans easier to understand. We only exploited semantic attachments in preconditions, but there is no limitation to add special constructions to generate effects and define costs through external calls. We believe that the external layer must keep its state separate from the symbolic state and unify variables in a way that already minimizes effort (which impacts cost), as HTN-GTP [18].

Our work is the first attempt at defining the syntax and operation of semantic attachments for HTNs, allowing further research on search in SA-enabled domains within HTN planners. This kind of extension comes in line with recent attempts at including more expressive planning languages than vanilla PDDL, and which include more functional elements in it (e.g. Tarski[1]). Dornhege et al. [20] uses semantic attachments to compute the truth value of facts in preconditions, and effects on numerical fluents that would be too complex to describe otherwise. Dornhege et al. [21] adds action grounding and action costs as semantic attachment modules. By contrast, our method can be used to both compute the truth value of ground external predicates or to unify free variables with values in preconditions, better reflecting the lifted reasoning already used by planners. Such unified values can be symbolic or numeric, and can be generated indefinitely, which is useful when searching for a numeric value that satisfies some property as one can define the equivalent of an iterator as a predicate. Further the grounding action approach [21] does not match the HTN domain style, where most operator parameters (groundings) are decided by method preconditions before decomposing to primitive tasks.

Future work includes implementing a standard cache mechanism to reuse previous values from external procedures applied to similar previous states [21] and a generic construction to access such values in the symbolic layer description, to obtain, add or remove data from explored branches outside the state structure, i.e. to hold mutually exclusive predicate information. We plan to develop more domains, with varying levels of domain knowledge and semantic attachments usage, to obtain

[1]github.com/aig-upf/tarski

better comparison with other planners and their resulting plan quality. The advantage of being able to exploit external implementations conflicts with the ability to incorporate such domain knowledge into heuristic functions, as such knowledge is outside the description. Further work is required to expose possible metrics from semantic attachments to heuristic functions, which recently have been explored by HTN planners [43].

BIBLIOGRAPHY

[1] Amato, N. M.; Bayazit, O. B.; Dale, L. K. “OBPRM: An obstacle-based PRM for 3D workspaces”, *Robotics: the algorithmic perspective*, 1998, pp. 155–168.

[2] Anderson, T. L.; Donath, M. “Animal behavior as a paradigm for developing robot autonomy”, *Robotics and autonomous systems*, vol. 6–1-2, 1990, pp. 145–168.

[3] Barraquand, J.; Langlois, B.; Latombe, J.-C. “Numerical potential field techniques for robot path planning”, *IEEE transactions on systems, man, and cybernetics*, vol. 22–2, 1992, pp. 224–241.

[4] Benjamin, M. R.; Curcio, J. A. “COLREGS-based navigation of autonomous marine vehicles”. In: Autonomous Underwater Vehicles, 2004 IEEE/OES, 2004, pp. 32–39.

[5] Bhattacharya, P.; Gavrilova, M. L. “Roadmap-based path planning-using the voronoi diagram for a clearance-based shortest path”, *IEEE Robotics & Automation Magazine*, vol. 15–2, 2008, pp. 58–66.

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

[7] Bonnet, B.; Geffner, H. “HSP: Heuristic search planner”, 1998.

[8] Bryce, D.; Gao, S.; Musliner, D. J.; Goldman, R. P. “SMT-Based Nonlinear PDDL+ Planning.” In: AAAI, 2015, pp. 3247–3253.

[9] Cambon, S.; Alami, R.; Gravot, F. “A hybrid approach to intricate motion, manipulation and task planning”, *The International Journal of Robotics Research*, vol. 28–1, 2009, pp. 104–126.

[10] Cambon, S.; Gravot, F.; Alami, R. “Overview of aSyMov: Integrating motion, manipulation and task planning”. In: Intl. Conf. on Automated Planning and Scheduling Doctoral Consortium, 2003.

[11] Cambon, S.; Gravot, F.; Alami, R. “A robot task planner that merges symbolic and geometric reasoning”. In: Proceedings of the 16th European Conference on Artificial Intelligence, 2004, pp. 895–899.

[12] Choi, J.; Amir, E. “Combining planning and motion planning”. In: Robotics and Automation, 2009. ICRA’09. IEEE International Conference on, 2009, pp. 238–244.

