Automated planning in hybrid domains for in-space robot assembly tasks

Mariana Dias Cunha

Thesis to obtain the Master of Science Degree in

Electrotechnical and Computer Engineering

Supervisor: Prof. Rodrigo Martins de Matos Ventura

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Rodrigo Martins de Matos Ventura
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

December 2021

Declaration

Acknowledgments

I want to acknowledge all of those whose help and support were essential for the completion of this work. I would like to start by expressing my gratitude towards my supervisor, Professor Rodrigo Martins de Matos Ventura, for his continuous guidance, wise advice and expertise. You were previously my teacher and your work inspired me to pursue this dissertation. Your insightful feedback was crucial throughout the development of this thesis and encouraged me to improve my work to a higher level.

I want to extend my deepest gratitude to PhD Antony Thomas, from the University of Genoa, whose work had a significant impact on this dissertation. You were very helpful in sharing your expertise, I could not have done this without your wisdom and your kindness, I am very grateful to you and I wish you all the success and prosperity in your academic and professional future.

During the development of this dissertation I was also incredibly fortunate to be working at Alfredo AI - Real Estate Analytics. I am very grateful to all of them, highlighting Guilherme Farinha, Mário Gamas and Gonçalo Abreu, for their advice, support and understanding, for being mentors in sharing their knowledge and experience, for being available when I needed help and for making me feel valued and supported throughout the development of my dissertation.

I could not go without expressing how grateful I am for my family, especially my siblings and my parents, for their unconditional love and support. I would never have gotten here without you and I feel very lucky that I have such wise and caring parents who provided me the most amazing opportunities and made me into the person I am today. You helped me the most through these last years whenever things seemed overwhelming, your guidance and support was crucial to me and I will be forever grateful.

I also want to thank my boyfriend Álvaro for his continuous emotional support, friendship and encouragement, for always being there for me and pushing me forward. You helped me grow as a person and were a great source of support and inspiration to me. Thank you.

I was very fortunate to meet some amazing people in school and during my years at Instituto Superior Técnico, whose friendship I cherish. For all the friends who were present through this university journey, which included some of the most challenging moments of my life, thank you for making these last years seem a little easier and full of great memories. For those who were in IST with me, we succeeded together and I am grateful for our mutual support and friendship.

Abstract

Recent developments in the area of automation of in-space additive manufacture and assembly of structures has resulted in the development of in-space mobile robots that perform assembly and logistics operations inside the International Space Station (ISS). Consequently, studies and improvements in the area of robot Task-Motion Planning (TMP) need to be made in order to improve robot navigation. Inspired by this, we have created a hybrid domain using PDDL that describes mobile robots navigating through a 2D approximation sampled roadmap that are responsible for moving, loading and assembling modules.

We have chosen a state of the art TMP approach called MPTP. However, in this approach, the motion planner is not aware of the world's configuration by the time it is invoked by the task planner, for we have contributed with an added logic to the approach that allows the motion planner to be informed of the map availability according to the actions expanded by the task planner.

We designed a scenario as a proof-of-concept to test and compare the use of task and motion planners separately versus a mixed TMP approach. Our results illustrate that using a mixed approach in domains such as the space assembly domain presents advantages when compared to a separate approach since it is aware of physical motion constraints while task planning.

Keywords

Hybrid domain; PDDL; Mobile robot navigation; Task planning; Motion planning.

Resumo

Avanços recentes na área da automação de impressão e montagem de peças 3D no espaço resultou no desenvolvimento de robôs móveis com capacidade de movimentar e montar peças, bem como executar tarefas de logística no interior da Estação Espacial Internacional. Consequentemente, é necessário realizar estudos e melhorias na área do planeamento de tarefas e movimento de robôs para melhorar a sua navegação. Posto isto, criámos um domínio híbrido em PDDL que descreve a navegação de robôs móveis numa aproximação 2D de um mapa de pontos amostrados responsáveis por mover, carregar e montar módulos.

Escolhemos uma abordagem recente de planeamento de tarefas e movimento chamada MPTP. No entanto, nesta abordagem o planeador de movimento não tem em conta a configuração do mundo no momento em que é invocado, pelo que contribuímos com a adição de uma nova lógica à abordagem que permite que o planeador de movimento receba informação acerca das zonas do mapa disponíveis consoante as ações expandidas pelo planeador de tarefas.

Definimos um cenário como prova de conceito para comparar o planeamento de tarefas e movimento em separado com o planeamento feito em conjunto. Os resultados obtidos ilustram que a utilização de uma abordagem conjunta, em domínios como o de montagem no espaço, apresenta vantagens relativamente a uma abordagem desacoplada, uma vez que o planeador de tarefas está ciente de restrições de movimento impostas pelas ações que vai expandindo, conseguindo assim um plano de melhor qualidade.

Palavras Chave

Domínio híbrido; PDDL; Navegação de robôs móveis; Planeamento de tarefas; Planeamento de movimento.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

BFS	Best First Search
CNF	Conjunctive Normal Form
EHC	Enforced Hill Climbing
IDTMP	Iteratively Deepened Task and Motion Planning
ISS	International Space Station
LP	Linear Programming
MPTP	Motion-Planning-aware Task Planning
OMT	Optimization Modulo Theories
PDDL	Planning Domain Definition Language
POPF	Forward-Chaining Partial-Order Planning
POPF-TIF	Forward-Chaining Partial-Order Planning with Timed Initial Fluents
PRM	Probabilistic Roadmap
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
TIF	Timed Initial Fluents
TIL	Timed Initial Literals
TMP	Task-Motion Planning

1 Introduction

Contents

In this chapter we will provide the motivation that led to the development of this dissertation, as well as a description of the proposed objectives. An overview of the main contributions of this thesis is also provided and lastly an outline of the structure of this document.

1.1 Motivation

There has been a collaboration between ISR-Lisboa and MIT Space Systems Laboratory in the context of autonomous robotic assembly of space structures using on-orbit additive manufacturing. By enabling the autonomous robotic assembly of space structures, near-Earth science can be improved which is the main goal of this international partnership.

Since 2014 there is an on board 3D printer operating in the International Space Station (ISS) [1] which enables the production of 3D parts and tools that can be used for repairing or improving existing hardware. Figure 1.1 shows the 3D printer on board of the ISS. However this technology of 3D printing architectures has the potential of being extended to fabricate bigger parts of space structures that can then be autonomously assembled in space, making way for the ISS to be able to repair and re-provision and thus being able to maintain and upgrade its own structure. By means of path planning and assembly of 3D printed parts this would make way for exploration and scientific operations of space structures.

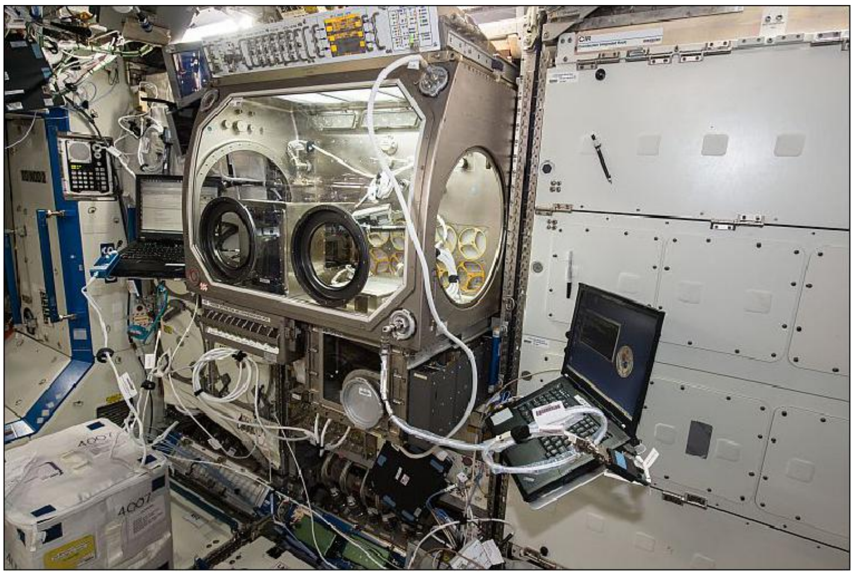


Figure 1.1: In space 3D printer aboard the ISS[1]

By analysing the ISR-Lisboa and MIT Space Systems Lab collaboration renovation proposal report for 2020 we can see that the proposed goals are as follow:

[1] Image source: https://directory.eoportal.org/web/eoportal/satellite-missions/i/iss-3d-print

- Understand how additive manufacturing of components for on-orbit assembly can be translated into assembly via high-level path and task planning;

- Address the problem of motion control while manipulating objects during assembly tasks under uncertainty of inertial parameters, including problems in transportation and physical contact of the assembler and part. Optimize trajectories to meet requirements for grasping and assembly including constraints in dynamics, collision avoidance, and contact;

- Perform experimental validation of the developed methods using the NASA Astrobee free-flyer robots on the ISS. It is envisioned that one or more Astrobee will be able to grasp (using its two degree of freedom arm) and transport small objects simulating manufacturing and assembly scenarios.

This dissertation focuses on the first described objective which concerns the autonomous assembly of parts via high-level path and task planning. The work in [2] shows a thorough study of different architectures involving in space printing and assembly of parts. Figure 1.2 shows one example of the architectures studied in [2] where a mobile robot (represented by the circle with number 5) that is operating in an environment where a 3D printer (block 4) printed three parts (represented by the modules labeled from 1 - 3) being the first picture square the initial state and the picture square 6 being the goal state. In this example the robot uses proximity operations to assemble the 3D printed modules next to the printer in reverse order.

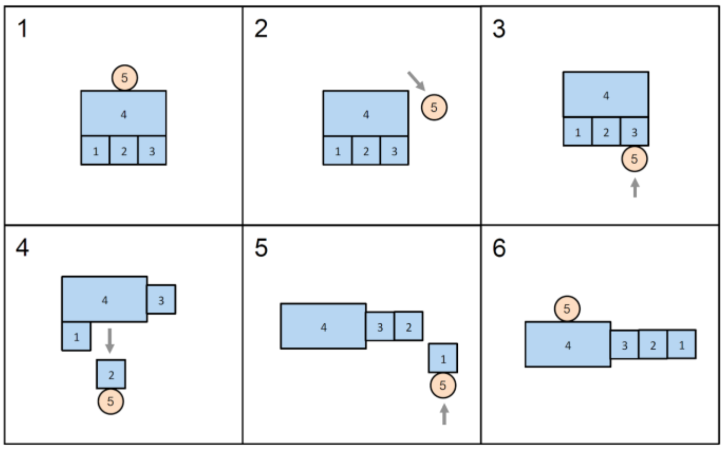


Figure 1.2: Manufacturing and assembly architecture consisting of one mobile robot (represented by the orange circle with label 5), a 3D printer (labeled as module 4) and three printed parts (labeled as modules 1 - 3) where the goal is for the robot to place the modules next to the printer[2]

Figure 1.2 represents one possible task plan for the robot to assemble the modules into the desired goal configuration. This may be referred to as hybrid planning since it involves discrete and continuous

[2]Image taken from [2]

aspects and it is integrated in the field of task-motion planning. This is an area that has been evolving and different approaches have been emerging: some where task and motion planning are done in a separate and sometimes interspersed way and others where this is done in a more mixed way, where task planning has a form of integrating motion planning. Some robots have been created in the context of robotic space assembly. Robots like the free-flyer Astrobee [3] which was made for investigation purposes can be used to autonomously carry out assembly tasks and logistic operations inside the ISS. Figure 1.3 shows Astrobee inside the ISS.