[13] Commandant, U. “International regulations for prevention of collisions at sea, 1972 (72 COLREGS)”, *US Department of Transportation, US Coast Guard, Commandant Instruction M*, vol. 16672, 1999.

[14] Cushing, W.; Kambhampati, S.; Weld, D. S.; et al.. "When is temporal planning really temporal?" In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007, pp. 1852–1859.

[15] De Silva, L.; Gharbi, M.; Pandey, A. K.; Alami, R. "A new approach to combined symbolic-geometric backtracking in the context of human-robot interaction". In: Robotics and Automation (ICRA), 2014 IEEE International Conference on, 2014, pp. 3757–3763.

[16] de Silva, L.; Meneguzzi, F. "On the design of symbolic-geometric online planning systems". In: 2015 Workshop on Hybrid Reasoning (HR 2015), 2015, pp. 1–8.

[17] de Silva, L.; Pandey, A. K.; Alami, R. "An interface for interleaved symbolic-geometric planning and backtracking". In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on, 2013, pp. 232–239.

[18] de Silva, L.; Pandey, A. K.; Gharbi, M.; Alami, R. "Towards combining HTN planning and geometric task planning". In: Robotics: Science and Systems Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications, 2013.

[19] Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; Nebel, B. "Semantic attachments for domain-independent planning systems". In: *Towards Service Robots for Everyday Environments*, Springer, 2012, pp. 99–115.

[20] Dornhege, C.; Gissler, M.; Teschner, M.; Nebel, B. "Integrating symbolic and geometric planning for mobile manipulation". In: Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on, 2009, pp. 1–6.

[21] Dornhege, C.; Hertle, A.; Nebel, B. "Lazy evaluation and subsumption caching for search-based integrated task and motion planning". In: IROS workshop on AI-based robotics, 2013.

[22] Dubins, L. E. "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents", *American Journal of mathematics*, vol. 79–3, 1957, pp. 497–516.

[23] Fabri, A.; Giezmann, G.-J.; Kettner, L.; Schönherr, S. "On the design of CGAL the computational geometry algorithms library", *Technical report/Departement Informatik, ETH Zürich*, vol. 291, 1998.

[24] Ferrer-Mestres, J.; Frances, G.; Geffner, H. "Planning with state constraints and its application to combined task and motion planning". In: Proceedings of Workshop on Planning and Robotics (PLANROB), 2015, pp. 13–22.

[25] 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.

[26] Finkel, R. A.; Bentley, J. L. “Quad trees a data structure for retrieval on composite keys”, *Acta informatica*, vol. 4–1, 1974, pp. 1–9.

[27] Fox, M.; Long, D. “PDDL+: Modeling continuous time dependent effects”. In: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, 2002, pp. 34.

[28] Fox, M.; Long, D. “PDDL 2.1: An extension to PDDL for expressing temporal planning domains”, *Journal of Artificial Intelligence Research (JAIR)*, 2003.

[29] Frances, G.; Geffner, H. “Modeling and computation in planning: Better heuristics from more expressive languages.” In: ICAPS, 2015, pp. 70–78.

[30] Garrett, C. R.; Lozano-Pérez, T.; Kaelbling, L. P. “STRIPS Planning in Infinite Domains”, *ICAPS Workshop on Planning and Robotics (PlanRob)*, 2017.

[31] Garrett, C. R.; Lozano-Pérez, T.; Kaelbling, L. P. “PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning”. In: International Conference on Automated Planning and Scheduling (ICAPS), 2020.

[32] Gaschler, A.; Kessler, I.; Petrick, R. P.; Knoll, A. “Extending the knowledge of volumes approach to robot task planning with efficient geometric predicates”. In: Robotics and Automation (ICRA), 2015 IEEE International Conference on, 2015, pp. 3061–3066.

[33] Gaschler, A.; Petrick, R.; Kröger, T.; Khatib, O.; Knoll, A. “Robot task and motion planning with sets of convex polyhedra”. In: Robotics: Science and Systems (RSS) Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications, 2013.

[34] Gaschler, A.; Petrick, R. P.; Giuliani, M.; Rickert, M.; Knoll, A. “KVP: A knowledge of volumes approach to robot task planning”. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on, 2013, pp. 202–208.

[35] Gehrels, B.; Lalande, B.; Loskot, M.; Wulkiewicz, A. “Boost Geometry: a generic geometry library”, *Software electronically available at http://www.boost.org/libs/geometry*, 2013.

[36] Ghallab, M.; Nau, D.; Traverso, P. “Automated planning: theory & practice”. Elsevier, 2004, 635p.

[37] Gharbi, M. “Geometric reasoning and planning in the context of human robot interaction”, Ph.D. Thesis, INSA de Toulouse, 2015, 179p.

[38] Goldman, R. P. “Durative Planning in HTNs”. In: International Conference on Automated Planning and Scheduling, 2006, pp. 382–385.

[39] Gravot, F.; Cambon, S.; Alami, R. “aSyMov: a planner that deals with intricate symbolic and geometric problems”. In: Robotics Research. The Eleventh International Symposium, 2003, pp. 100–110.

[40] Hertle, A.; Dornhege, C.; Keller, T.; Nebel, B. “Planning with semantic attachments: An object-oriented view”. In: Proceedings of the 20th European Conference on Artificial Intelligence, 2012, pp. 402–407.

[41] Hoffmann, J. “The Metric-FF planning system: Translating “ignoring delete lists” to numeric state variables”, *Journal of Artificial Intelligence Research (JAIR)*, 2003, pp. 291–341.

[42] Holleman, C.; Kavraki, L. E. “A framework for using the workspace medial axis in PRM planners”. In: Robotics and Automation, 2000. Proceedings. ICRA’00. IEEE International Conference on, 2000, pp. 1408–1413.

[43] Höller, D.; Bercher, P.; Behnke, G.; Biundo, S. “A generic method to guide HTN progression search with classical heuristics”. In: Twenty-Eighth International Conference on Automated Planning and Scheduling, 2018.

[44] Howey, R.; Long, D. “Validating plans with continuous effects”. In: 22nd Workshop of the UK Planning and Scheduling Special Interest Group, 2003.

[45] Hwang, J. Y.; Kim, J. S.; Lim, S. S.; Park, K. H. “A fast path planning by path graph optimization”, *IEEE Transactions on systems, man, and cybernetics-part a: systems and humans*, vol. 33–1, 2003, pp. 121–129.

[46] Ilghami, O.; Nau, D. S. “A general approach to synthesize problem-specific planners”, Technical Report, DTIC Document, 2003, 9p.

[47] Kautz, H.; Selman, B.; Hoffmann, J. “SatPlan: Planning as satisfiability”, *Booklet of the 2006 International Planning Competition*, 2006.

[48] Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; Overmars, M. H. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”, *IEEE transactions on Robotics and Automation*, vol. 12–4, 1996, pp. 566–580.

[49] Kelly, J.-P.; Botea, A.; Koenig, S.; et al.. “Planning with hierarchical task networks in video games”. In: Proceedings of the ICAPS-07 Workshop on Planning in Games, 2007, pp. 22–26.

[50] Lallement, R.; De Silva, L.; Alami, R. “HATP: An HTN planner for robotics”. In: Proceedings of the ICAPS-14 Workshop on Planning and Robotics, 2014, pp. 20—-27.

[51] Latombe, J.-C. “Robot Motion Planning”. Kluwer Academic Publishers, 1990, vol. 25, 651p.

[52] LaValle, S. M. “Rapidly-exploring random trees: A new tool for path planning”, 1998.

[53] Lekavỳ, M.; Návrat, P. “Expressivity of STRIPS-like and HTN-like planning”. In: KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications, 2007, pp. 121–130.

[54] Magnaguagno, M. C.; Meneguzzi, F. “Method composition through operator pattern identification”, *Workshop on Knowledge Engineering for Planning and Scheduling, KEPS*, 2017, pp. 54.

[55] Magnaguagno, M. C.; Meneguzzi, F. “Method composition through operator pattern identification”. In: ICAPS Workshop on Knowledge Engineering for Planning and Scheduling, 2017, pp. 54–61.

[56] Magnaguagno, M. C.; Meneguzzi, F. “HTN Planning with Semantic Attachments”. In: Workshop on Hierarchical Planning, 2019.

[57] Magnaguagno, M. C.; Meneguzzi, F. “Semantic Attachments for HTN Planning”. In: Proceedings of the Third-Fourth international joint conference on Artificial Intelligence, 2020.

[58] Magnaguagno, M. C.; Pereira, R. F.; Meneguzzi, F. “Dovetail – an abstraction for classical planning using a visual metaphor”. In: The Twenty-Ninth International Flairs Conference, 2016.