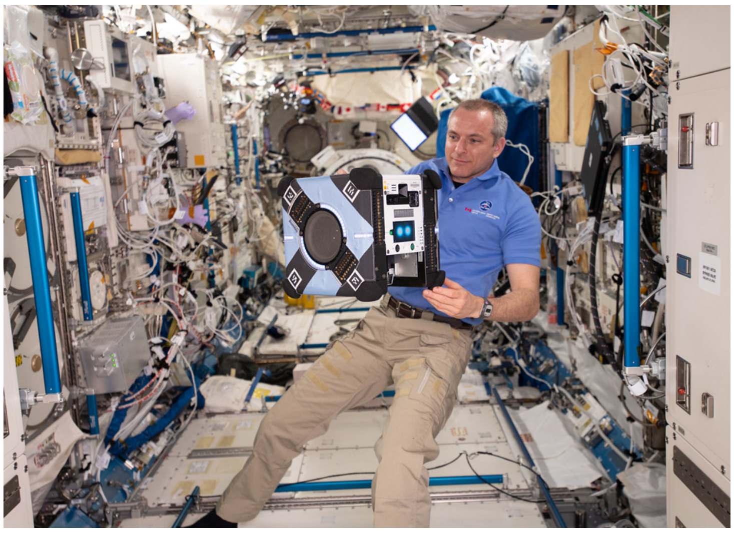


Figure 1.3: Astrobee operating inside the ISS[3]

We want to contribute to the study of these different task-motion planning approaches in the context of the space assembly of 3D printed parts in order to know what would be more appropriate and produce better results in terms of solution task-motion plans for environments such as the one described in figure 1.2. For this reason, this thesis arises from the motivation of improving the area of autonomy of robotic assemblers for assembly of space structures. More specifically, this dissertation was motivated by the need to advance the area of task-motion planning in the context of the autonomous logistic operations inside a space station done by robots.

1.2 Objectives

This thesis is focused on studying which type of task-motion planning approach is better suited for a scenario of a robotic autonomous assembly of modules through experimental results in a simulated environment. In order to reach this objective we define the following goals:

[3]Image source: http://spaceref.com/international-space-station/testing-the-astrobee-robotic-assistant-droid.html

1. Definition of a space assembly domain into PDDL where free-flyer robots like Astrobee can move and assemble parts inside an environment like the ISS; this includes defining all the actions with their respective preconditions and effects, along with the propositions needed to accurately represent and simulate the domain;

2. Definition of problems into PDDL within the context of the space assembly domain;

3. Choosing a state of the art task-motion planning approach to run tests using the space assembly domain;

4. Development of a new logic to implement on the MPTP approach in the sense of making the motion planner aware of the world's configuration at the time it is invoked by the task planner;

5. Comparison between using task and motion planning separately and using a task-motion planning mixed approach in the context of the space assembly domain.

Our goal is consequently to help improve existing methods for automated task planning in hybrid domains by proving which type of planning approach is better suited for domains such as the space assembly domain where the tasks to carry out include assembly of structures and logistics operations inside a space station. We hope that this work is a contribution to the area of autonomy of robotic assemblers.

1.3 Contributions

The main contributions of this work include:

- Development and implementation of an approach that allows the motion planner to be informed of the world's configuration and map availability according to the actions expanded by the task planner. This, as the main contribution of this work, will allow an improvement on the previously existing state-of-the-art task-motion planning approach, MPTP, by making the motion planner have an updated view on the world's configuration according to the changes imposed by the task planner's expanded actions;

- Development of a PDDL domain that represents in space assembly of modules by robots like Astrobee aboard the International Space Station;

- Experimental results on the advantages of using a mixed task-motion planning approach in certain scenarios like the space assembly domain;

- Development of a python script that allows the user to visually define a problem in the space assembly domain and the PDDL file is automatically generated without the user having to understand PDDL.

1.4 Outline

This thesis is organized as follows:

- In chapter 2 there is an introduction to the background concepts that support this dissertation, followed by the state of the art approaches relevant to this work and it ends by explaining the approach that will serve as baseline for this thesis;
- Chapter 3 is divided into two sections: the first one refers to the methodology and contains a detailed explanation of the developed domain and approach used under this dissertation; the second part refers to the implementation, where there is a thorough description of the way the approach was implemented in terms of the developed code;
- Chapter 4 shows the tests that were made as well as the obtained results. It contains three sections that refer to three different goals to be tested and each section has the conclusions reached by the analysis of the obtained results.
- Chapter 5 is the culmination of all this work's conclusions.

All the bibliographical references that are cited in this dissertation can be found in the Bibliography section. There is also an appendix at the end, appendix A, which contains the PDDL files for the space assembly domain and for the Columbus lab scenario problem that is described in the results section.

2 Background

Contents

We start by giving a brief introduction to the background concepts of this thesis, as well as a description of the current existing approaches and then explaining the state of the art planning approach that serves as baseline for this work.

2.1 Background Concepts

The background concepts explained in this section have been studied from different sources, including mainly the textbooks [4] and [5].

2.1.1 Propositional Logic

Propositional logic is a simple but widely used type of logic. A **proposition** is a declarative sentence (or statement) that can be either true or false and is represented by a **proposition symbol** which is a letter (commonly a capital letter). Propositional logic does not look into a statement's content but instead focuses on how statements interact with each other in terms of their logical form.

A **connective** can logically connect propositions to form more complex sentences. The main connectives are:

- **not** ($\neg$): negates a statement;
- **and** ($\wedge$): the conjunction of 2 propositions;
- **or** ($\vee$): the disjunction of 2 propositions;
- **implies** ($\Rightarrow$): for example, if $P \Rightarrow Q$ it means that if P is true, Q is also true;
- **if and only if** ($\Leftrightarrow$): for example, if $P \Leftrightarrow Q$ it means that P is true if and only if Q is true.

The full meaning of the connectives is illustrated by their respective truth tables (all represented together on Table 2.1) which will be further explained in the next paragraph.

Table 2.1: Truth tables for each connective ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\neg$) according to the truth value of p and q (T: *true*; F: *false*; p and q are different propositions)

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$	$\neg p$
T	T	T	T	T	T	F
T	F	F	T	F	F	F
F	T	F	T	T	F	T
F	F	F	F	T	T	T

Going further into the semantics of propositional logic, to determine the truth value of connected statements, there are **truth tables**. Truth tables define the truth value of connected statements for each and every possible combination of each statement's truth value. Knowing the truth table of each connective allows us to know the truth value of connected statements. Table 2.1 shows the truth tables for the most commonly used connectives defined above.

There are also some properties of the connectives that are important to know to help solving more complex propositional logic problems. These properties can be seen on Table 2.2.

Table 2.2: Properties of propositional logic connectives

Logical property	**Property's name**
$(\alpha \land \beta) = (\beta \land \alpha)$	commutativity of $\land$
$(\alpha \lor \beta) = (\beta \lor \alpha)$	commutativity of $\lor$
$((\alpha \land \beta) \land \gamma) = (\alpha \land (\beta \land \gamma))$	associativity of $\land$
$((\alpha \lor \beta) \lor \gamma) = (\alpha \lor (\beta \lor \gamma))$	associativity of $\lor$
$\neg(\neg\alpha) = \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) = (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) = (\neg\alpha \lor \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) = ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \land \beta) = (\neg\alpha \lor \neg\beta)$	De Morgan
$\neg(\alpha \lor \beta) = (\neg\alpha \land \neg\beta)$	De Morgan
$(\alpha \land (\beta \lor \gamma)) = ((\alpha \land \beta) \lor (\alpha \land \gamma))$	distributivity of $\land$ over $\lor$
$(\alpha \lor (\beta \land \gamma)) = ((\alpha \lor \beta) \land (\alpha \lor \gamma))$	distributivity of $\lor$ over $\land$

In section 2.1.3 it will be specified one situation where using the properties in Table 2.2 is very helpful to solving a problem.

2.1.2 First-Order Logic

When compared to propositional logic, the main advantage of first-order logic is that it can evaluate the content of the propositions, whereas propositional logic cannot, which means that first-order logic is more expressive. First-order logic contains predicates and quantifiers.

A **predicate** is a statement that contains a variable and it cannot be given a truth value (*true* or *false*) since the variable still has no specified value. An example of a predicate can be "x is a student", which in first-order logic is represented with a shorthand notation, so in this case it would be represented, for example, by $Student(x)$, where $Student()$ denotes the predicate "is a student" and x is the variable.

It is only called a **proposition** when a statement contains a predicate and a subject (which is a specified value for the predicate's variable). So, in the previous example, $Student(x)$ becomes a proposition when a value is assigned to the variable x, for example, the model $x \leftarrow Sarah$ makes the proposition $Student(Sarah)$. This is a proposition since it has a truth value, i.e. the statement that "Sarah is a student" is either true or false.

A **quantifier** expresses the quantity of a variable:

- **universal quantifier** ($\forall$): for example, if we write $\forall x : Student(x)$ it means that the predicate $Student(x)$ is true for all x (translated into a sentence it means ”Every x is a student”);
- **existential quantifier** ($\exists$): for example, if $\exists x : Student(x)$ it means that there exists at least one x for which $Student(x)$ is true (translated into a sentence it means ”Some x is a student”).

Both quantifiers have some relationship properties between them. These properties are represented in Table 2.3 and they are relevant for solving more complex problems.

Table 2.3: Properties of universal and existential quantifiers

Property	Property’s name
$\forall x : \neg P(x) \equiv \neg \exists x : P(x)$	negation
$\neg \forall x : P(x) \equiv \exists x : \neg P(x)$	negation
$\forall x : P(x) \equiv \neg \exists x : \neg P(x)$	De Morgan
$\exists x : P(x) \equiv \neg \forall x : \neg P(x)$	De Morgan

Just like propositional logic, first-order logic also uses the operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\neg$ to connect propositions and form complex sentences. The equality symbol is also used to express that two propositions are the same, for example $Mother(Sarah) = Julia$, meaning that these two terms refer to the same object.

An important concept to mention is **entailment**, which is represented by the symbol $\vDash$ and it means that, for example if $KB \models q$, the knowledge base KB entails the sentence q if and only if every positive literal in q is in KB and every negated literal in q is not.

2.1.3 Boolean Satisfiability Problem

A **literal** refers to a proposition or its negation (for example, p or $\neg p$), respectively denominated a positive literal and a negative literal. The **Conjunctive Normal Form (CNF)** refers to a propositional formula that is a conjunction of disjunctions, meaning that it is a conjunction of clauses and each clause is a disjunction of literals. Any propositional formula that is not in CNF can be converted into it by means of properties like the ones in Table 2.2. The formula

$$(x \vee y) \wedge (\neg y \vee z) \tag{2.1}$$

is a simple example of a propositional formula in CNF.

A **model** is the assignment of the truth value (*true* or *false*) to each propositional symbol in a propositional formula. The goal of the **Boolean Satisfiability Problem (SAT)** is to find a model that makes a formula with Boolean variables true. Most SAT solvers take in formulas that are in CNF. In the case of the propositional formula (2.1), one possible solution for the SAT problem, i.e. a model that would make the sentence true, would be $\{x = true; y = false; z = true\}$, for example. One way of checking if a model makes a sentence true is by looking at the truth tables (in Table 2.1).

If a sentence is true for each and all the different models then the sentence is said to be **valid**. A valid sentence can also be called a **tautology**. A sentence is **satisfiable** if there is at least one model for which the sentence is true (or satisfied). If there is no model at all that can make the sentence true, it is said to be **unsatisfiable**.

The concepts of validity and satisfiability are connected and the relationship between them is as follows:

- α is valid $\equiv \neg\alpha$ is unsatisfiable;
- α is satisfiable $\equiv \neg\alpha$ is not valid.

2.1.4 Satisfiability Modulo Theories

Some real-world applications require a different and more expressive type of logic other than propositional logic or even first-order logic. For example, the dynamics of a robot may contain linear and non-linear arithmetic. The way to solve the satisfiability problem for these different types of formulas is by resorting to **Satisfiability Modulo Theories (SMT)**, which verifies satisfiability with respect to a background theory, since a supporting theory is required to capture the meaning of these formulas.

SMT combines the Boolean satisfiability problem with domains. The SMT problem consists of trying to find out if it is possible to assign real values to the continuous variables that satisfy the formula for quantifier-free first-order logic formulas over different theories, i.e. formulas that contain Boolean combinations of polynomial equations and inequalities over continuous variables (real-valued variables).

Some examples of different theories are array theory, bit-vector and floating-point arithmetic, linear and non-linear arithmetic, etc. The focus of this thesis will be on quantifier-free linear and non-linear real arithmetic.

The overall way an SMT problem solver works is depicted in Figure 2.1. First, an input CNF formula is abstracted into a Boolean formula. If the input formula is, for instance,

$$x \geq 1 \wedge (y \geq 2 \vee x < 1) \wedge x > y, \tag{2.2}$$

then its Boolean abstraction would be

$$p1 \wedge (p2 \vee \neg p1) \wedge p3. \tag{2.3}$$

After the Boolean abstraction is made, the formula is ready to enter the SMT solver. The solver finds a model and the output is the theory constraints that come from that model. The theory constraints then

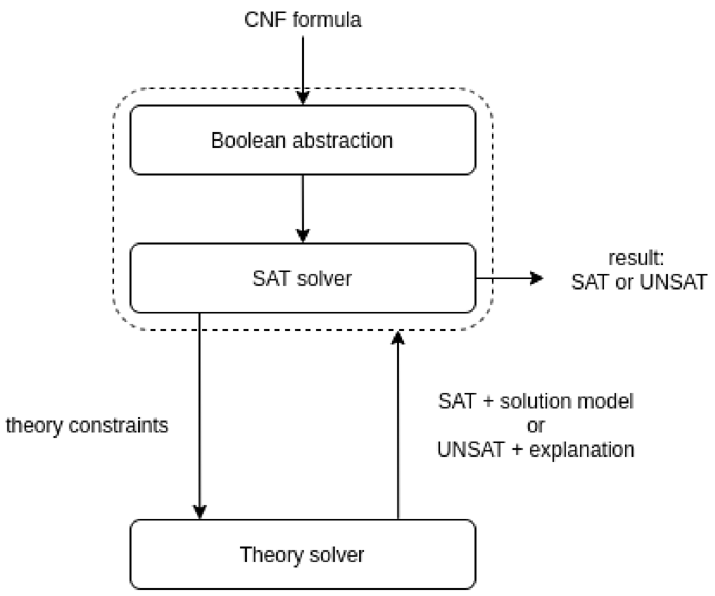


Figure 2.1: Overview of SMT problem solving

enter a theory solver that verifies whether or not these constraints are consistent with the theory. If they are, it means that the model is satisfiable (SAT) and the output of the solver is the model that corresponds to the problem's solution. On the other hand, if the theory constraints are inconsistent with the theory, then the model is unsatisfiable (UNSAT) and the theory solver's output will give an explanation (a theory lemma) that can be, for instance, the negation of the theory constraints that are inconsistent. With this explanation, the Boolean abstraction will be refined, changing to the conjunction of the consistent theory constraints and the theory lemma, and the SAT solver will reattempt to find a different model.

This process is repeated iteratively until a satisfying solution is found or a boundary is met (such as a time limit or all the possible models were checked and the problem is unsatisfiable).

2.2 Classical Planning

Planning refers to finding a model and a sequence of actions that will lead the system from its initial state to a desired state. In **classical planning**, a few assumptions are made: (a) it is assumed that the environment is finite and static, meaning that the state only changes if an action is performed, excluding any type of effects from different agents or exogenous events; (b) it is also assumed that there is no concurrency of events and time is not represented, assuming only a discrete set and sequence of actions and states; (c) there is also the determinism assumption, which means that it is assumed that there is no uncertainty when predicting the resulting state of an action in a given state, excluding any type of execution errors and non-deterministic actions.

To sum it up, classical planning is a very simple way of planning that assumes a single agent acting in a fully observable world with deterministic actions. The aim of classical planning is to find a sequence of actions that leads from an initial state to a desired goal state. This sequence of actions is called a **plan**.

Finding any plan that satisfies the goal is known as **satisficing** planning. There is also another type of planning that focuses, not only on finding a plan that is a solution to the problem, but also on finding the plan with the lowest cost, i.e. the optimal plan. This type of planning is referred to as **optimal** planning, however this is not the main focus of this work.

2.2.1 PDDL

McDermott first released the standard version of the Planning Domain Definition Language (PDDL) in 1998 [6, 7]. As the name states, it is a language used to define a classical planning problem domain. This means that PDDL is a language capable of representing the four main aspects of the classical planning domain: the initial state, the actions that can be executed in each state, the effects of each of those actions and the goal state test.

In PDDL, a state is represented by conjunctions of fluents, for instance, $Happy \wedge Student$ if we want to represent the state of a person , or $At(robot_1, area_1) \wedge At(robot_2, area_2)$ for describing a state in which multiple robots are in different areas. Also, something like $At(robot_1, area_1)$ is defined in PDDL as a **predicate**, which is a relation between one or more objects. In this case, $robot_1$ and $area_1$ are instantiated **objects** for example of the object types $robot$ and $area$, respectively.

An action is represented by an **action schema**, which specifies the name of the action and defines what are the **preconditions** necessary for the action to be able to be executed, as well as the **effects** that it produces, i.e. what parts of the state change with the execution of that action. Both the preconditions and effects are defined as a conjunction of literals. An example of an action schema for the movement of a robot from one area to a different area is as follows:

$Action(Move(r, from, to),$

$PRECOND : At(r, from) \wedge Robot(r) \wedge Area(from) \wedge Area(to)$

$EFFECT : \neg At(r, from) \wedge At(r, to))$

As can be seen from the definition of the action schema's preconditions and the concept of entailment, an action a can be executed in a state s if the state entails the action's preconditions, which can be represented by $(a \in ACTIONS(s)) \equiv s \models PRECOND(a)$.

If the preconditions of an action a are satisfied by the state s, then a is **applicable** in s. An action schema may contain variables that can be assigned values. An action can have multiple models that make it applicable in a state s, making it have multiple applicable instantiations in s.

When applying an action in a state s, the resulting state s' is defined by taking the initial state s and removing all the negative literals in the action schema effects and adding all the positive ones. An effect always comes sequentially after the action's preconditions, we can translate this as if the preconditions occurred at time t and the effects at time $t + 1$.

In order to define a planning problem using PDDL we need to provide a domain description and a problem description. For the domain description we simply need to define all necessary action schemas and all the predicates with their respective free variables. The problem description has to specify the objects that will instantiate the predicates and the action schemas by replacing all the free variables, and it also has to specify an initial state and a goal condition.

The **initial state** is defined as a conjunction of literals that are assumed to be true (no need to represent the ones that are false due to the closed world assumption). Just like the preconditions and effects, the **goal** is defined as a conjunction of literals that may contain variables and it describes the result state we want to achieve.

Several versions of PDDL have emerged through the years, such as PDDL2.1 and PDDL+, which will be mentioned in a later section of this chapter.

2.2.2 Planning as SAT

As Kautz and Selman referred in their paper in 1992 [8], it is possible to model a planning problem as a satisfiability problem. The core idea is to encode the planning problem as a satisfiability problem and then use a SAT solver on the encoding to find a solution. Later works improved Kautz and Selman's satisfiability encoding of planning problems, mainly by adding parallel encodings and substituting the classical frame axioms by explanatory frame axioms.

For an action to be able to be executed, all its preconditions must hold at time t and all its effects must hold at the time step $t + 1$. So to convert an action from a planning problem into a satisfiability problem is to define that if an action is true then it implies that all of those action's preconditions are also true at the same timestamp and it also implies that all the action's effects are true in the next timestamp (or state).

It is also determined that only one action is allowed to occur at the same time and that it is mandatory to have one action being executed at every time step. According to [8], as long as a complete initial state is defined on the planning problem, all models are guaranteed to be valid plans due to these previously defined axioms.

2.3 Hybrid Systems

Simply put, a hybrid system is one that contains both continuous and discrete state variables. When talking about a system with multiple states, discrete variables are the ones associated with discontinuous transitions between different states and continuous variables represent continuous evolution within a state. In the context of this thesis, since we are dealing with mobile robots, there is the need for a hybrid plan since we need to know, not only the task plan, but also a motion plan for the robot.