[59] Magnaguagno, M. C.; Pereira, R. F.; Móre, M. D.; Meneguzzi, F. “Web planner: A tool to develop classical planning domains and visualize heuristic state-space search”. In: Proceedings of the Workshop on User Interfaces and Scheduling and Planning, UISP, 2017, pp. 32–38.

[60] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; Wilkins, D. “PDDL – the planning domain definition language”, Technical Report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998, 26p.

[61] Meneguzzi, F.; Magnaguagno, M. C.; Singh, M. P.; Telang, P. R.; Yorke-Smith, N. “GoCo: Planning expressive commitment protocols”, *Autonomous Agents and Multi-Agent Systems*, vol. 32–4, 2018, pp. 459–502.

[62] Moura, A. L. D.; Ierusalimschy, R. “Revisiting coroutines”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31–2, 2009, pp. 6.

[63] 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.

[64] Nau, D.; Munoz-Avila, H.; Cao, Y.; Lotem, A.; Mitchell, S. “Total-order planning with partially ordered subtasks”. In: IJCAI, 2001, pp. 425–430.

[65] Nebel, B. “On the compilability and expressive power of propositional planning formalisms”, *Journal of Artificial Intelligence Research (JAIR)*, vol. 12, 2000, pp. 271–315.

[66] Ontanón, S. “The combinatorial multi-armed bandit problem and its application to real-time strategy games”. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference, 2013, pp. 58–64.

[67] Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; Mercorio, F. "Heuristic Planning for PDDL+ Domains". In: AAAI Workshop: Planning for Hybrid Systems, 2016.

[68] Ponsen, M. J.; Muñoz-Avila, H.; Spronck, P.; Aha, D. W. "Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning". In: Proceedings Of The National Conference On Artificial Intelligence, 2005, pp. 1535.

[69] Russell, S.; Norvig, P. "Artificial Intelligence: A Modern Approach", *Artificial Intelligence. Prentice-Hall, Englewood Cliffs*, vol. 25, 1995.

[70] Scala, E.; Haslum, P.; Thiébaux, S.; Ramirez, M. "Interval-based relaxation for general numeric planning." In: ECAI, 2016, pp. 655–663.

[71] Schüller, P.; Patoglu, V.; Erdem, E. "Levels of integration between low-level reasoning and task planning". In: AAAI Workshop on Intelligent Robotic Systems, 2013, pp. 73–78.

[72] Schwartz, J. T.; Sharir, M. "On the "piano movers" problem. II. general techniques for computing topological properties of real algebraic manifolds", *Advances in applied Mathematics*, vol. 4–3, 1983, pp. 298–351.

[73] Siméon, T.; Laumond, J.-P.; Nissoux, C. "Visibility-based probabilistic roadmaps for motion planning", *Advanced Robotics*, vol. 14–6, 2000, pp. 477–493.

[74] Smith, D. E.; Weld, D. S. "Temporal planning with mutual exclusion reasoning". In: IJCAI, 1999, pp. 326–337.

[75] Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; Abbeel, P. "Combined task and motion planning through an extensible planner-independent interface layer". In: Robotics and Automation (ICRA), 2014 IEEE International Conference on, 2014, pp. 639–646.

[76] Srivastava, S.; Riano, L.; Russell, S.; Abbeel, P. "Using classical planners for tasks with continuous operators in robotics". In: International Conference on Automated Planning and Scheduling, 2013, pp. 85–91.

[77] Şucan, I. A.; Kavraki, L. E. "Mobile manipulation: Encoding motion planning options using task motion multigraphs". In: Robotics and Automation (ICRA), 2011 IEEE International Conference on, 2011, pp. 5492–5498.

[78] Şucan, I. A.; Kavraki, L. E. "Accounting for uncertainty in simultaneous task and motion planning using task motion multigraphs". In: Robotics and Automation (ICRA), 2012 IEEE International Conference on, 2012, pp. 4822–4828.

[79] Weyhrauch, R. W. "Prolegomena to a theory of mechanized formal reasoning". In: *Readings in Artificial Intelligence*, Elsevier, 1981, pp. 173–191.

[80] Wooldridge, M.; Jennings, N. R. "Intelligent agents: Theory and practice", *The knowledge engineering review*, vol. 10–2, 1995, pp. 115–152.