It is possible to formally describe a hybrid model through **Hybrid Automata**, which allows for the verification of a hybrid model. The following hybrid automaton definition was inspired by and adapted from [5]. A hybrid automaton is modeled by equation (2.4),

$$H = (X, G, Init(v_i), Inv(v_i), Flow(v_i), Jump(e_{ij})), \tag{2.4}$$

where $X = \{x_1, ..., x_n\}$ is a finite set of continuous variables with real values over time; $G = (V, E)$ is called the control graph, where $v_i \in V$ represents a control mode and $e_{ij} = (v_i, v_j) \in E$ is a control switch: the control graph refers to the discrete part of the automaton and there is a finite set of control modes V; $Init(v_i)$, $Inv(v_i)$ and $Jump(e_{ej})$ are all conditions over X; and $Flow(v_i)$ is a condition over X; and $\dot{X}$ (which is the first derivative of X with regards to time): the $Init(v_i)$ condition is the set of initial values of X for each control mode v_i; $Inv(v_i)$ is an invariant for each control mode v_i; $Flow(v_i)$ describes the continuous change of the X variables for each control mode and $Jump(e_{ej})$ represents the discrete change and it is a condition that triggers a control switch e_{ij} and when satisfied makes the system jump from state v_i to v_j.

To better illustrate the concept of hybrid automata let us analyse the example on figure 2.2 (inspired by the thermostat example from [5]): we have a system that consists of a houseplant in a vase with a sensor that measures the volume of water in liters (L) contained in the soil and an irrigation system that can be turned off or on according to the measured water volume.

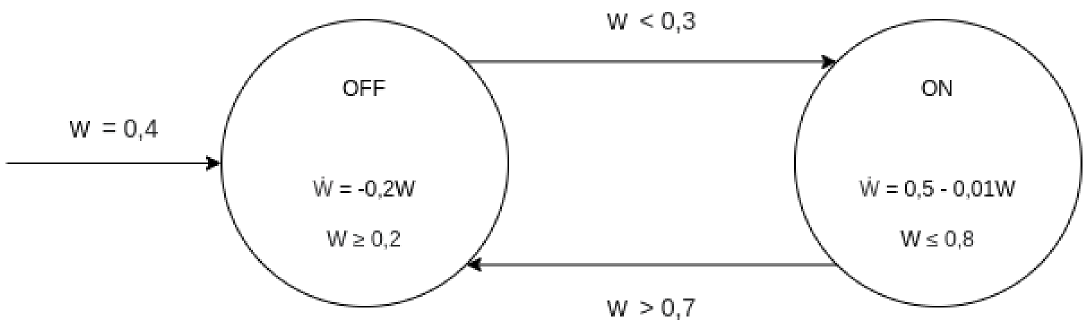


Figure 2.2: Example of a hybrid automaton for an irrigation system in a houseplant (W is the variable that represents the water volume in L)

For the hybrid automaton H in this example, $X = \{W\}$ since there is one continuous variable W which is the volume of water in the soil measured by the sensor. Then we have the irrigation that can be in one of two control modes: off and on, so the set of control modes is defined by $V = \{on, off\}$, and consequently there are two control switches $E = \{(on, off), (off, on)\}$ and the control graph is $G = \{V, E\}$. It is assumed that the irrigation system is initially off, which means that the initial condition is $Init(off) \triangleq W = 0,4$ and as we can see inside the *off* state in figure 2.2, the irrigation system will remain turned off while the water volume present in the soil is bigger than or equal to $0,2L$, which means that $Inv(off) \triangleq W \geq 0,2$. We can also see inside the *off* state in figure 2.2 the equation that describes how the water volume drops over time, i.e. the flow of the continuous variable W, so we have the flow condition $Flow(off) \triangleq \dot{W} = -0,2W$. As soon as the water volume drops to less than 0,3L, the irrigation system may turn on, which translates to the jump condition $Jump((off,on)) \triangleq W < 0,3$ and the latest that the irrigation system will turn on is when $W = 0,2L$ since $Inv(off) \triangleq W \geq 0,2$.

The same reasoning applies to the other side of figure 2.2: the irrigation system will remain in the control mode *on* while the water volume on the vase is below 0,8L, so $Inv(on) \triangleq W \leq 0,8$ and while it is turned on, the volume of water on the soil increases following the equation $Flow(on) \triangleq \dot{W} = 0,5 - 0,01W$. The irrigation system may turn off as soon as the water volume rises above 0,7L: $Jump((on,off)) \triangleq W > 0,7$ (and the latest that it will turn off is when the water volume is 0,8L, due to $Inv(on) \triangleq W \leq 0,8$).

The described jump conditions represent a non-deterministic jump since there is an interval of values for the continuous variable W where the jump can happen instead of a single fixed value as it would happen in the case of a deterministic switch. As described in the previous paragraph, the actual switch condition is given by the combination of both the jump condition and the invariant of the control mode ($Jump + Inv$). Together these two conditions define the interval of values of W for which the irrigation system may switch for each control mode.

Adding the concept of input and output variables to hybrid automata enables the planning and acting with hybrid models. Input and output variables contain both the set of continuous and discrete variables and the input variables can be divided into controllable and uncontrollable. The set of continuous variables is X and the set of discrete variables can be represented by $Y = \{y_1, ..., y_n\}$ and each y_i is a discrete variable associated with a control mode v_i, similarly to a state representation. This way, an actor can affect a hybrid model by defining the controllable continuous and discrete input variables and it can then perceive the system's status by analysing the output continuous and discrete variables. Uncontrollable input variables are modeled by different dynamics, i.e. they may change according to the environment, exogenous events, etc. Planning and acting with hybrid models is a very challenging problem, much more complex than planning with classic discrete models.

2.3.1 PDDL2.1 and PDDL+

As mentioned in section 2.2.1, there is a version of PDDL called PDDL2.1 [9]. This PDDL version can handle temporal considerations (scheduling) and numeric considerations (resources) and introduced, for that purpose, numeric fluents for continuous change, plan-metrics and durative/continuous actions.

A **durative action**, besides having preconditions and effects, also has a duration and a way to assign continuous time to each precondition and effect by specifying that each one occurs in the beginning of the action, by the end of the action or throughout the action, which corresponds to the PDDL construct *at start*, *at end*, and *over all*, respectively.

A **numeric fluent** is similar to a predicate and is declared in the PDDL domain file as a function. A numeric fluent is a variable that has a value throughout the plan and applies to zero or more objects from the PDDL domain. Both actions and durative actions' effects can change the value of a numeric fluent. It is declared with a name, an object name and object type.

Another version of PDDL is PDDL+ [10] which is an extension of PDDL2.1 and it was created to enable the modelling of hybrid domains. It also provides a more flexible model of continuous change through the use of processes and events and supports modelling of exogenous events.

While an action changes the state, and a durative action is an action that lasts for a certain period of time (it has a defined duration), a **process** is a durative action that holds for as long as its preconditions are true and it can provoke a continuous effect on a state. An **event** is something that comes from the outside environment and triggers a change of state, in other words it is an instantaneous action that is not controllable and that makes the state change, negating its preconditions.

2.3.2 Hybrid System Based Planners

There are a lot of different hybrid system based planning approaches that emerged over the years and we will briefly describe four of them in this section in chronological order of when they appeared. The first approach that we will mention is called UPMurphi [11] (2009) and it performs universal planning using a model checking based algorithm for hybrid and nonlinear systems. It is also able to read problem specifications from PDDL+ files and plan for problems with time and resources constraints.

Another hybrid planning approach is DReach [12] (2015) which also uses PDDL+. This approach is able to accommodate nonlinear change by encoding problems as nonlinear hybrid systems and then applying Satisfiability Modulo Theories (SMT) (to know more about SMT in the context of hybrid system planning we refer the reader to the work in [13] and [14]). This planner finds plan tubes instead of concrete plans because it solves a δ relaxation of the problem. This also presents heuristics for improving SMT variable selection and pruning. This approach can prove plan non-existence.

A different approach that also uses an SMT encoding of PDDL+ domains is SMTPlan+ [15] (2016). SMTPlan+ is also able to deal with nonlinear arithmetic and it can use any SMT solver. This planner is also efficient in proving plan non-existence up to a certain bound and has proven to outperform UPMurphi and DReach in terms of time and number of instances it is able to solve using different types of domains.

A fourth and more recent hybrid system based planning approach is Optimization Modulo Theories (OMT) [16] (2018). It is a task planner that is capable of finding optimal solutions and working in a multi-robot system with concurrent actions. OMT consists of SMT solving with optimization capabilities. This approach has been used and tested in the RCLL competition where results showed that the solving time increases with the increase in the number of robots, however the times were within the desirable bounds according to the competition's rules. Results also show that OMT outperforms SMTPlan+ and produces solutions with smaller average makespans compared to other approaches.

2.3.3 POPF-TIF

Forward-Chaining Partial-Order Planning (POPF) [17] is a forward-chaining state-based search planner designed to solve temporal-numeric problems supporting the concept of partial order planning. This planner handles continuous linear numeric change and is based on grounded forward search combined with Linear Programming (LP).

This planner does not sequentially build a plan, instead it builds partially ordered collections of actions. The partial ordering is achieved by delaying the commitment to ordering the decisions, timestamps and values of the numeric parameters by managing sets of constraints as actions start and end, meaning that the precise embedding of actions in time is delayed until constraints emerge. This approach benefits from the informative search control of forward planning and at the same time it has some level of flexibility due to its late commitment strategy. The partial ordering is done in a way that ensures the consistency of the plan.

Since it is a temporal planner, POPF uses PDDL2.1 which supports durative-actions. However, this is a versatile approach that can be used both for temporal and non-temporal planning and also for both purely propositional representations or instantaneous and linear continuous numeric change. That being said, POPF was created with the main goal of solving temporal-metric planning problems. In a problem like this, a state is characterised by:

- the set of propositions that are true in that respective state;
- the set of values of the numeric variables;
- an event queue, which contains the actions that have started but still haven not ended their execution, as well as the respective step at which the action started;
- a list of all the temporal constraints that hold for the set of steps.

A state is updated whenever an action is added to the plan, since the action's effects will update the set of propositions that hold, as well as the set of values of the task numeric variables. An action can only be applied at a certain time step as long as its effects do not interfere with the invariants of the actions from the event queue from that current state. Since this is a temporal planning problem, the set of temporal constraints is updated every time a step is added to the plan.

To clarify, a plan can be seen as a sequence of steps. A durative action is an action that holds for a certain amount of steps and its effects can be seen as instantaneous effects that take place at certain time instants: the first step is when the action starts and it is where the $at\ start$ effects become true, the last step is where the $at\ end$ effects of the action take place and all the steps in between including the starting and last steps hold all the $over\ all$ effects of the action. Any action can take place at any step as long as its preconditions do not cause conflict with the effects of the actions that are still being executed. That being said, each state records which steps in the plan interact with a given fact p or numeric variable v. According to [18] POPF defines each state by keeping a record of each of the following aspects:

- $F^{+}(p)$ records the step of the plan that most recently introduced an add effect on the fact p (and $F^{-}(p)$ for a delete effect);
- $FP(p)$ is the set of preconditions that involve p, as well as their respective step index i. Each set element is of the form $< i, d >$, where d is 0 (in case p can be deleted in parallel to step i) or ϵ (in case p can only be deleted ϵ steps after step i);
- $V^{eff}(v)$ is the most recent step that introduced a numeric effect on the variable v;
- $VP(v)$ is the set of all the steps that depend on the variable v - this includes steps that have either preconditions, effects or duration depending on the current value of v.

As it is looking for a plan, the planner then adds the necessary ordering constraints to make sure that the propositional preconditions, numeric preconditions, and duration constraints are always satisfied. An important aspect of POPF is that it imposes total ordering in steps that change the value of a variable v, imposing the order in which the steps are added to the plan. The planner also forces conditions that depend on active process effects to stay within those processes. In general, this approach represents a middle ground between least and total commitment, as the ordering constraints imposed by POPF result in the metric fluents being able to maintain an unambiguous value while avoiding commitments with ordering actions that do not affect the inserted step.

A new version of POPF was developed and introduced in [18] and its name is POPF2. There were mainly two improvements: it can make less commitments to ordering constraints and it implements a cost-optimisation approach that allows it to improve the first found plan by using a modified planning graph heuristic and by using anytime search. In other words, POPF terminates after finding the first plan while POPF2 seeks a plan that minimizes the total plan cost.

The way that POPF2 reduces the amount of ordering constraints is by doing a static analyses of the structure of the problem with the goal of identifying patterns of numeric behaviour in order to handle them more efficiently.

By doing this static analyses, POPF2 can identify the metric-tracking variables. These variables correspond to what a respective problem is trying to minimize as its cost function and their values are only changed by the action's effects. Since metric-tracking variables never appear in preconditions or duration constraints, their values do not affect the correctness of the plan and their values are also not used for numeric effects. Because of this, the order of the actions does not matter as we only need to know what were the executed actions at the end in order to know the final value of a tracking-variable v and POPF2 is able to update it without adding any ordering constraints.

Later, an extension of POPF2 named Forward-Chaining Partial-Order Planning with Timed Initial Fluents (POPF-TIF) was introduced in [19] and was also explored in [20] where the authors provided evidence that POPF-TIF is a powerful solution and extended the concept of using an external advisor. The name comes from the fact that this planner's goal is to handle problems with numeric Timed Initial Fluents (TIF) which are an extension of Timed Initial Literals (TIL) (a PDDL feature that allows the representation of exogenous events in a restricted way by assuming a value of *true* or *false* at a certain time step known to the planner before even knowing the actions that will take place) to numeric fluents. POPF-TIF presents two main improvements compared to POPF2: a better heuristic evaluation and the addition of alternative search methods using a combination between Enforced Hill Climbing (EHC) and Best First Search (BFS).

This new version of the planner is able to include exogenous events that assign values to fluents but not exogenous events that add or delete some effect. This is possible with the TIFs and the way it is done is by separating each domain fluent into two parts: one that is changed by the action and another that is changed by TIFs. An example of a fluent `(x)` would be represented in the domain as follows:

```
(at end (increase (x) 1))
```

where an action can increase the value of `(x)` by 1 as an effect. It could also be represented as

```
(at end (increase (x) (external)))
```

where the value of `(x)` is increased by the value of the `(external)` variable that is a TIF and is calculated externally.

As described in [20], POPF-TIF comes equipped with the ability to be connected to an external solver/advisor which allows the planner to perform sophisticated mathematical operations, which is something that PDDL alone is not prepared to handle. The connection between the planner and the external advisor is done through semantic attachments (concept that will be further explored in section 2.5.2). This external advisor can be used to compute numeric information and more effective heuristic values. This feature makes POPF-TIF a powerful framework especially for solving problems with numeric goals.

2.4 Sampling: Probabilistic Roadmaps

In path planning (or motion planning), the configuration space, also known as **C-space**, is the set of all possible robot configurations regarding its position and orientation. There is also the concept of **free space** which refers to the space where the robot can move free of obstacles.

Probabilistic Roadmaps (PRMs) [21] are a sampling method used to get an approximation of a roadmap. They are constructed from a set of configurations sampled from a collision free C-space. PRMs are often used to solve complex planning problems in high-dimensional C-spaces and in this context they are better than grid graphs since they can capture the C-space structure with many fewer nodes.

A particularity of the PRM is that the likelihood that the graph is a roadmap tends to 100% as the number of samples tends to infinity. There are algorithms that can be used to produce a PRM graph (nodes and edges) to approximately represent the free space. The PRM is probabilistically complete since an increase in the number of sampled points results in a higher probability of finding a path (if one exists).

2.5 Task-Motion Planning

Task planning is the aforementioned classical planning and it refers to the process of finding a discrete sequence of actions that lead from a starting state to a desired goal state.

Motion planning is the process of finding a sequence of collision-free poses, with the respective position and orientation values, that the robot has to go through to get from a starting point to a desired goal pose.

When planning in robotics there is the need to have both a task plan of the discrete actions and a motion plan describing the robot's motion, but combining both is a complex problem. Task-Motion Planning (TMP) involves an interaction between the decision-making process on both the discrete and continuous domains. Different approaches have been suggested for TMP and this is an area of research that keeps getting attention.

Initial TMP approaches would work by doing task planning first and then using the resulting sequence of actions to instruct the motion planner. This is done under the assumption that the robot will be able to carry out each motion with no restrictions, which is sometimes not the case, since there may be some geometric constraints. Other approaches try to mix task and motion planning, presenting a task-motion interface. The following sections will briefly show some recent TMP approaches and then get into more detail about one of them (MPTP) due to its relevance for this dissertation.

2.5.1 Task-Motion Planning Based Planners

In this section we compare three state of the art planning approaches which are meant for task-motion planning: Iteratively Deepened Task and Motion Planning (IDTMP) [22] (2016), PDDLStream [23] (2020) and Motion-Planning-aware Task Planning (MPTP) [24] (2021).

IDTMP [22] is a constraint based task-planning approach and it uses SMT as a way of making the task planner aware of geometric constraints and have some level of motion feasibility awareness at the task planning level. The IDTMP algorithm is probabilistically complete and is able to handle domains with diverse actions as well as model kinematic coupling.

PDDLStream [23] is a TMP approach that extends PDDL to incorporate sampling procedures. It also presents two new algorithms: Binding and Adaptive. These two new algorithms reduce PDDLStream planning to solving a series of finite PDDL problems. In [23] these algorithms are compared to each other and to previously existing algorithms and the results show that the Adaptive algorithm outperforms the other algorithms in three distinct domains. The Adaptive algorithm balances the time spent searching and sampling and aggressively explores many possible bindings. This algorithm outperforms existing algorithms (particularly in tight-constrained and cost-sensitive problems) by greedily optimizing discovered plans. PDDLStream can be used to plan for real-world robots operating using a diverse set of actions.

MPTP [24] is an approach that uses a task planner combined with an external solver which has a motion planner incorporated. The task planner used is POPF-TIF, but a different one could be used. It uses PDDL2.1 to define the task level actions and the way it works is by defining numeric fluents in the domain and taking advantage of the concept of semantic attachments in order to have an external solver calculate some of these numeric fluent variables, including the motion cost and the respective path plan, returning the cost to the task planner. MPTP is optimal at the task level.

Even though all three approaches are capable of doing task-motion planning they have limitations. PDDLStream is undecidable and its algorithms are semi-complete, meaning that it is complete only under feasible instances. IDTMP and MPTP are both probabilistically-complete. The task-motion interaction in IDTMP is done by the use of abstraction and refinement functions while MPTP uses semantic attachments as its task-motion interface.

Another difference is the fact that, while PDDLStream and IDTMP approaches both involve TMP for manipulation, MPTP is focused on TMP for navigation. MPTP involves a motion planning aware task planner, taking motion costs and motion plan feasibility into account.

2.5.2 MPTP

The Motion-Planning-aware Task Planning approach (MPTP) documented in [24] is a planning approach that will serve as baseline for this work. For this reason it is thoroughly explained in this section. Being a Motion-Planning-aware Task Planning approach means that there is an interaction between the task and motion planners. This approach is meant for navigating in large knowledge-intensive domains, it is probabilistically complete and returns a solution plan that is optimal at the task level. The MPTP framework is also prepared to deal with belief space planning, i.e. motion planning under motion and sensing uncertainty in partially-observable state-spaces, although this is not explored within the scope of this thesis.

In most traditional approaches, the high level task planning and the low level motion planning are done separately and PDDL and other planning frameworks make the assumption that the motion needed to carry out the planned tasks is achievable. However this assumption may not always be accurate, which can result in an unfeasible plan at the execution level. To present a solution for situations like this, the MPTP approach presents a task-motion interface layer that allows the motion planner to inform the task planner of the motion feasibility as well as the associated costs.

In this approach, there is a set of discrete actions that can be expanded by the task planner. Every time the task planner expands an action that requires robot motion, an external solver is called/triggered. When this happens, the discrete symbolic parameters are converted to the corresponding continuous geometric instantiations. These geometric instantiations are pre-sampled upon knowing the map of the environment (a roadmap-based sampling is used, specifically PRM). For each call of the external solver different motion plans are obtained for these instantiations and the best one is chosen according to a specific metric. Finally, the cost associated with the chosen motion plan is returned to the task planner and it corresponds to the cost of the associated action. Because the motion cost is returned as every action is expanded, the resulting plan will be optimal at the task level.

The cost used in this approach is the sum of the trajectory length and the cost associated with motion and sensing uncertainty. However, MPTP is capable of supporting any cost function and since uncertainty is not within the scope of this thesis, the respective motion and sensing uncertainty cost will not be considered.

Since this approach will be used to accomplish this thesis' goal, we provide a more practical and in depth explanation of how it works. MPTP uses POPF-TIF [19], which is an extension of POPF2 [18], as the task planner. POPF-TIF is used because it is a temporal task planner that can handle numeric time initial fluents. It is also PDDL based, using the version PDDL2.1 since this version supports durative actions as well as numeric fluents. The work in [19] and [20] introduces the concept of semantic attachments, allowing the POPF-TIF planner to have an external solver call and MPTP takes advantage of this concept to incorporate the motion planner.

According to [20], a **semantic attachment** evaluates numeric fluents using externally specified functions. For this interface to work, all the domain's numeric variables are categorised either as indirect variables (V^{ind}), direct variables (V^{dir}) or free variables (V^{free}). The planner, in this case POPF-TIF, determines the value of the V^{dir} variables and when these values change they affect the V^{ind} variables. The V^{ind} variables are calculated by the external function/advisor based on the information provided by the planner. The V^{free} variables are the remaining variables evaluated by the planner but they do not trigger any external computation. Figure 2.3 shows an overall view of the described structure.

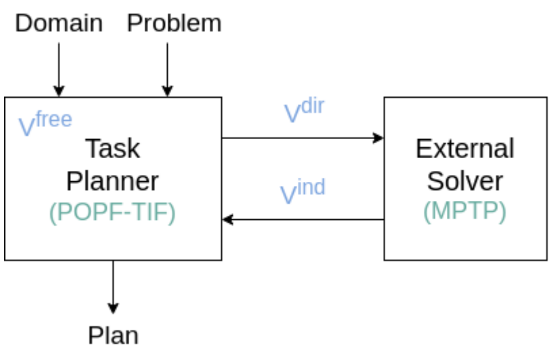


Figure 2.3: Overview of the overall structure of POPF-TIF combined with an external solver through the use of semantic attachments, where V^{dir} are the direct variables, V^{ind} the indirect variables and V^{free} the free variables that stay within the task planner

In other words, every time the POPF-TIF planner expands an action, the planner passes the V^{dir} variables to the external function and in turn the advisor returns the V^{ind} variables to the planner, so a semantic attachment can be seen as a function that is dependent on the V^{dir} variables and computes the V^{ind} variables. More precisely, when POPF-TIF first updates a state, the V^{dir} and V^{free} variables are computed. If any of the V^{dir} variables changed, then the external function is called to compute the V^{ind} variables and it receives all the V^{dir} values as input.

3 Proposed Solution

Contents

The approach that was used to fulfil the proposed goal is described in this chapter. The approach will be explained from a more theoretical perspective in the first part. After that, the practical implementation will be described in more detail, as well as the relevant developed code.

3.1 Methodology

3.1.1 The Space Assembly Domain

We consider a mobile robot navigating through a corridor inside the International Space Station (ISS) that holds, moves and assembles modules and we refer to it as the space assembly domain. Figure 3.1 shows a portion of the ISS[1] and we consider an environment similar to the Columbus lab which is the European laboratory that has a width of approximately 6.8 meters.

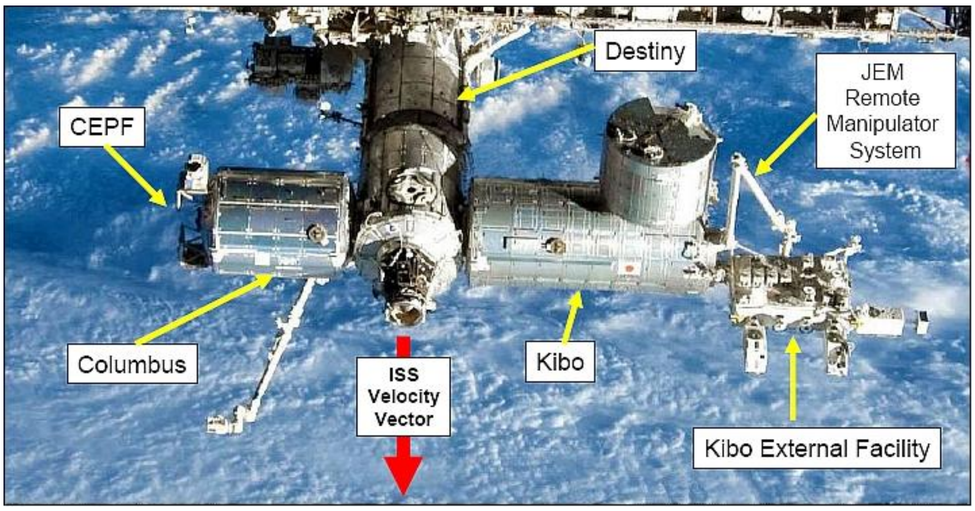


Figure 3.1: Portion of the International Space Station

To simplify the problem, we consider a 2D approximation of the ISS Columbus lab as a corridor of 3×6 meters. This means that the environment's map is known a priori. As can be seen in figure 3.2 we design the corridor as a grid of 1×1 meter squares so that each of these squares can be used as a location to the task planner when defining the domain.

In order to guarantee that each location contains at least one waypoint, the center of all grid squares is manually added to the roadmap and labeled as shown in figure 3.3. This figure also shows how the locations are going to be referred to in the problem file: we label each row with a letter in alphabetical order and each column with a number $i \in \mathbb{R}$ starting with zero.

In this domain we define 3 different types of objects:

[1] Image from https://earth.esa.int/web/eoportal/satellite-missions/i/iss-columbus

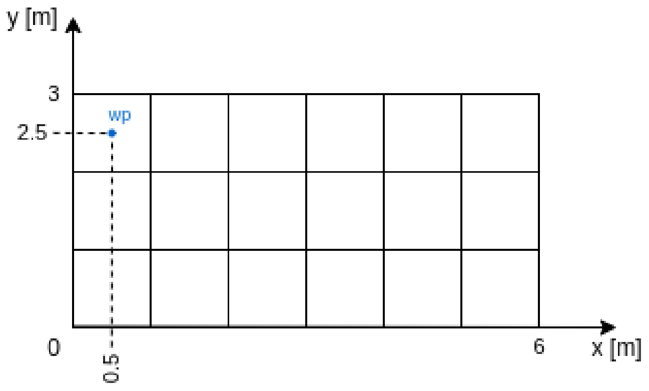


Figure 3.2: Representation of a 3×6 meter 2D ISS corridor approximation in an xy coordinate axis (wp stands for waypoint and represents an example of a sampled roadmap point and its respective coordinates)

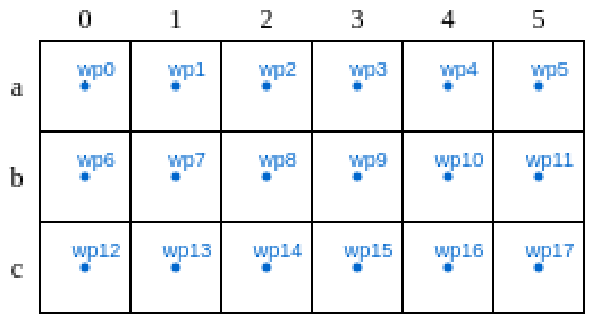


Figure 3.3: The 2D ISS corridor approximation with a waypoint for each location as well as the used nomenclature for each location (the rows are identified by letters and the columns by numbers, so the location containing $wp0$, for example, is location $a0$)

- **location** - a location is an area of the ISS corridor corresponding to a square from the grid approximation;
- **module** - these are assumed to be equally sized squares that correspond to the parts that the robot has to assemble. It is also assumed that a module occupies all the area covered by the location where it is placed;
- **robot** - the mobile robot that can grab modules and navigate through the ISS.

The goal is to have the robot assemble the modules into a desired configuration. A robot may not be carrying anything, which is encoded by the predicate `(empty ?r)`, where `r` is the robot, or it may be holding a module, which is encoded by the predicate `(loaded ?r ?m)`, where `m` is the module that is being carried by the robot `r`. In either case, the robot can move from one location to another using the high-level action `move_robot`. To describe the current location of a robot we use the predicate `(at ?r ?l)`, where `l` is the location where the robot `r` is.

If a location has a module placed on it, it is encoded by the predicate `(on ?m ?l)`, where `m` is the module placed on the location `l`. On the other hand, if a location does not have a module on it, then this corresponds to the predicate `(clear ?l)`, which encodes that the location `l` is clear.

One of the predicates defined in this domain is `(adjacent ?l0 ?l1)`, where `l0` and `l1` are two distinct locations that are adjacent to each other. This predicate has a crucial role in this domain, since we consider that a robot cannot leave a module floating inside the ISS, one of the preconditions for a robot to unload a module is that it has to be assembled to another module, which can be translated as: only unload the module if there is another module on an adjacent location. The high-level action that encodes a robot unloading/assembling a module is `assemble`. For simplicity purposes we assume that all modules can be assembled to each other with no restrictions.

For a robot to pick up a module it has to be within the reach of the module's location, which means that the robot has to be at a location adjacent to a different location with a module in it. The robot also has to be empty for it to be able to pick up a module since a robot can only grab one module at a time. The high-level action `load` is the one responsible for encoding a robot grabbing a module and therefore one of its consequences is that the robot is no longer empty, which can be translated in the negation of the predicate `(empty ?r)`.

In this domain there are also the following numeric fluents:

- `(act-cost)`: this is a direct variable that models the cost associated with the actions and it will be used as the cost to minimize by the task planner, as we will define in the problem file;
- `(extern)`: the motion cost, an indirect variable that is returned by the external solver. In the action that involves robot movement, after being computed by the external solver, this variable's value will be added to the `act-cost` value as an effect of this action;
- `(triggered ?from ?to)`: a direct variable used for the action that involves robot motion (which is `move_robot`) whose value is 1 at the beginning when the action is expanded and 0 once the action duration is complete. Every time this variable changes it triggers the motion planner in the external solver and gives two locations as arguments - these two locations represent the from and to positions where the robot is and to where it wants to move, respectively;
- `(occupied ?loc)`: a direct variable used for the `assemble` action that takes as argument the location where a module is being placed in order to let the external solver know that this location is now occupied. This is the core of our approach, we program the external solver to remove the waypoints that are within this occupied location for the motion planner to know that they are no longer available.
- `(unoccupied ?loc)`: a direct variable used for the `load` action. This is the opposite of the `occupied` variable, since its goal is to inform the external solver that a module was removed from a location

which means that this location is now unoccupied and the external solver can register that the respective waypoints became available for the motion planner to use.

The first three numeric fluents described above were already a part of the MPTP approach. The last two were implemented by us in order to achieve this thesis goal. The PDDL code for the described domain can be found in appendix A.

3.1.2 Choosing a TMP Approach

In order to fulfil the main goal of this thesis we need to choose an appropriate task-motion planning approach. In chapter 2, section 2.5.1, three state of the art TMP approaches are described and compared. All three approaches are capable of doing with task-motion planning, however we highlight the MPTP approach as the most appropriate to use in the context of this dissertation. This is because while other TMP approaches are more focused on TMP for manipulation, MPTP is focused on solving TMP for navigation which is what the space assembly domain requires since it revolves around a mobile robot carrying and assembling modules.

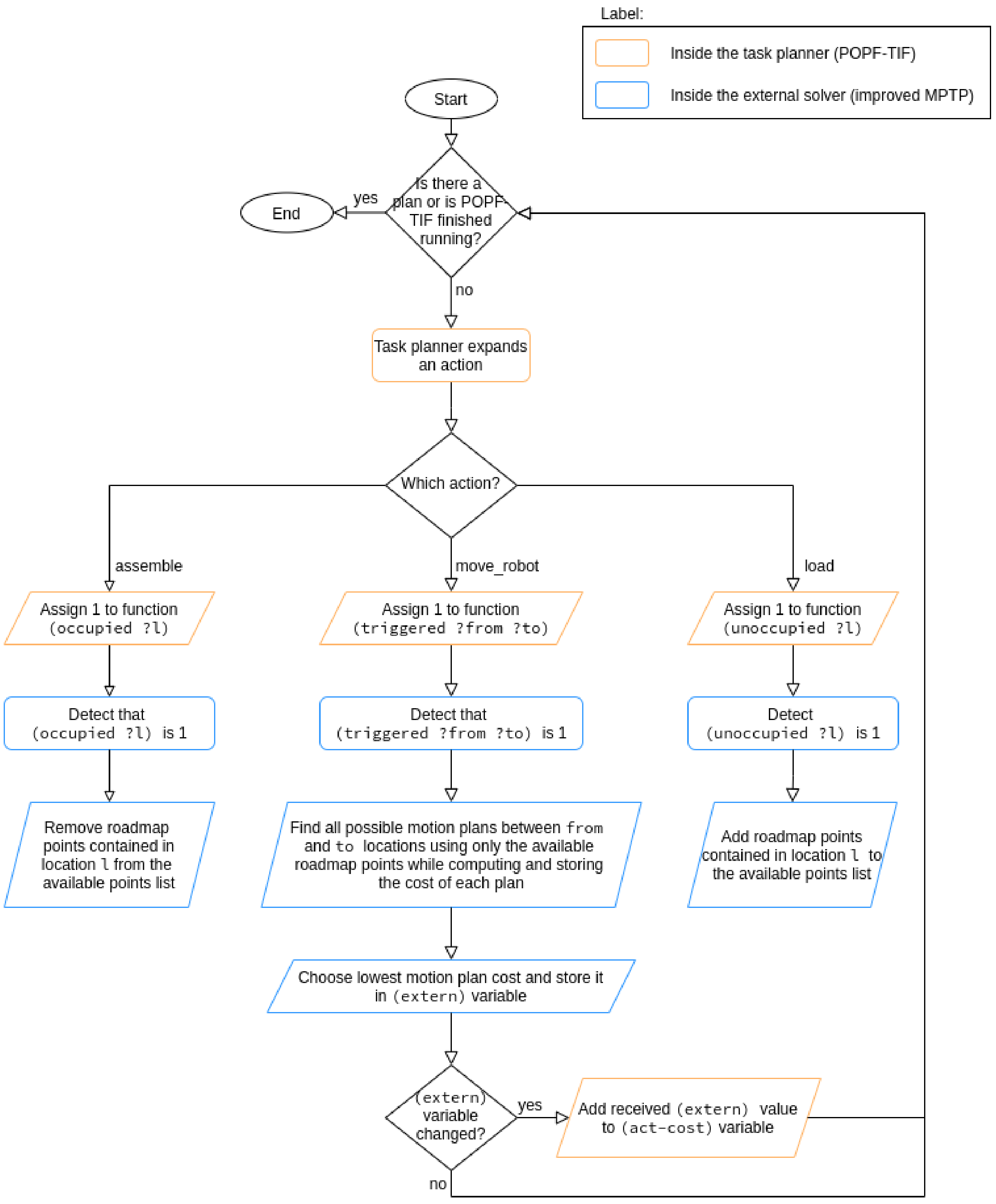
Taking into account the detailed explanation of MPTP in section 2.5.2, we proceed to explain the improvements that were added in order to be able to make this approach more suited for our domain and to have a more accurate task and motion plan.

3.1.3 External Solver Expansion

In section 2.5.2 it is explained that the communication between the task planner (POPF-TIF) and the external solver is done through the use of V^{dir} and V^{ind} variables. In our approach there are four direct variables: `(act-cost)`, `(triggered)`, `(occupied)` and `(unoccupied)`; and one indirect variable: `(extern)` which is calculated by the external solver.

The external solver defined by the MPTP approach mainly consists of a motion planner and it works by using `(triggered ?from ?to)` and `(increase (act-cost) (extern))`. The `triggered` numeric fluent is what allows the motion planner to receive information about the start and goal locations of the robot motion that the action involves and the indirect variable `extern` is the motion cost computed and returned to the task planner by the external solver/motion planner. The `act-cost` is the direct variable that the task planner uses to accumulate the cost of the expanded actions. Consequently, `(increase (act-cost) (extern))` is used as an effect in the `move_robot` action, since it involves robot motion, to increase the `act-cost` variable by the motion cost value returned by the motion planner, while the remaining high-level actions that do not involve robot motion use something like `(increase (act-cost) n)` where n is a value defined by the task planner which corresponds to the action's cost.

In order to accomplish this thesis' goal, we change the external solver to incorporate our approach as well as the motion planner defined by MPTP. An overview of our approach is shown in figure 3.4 in



Label:
Inside the task planner (POPF-TIF)
Inside the external solver (improved MPTP)
Start
Is there a plan or is POPF-TIF finished running?
yes
End
no
Task planner expands an action
Which action?
assemble
move_robot
load
Assign 1 to function (occupied ?l)
Assign 1 to function (triggered ?from ?to)
Assign 1 to function (unoccupied ?l)
Detect that (occupied ?l) is 1
Detect that (triggered ?from ?to) is 1
Detect (unoccupied ?l) is 1
Remove roadmap points contained in location l from the available points list
Find all possible motion plans between from and to locations using only the available roadmap points while computing and storing the cost of each plan
Add roadmap points contained in location l to the available points list
Choose lowest motion plan cost and store it in (extern) variable
(extern) variable changed?
yes
Add received (extern) value to (act-cost) variable
no

Figure 3.4: Flowchart showing an overview of planning with MPTP integrated with our approach

the form of a flowchart. Our goal is to see if the task plan changes when the high-level actions result in motion constraints in cases like the space assembly domain. In this domain a robot can move modules from one location to another. By doing so, the sampled robot poses that are contained within the new module's location are now unavailable and this may imply that the robot has to find a different path to go around the module, which may result in a larger path with a higher motion cost. A case like this would mean that the task plan would have to change according to the motion planner information in order to find the optimal plan.

We extend the external solver to keep track of a list of currently available waypoints. This includes all the map points that belong to locations that do not have a module placed in it. Every time an `assemble` action is expanded by the task planner (POPF-TIF), the `(occupied ?l)` numeric fluent informs the external solver of the location `l` that is now occupied and the list of available waypoints is updated by removing all the waypoints contained within the given location. A similar thing happens whenever the task planner expands a `load` action: the `(unoccupied ?l)` fluent informs the external solver of the location that has just been unoccupied and the list of available waypoints is updated by adding the waypoints within the respective location.

The way that the motion planner works in MPTP is by expanding the waypoints one by one between the initial and goal locations. By doing this, it calculates and stores the cost of each different available path between the two locations returning the smallest cost at the end. The cost we use is the sum of the distance between all the waypoints that constitute the path. Our implementation gives the motion planner access to an updated list of available waypoints at all times so whenever the motion planner is expanding the possible paths between two locations it checks if the next waypoint is available and in case it is not available, the motion planner skips it and does not take into account for the path.

The external solver can only return numeric variable values to POPF-TIF. This means that in case there are modules blocking all possible paths between two locations it is not possible for the external solver to inform the task solver that there is no feasible motion plan for that particular action. To get around this situation we make the external solver return a very high cost whenever there is no path available in order for the task planner not to include that action in the final plan. Since the goal is to find a plan that minimizes the `act-cost` variable which corresponds to the sum of all action costs from the task planner output plan, the motion cost has an impact on the chosen plan actions.

3.2 Implementation

POPF-TIF (implemented in C++) works with the following inputs: the PDDL domain file, the problem file and a user defined external solver/advisor (which is a dynamically loaded shared library). MPTP defines an external solver that works as the motion planner.

When an external solver is provided, POPF-TIF calls it using the function: `virtual map<string, double> callExternalSolver (map <string,double> initialState, bool isHeuristic)`. As can be seen by the function's declaration, POPF-TIF calls an external solver and gives the state as argument as well as a boolean variable that states if it wants to use a heuristic or not. The MPTP approach defines the motion planner in a function called $callExternalSolver$ that receives the 2 given arguments, one of them being the current state. This includes not the predicates defined on the $problem.pddl$ file, but a map from the names of the numeric fluents defined on the $domain.pddl$ file to their respective current values.

In addition to the $domain.pddl$ and $problem.pddl$ files, MPTP also needs three more files in order to work and to know the environment's map:

- $edge.txt$ - this file contains a list of all the edges in the form of pairs of connected waypoints. If we use the map on figure 3.3 an example of an edge would be $(wp0, wp1)$ which are two adjacent locations and therefore they are connected;
- $waypoint.txt$ - this file contains a list of all the map's waypoints, each with a respective set of coordinates x, y and θ for the orientation. Once again, if we use the map on figure 3.3 an example of a waypoint contained in this file would be $wp0[0.5, 2.5, 0]$;
- $region\_poses$ - this file lists the correspondence between each location and the respective set of waypoints that is contained within that location. Using the map on figure 3.3, what would appear in this file for location $a0$, for example, would be: $a0$ $wp0$, since $wp0$ is the only waypoint inside the location $a0$.

The domain file used in this work is the one contained in appendix A and described in section 3.1.1.

3.2.1 Automatic file generation

The problem file can change according to whatever example we want to test out. Since there can be an infinite number of problem files and consequently edge, waypoint and region_poses files (because these three change according to the map of the problem) we automated the process of creating these four files by developing a python script to quickly be able to define new problems.

The script uses the $pygame$ library in order to have a graphical interface that shows a grid. The size of the grid can be changed by choosing the number of rows and columns. When running this script a grid with the specified size will appear and three steps will be prompted:

1. first is the selection of all the locations that correspond to the initial module positions; each module will be named $m1$, $m2$, etc according to the order of selection;

2. then the selection of locations for the robots (the amount of locations selected will correspond to the amount of robots used and they will be named by order of selection as $r1$, $r2$, etc);
3. lastly is the selection of the locations corresponding to the desired goal configuration for all the modules selected in step 1.

These are all the required steps to build the necessary files. The waypoint file will define one waypoint for every location which corresponds to the center coordinates of each grid location. For the edges we consider that every location forms an edge with all directly adjacent locations vertically and horizontally (diagonal edges are not considered).

All the produced problem files have in common the initialization of the `(act-cost)` and `(extern)` variables with the value of zero and the goal of having a positive `(act-cost)` value ($>= 0$). All the files also have in common the fact that the robots are empty in the initial and goal state definitions and the specification of `(act-cost)` as the metric to minimize. They also all specify the robots and modules (quantity and names) as well as the locations for which we use the same nomenclature described in section 3.1.1.

3.2.2 Addition and Removal of Waypoints

Besides defining the space assembly domain and problem, our approach consists in improving MPTP in order for it to know and update which waypoints are available while the task planner is expanding the actions to find a solution plan.

We start by initializing a variable called $available\_wp\_list$ which is a list of all the available waypoints. This list is initialized with all the waypoints in the provided map and then we read from the $problem.pddl$ file in order to know which are the initial module locations that we initially remove from the available waypoints list.

Then we take advantage of the fact that the external solver can receive information from the task planner with the use of numeric fluents and define for that purpose the variables `(occupied ?l)` and `(unoccupied ?l)` where `l` is the occupied or unoccupied location, respectively. Every time the task planner expands an `assemble` or `load` action, one of the $at\ start$ effects of each of these actions is to change the value of the variables `occupied` and `unoccupied`, respectively, to 1.

Inside the external solver function there is a loop that goes through all the numeric fluents received as the state to check which ones have a value bigger than zero (every numeric fluent is initialized as zero so the ones which are zero did not suffer any changes). Since an action was expanded and at least one direct variable changed (in this case either `(occupied ?l)` or `(unoccupied ?l)` changed to 1), then inside the loop either the code on listing 3.1 or the code on listing 3.2 will be triggered. The $remove\_wps$ variable present in both these listings is a global boolean variable that is $true$ whenever we want to use our approach of updating the waypoints and $false$ to use the original MPTP approach without the waypoint removal.

Listing 3.1: C++ code for removing waypoints from the available waypoints list invoked by the $occupied$ numeric fluent

```
if (function=="occupied"){
    if (value>0 && remove_wps) {
        // Remove all the waypoints within the occupied location
        vector<string>::iterator it = region_mapping[location].begin();
        for (; it != region_mapping[location].end(); it++){
            available_wp_list.remove(*it);
        }
    }
}
```

In the MPTP approach there is a variable called $region\_mapping$ which is a map from all the locations to their respective list of waypoints, so this is essentially a map that keeps track of the waypoints that correspond to each location. The code on listing 3.1 will be invoked whenever an `assemble` action is expanded by the task planner and the variable $location$ is the information sent by the task planner of the location that has been occupied. The code then loops through the list of waypoints from $region\_mapping$ associated to that location to remove all of them from the list of available waypoints.

Listing 3.2: C++ code for adding waypoints to the available waypoints list invoked by the $unoccupied$ numeric fluent

```
if (function=="unoccupied"){
    if (value>0 && remove_wps) {
        // Add all the waypoints within the unoccupied location
        vector<string>::iterator it = region_mapping[location].begin();
        for (; it != region_mapping[location].end(); it++){
            available_wp_list.push_back(*it);
        }
    }
}
```

On the other hand, the code on listing 3.2 will be invoked whenever the task planner expands a `load` action. Just like in listing 3.1, the $location$ variable is also the information sent by the task planner and it informs the external solver of what location has been unoccupied. The code will then loop through the waypoints stored in the $region\_mapping$ variable associated to that location and add all of them back to the available waypoints list.

3.2.3 Motion Planner

When the task planner expands the `move_robot` action, one of its $at\ start$ effects is to change the value of the numeric fluent `(triggered ?from ?to)` to 1. Whenever this happens, the external solver will execute the motion planner code which expands the waypoints in order to find all possible paths from the $from$ location to the to location variables sent by the task planner and returning an external cost (which corresponds to the `(extern)` variable) of the path with the smallest cost.

There is a loop inside the motion planner that starts with the $from$ location, gets all of its child nodes, i.e. all the roadmap's poses that are directly connected to it according to the list of edges from the $edge.txt$ file, and iterates through all the child nodes, expanding all the possible waypoints until reaching the goal location.

In order to incorporate our approach into the motion planner we add a step that verifies if a child node is present in the list of available waypoints and only expand it in case it is. If it is not on the list it means that that waypoint has a module placed on it and the robot cannot use that point as a part of its path. Since the list is constantly updated while the planner is running, then the motion plan can be done with updated information on the available paths. In algorithm 3.1 there is a pseudo code description of the motion planner behaviour with our approach included and using the euclidean distance as the motion cost.

There are situations that may require some additional steps due to the way the planner and the external solver work. One situation is when there is no available path between the start and the goal locations, in other words when the path is blocked by modules. In this case we would want to inform the task planner that there is no available path between those two locations. However, POPF-TIF is only prepared for the external solver to return the indirect variables (V^{ind}) with numerical values in order to update the V^{dir} variables internally.

For this reason, in our approach, whenever the motion planner is unable to find a path because it is blocked and there are no available roadmap points between the two locations, it returns a very high motion cost, for example of 10000, so that the task planner will not choose that specific `move_robot` action to be included in the output plan.

Another situation to take into account is that whenever the goal waypoint is not available, instead of returning a high cost we just temporarily add it and expand it. This is because the task planner has the ability to know that the action cannot be done since it is able to use the predicate `(clear ?l)` to know that the goal location is actually not clear and has a module on it. Because the task planner will exclude this action, the external solver does not need to.

Algorithm 3.1: Motion planner with waypoint removal

```
Data: Roadmap (sampled poses and edges), $start$: starting location, $goal$: goal location
$cost \leftarrow 0$
available waypoints list ← list of the currently available roadmap poses
$to\_expand \leftarrow start$
$expanded \leftarrow empty$
while $to\_expand$ is not empty do
    $current\_node \leftarrow$ first element of $to\_expand$
    child nodes ← Find children of $current\_node$ (all waypoints that form edges with current node)
    $expanded \leftarrow$ add $current\_node$
    foreach child node not in $expanded$ do
        if $child$ exists in available waypoints list then
            $to\_expand \leftarrow child$
                        ▷ Calculate euclidean distance between $current\_node$ and $child$
            $distance \leftarrow \sqrt{(x_{current} - x_{child})^2 + (y_{current} - y_{child})^2}$
            cost of current node ← cost of current node + $distance$
            if $child$ is $goal$ then
                Store this path's cost
                Break out of for loop
    $to\_expand \leftarrow$ remove current node
$extern \leftarrow$ Choose smallest path cost
Print motion path into output file
return $extern$
```

4 Results

Contents

We start by testing the difference between planning with a task planner combined with a motion planner and planning only using a task planner (and forcing it to make a motion plan, which can be done by changing the definition of the domain). After that, we test our waypoint removal approach in a simple example to show how the incorporation of a motion planner with updates on the roadmap points availability can change the robot's motion plan. We then create a more elaborate scenario where a robot navigates in a corridor similar to the Columbus lab using the approximation described in section 3.1.1 and represented in figure 3.2 to show how the output task plan can change by incorporating the waypoint removal approach.

All the problem files needed for the tests were generated using the code developed for that purpose described in section 3.2.1. POPF-TIF planner was run using Ubuntu version 18.04.

4.1 Motion Plan: Using Task Planner vs Using Motion Planner

To test the difference between using only the task planner (and getting a motion path by changing the way the domain is defined) and using the task planner along with a motion planner with the domain definition described in section 3.1.1 we made a simple change in the domain file from listing A.1: we added the precondition in listing 4.1 to the `move_robot` action, which forces the robot to only move between adjacent locations. This way the solution plan outputted by the task planner will have a description of the locations where the robot had to walk through in order to get from one location to another.

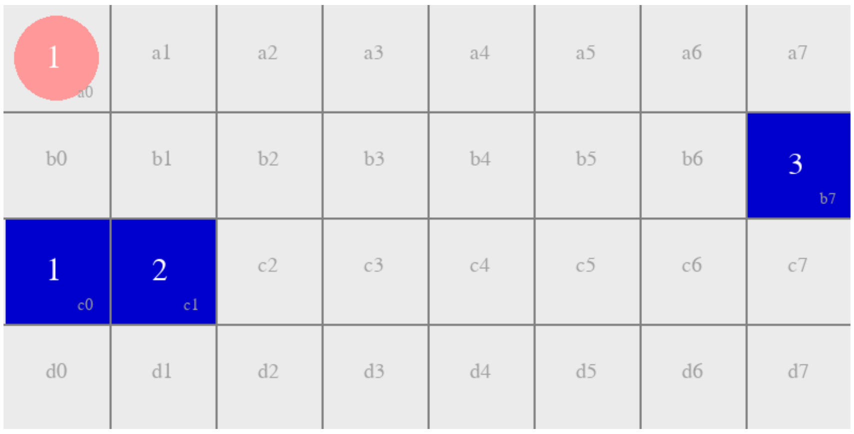
Listing 4.1: Precondition added to the domain file to force the robot to only be able to move between adjacent locations

```
(at start (adjacent ?from ?to))
```

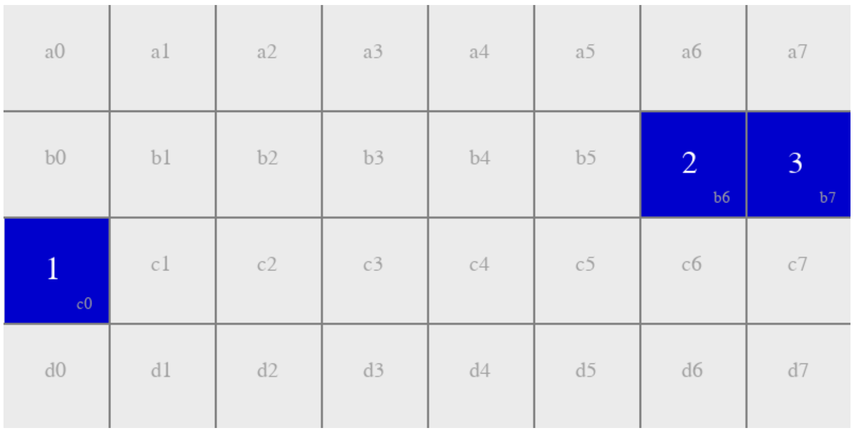
When the external solver is being used, the `move_robot` action's cost is the cost returned by the motion planner. Since we are only using the task planner in this test we changed this cost to a fixed value of 1.

The problem that was defined for this test is shown in figure 4.1 where the blue squares represent the positions of the three modules numbered from 1 to 3 and the red circle represents the mobile robot. We can see from figure 4.1(a) that the robot's initial position is $a0$ and modules 1, 2 and 3 are initially in locations $c0$, $c1$ and $b7$, respectively. From figure 4.1(b) we can see that only module 2 is required to be relocated as its goal position is location $b6$. This problem requires the robot to move close to module 2 and then move once again to take the module to its goal position.

In this test we are going to run the task planner with two different domains: the one in listing A.1 and the same one but with both changes mentioned above forcing the robot to only move between adjacent locations.



(a) Initial state



(b) Goal state

Figure 4.1: Visual representation of a problem in a 4×8 map with three modules and one mobile robot where the blue squares represent the module positions and the red circle represents the initial robot position

The output from POPF-TIF after running with the original domain was the one in table 4.1 and the POPF-TIF output after running with the modified domain that forces movement between adjacent locations is the one in table 4.2. In both figures the numbers inside the squared brackets represent the duration of the respective action and the numbers on the left of each action represent the sum of the actions duration so far.

As we can see, the output plan for the original domain in table 4.1 is composed by four actions. There is a move action for the robot to go from region $a0$ to $b1$, when in $b1$ the robot can execute a load action and load module $m2$ which is within the robot's reach since the module's location is $c1$ which is adjacent to the current robot's location $b1$. Then another `move_robot` action is executed from $b1$ to $a6$ where the robot can then unload module $m2$ at its goal location $b6$ assembling it to module $m3$ which is

Table 4.1: POPF-TIF output after running the problem defined in figure 4.1 with the domain described in listing A.1

Actions
`0.000: (move_robot r1 a0 b1)  [20.000]`
`20.001: (load r1 m2 b1 c1)  [5.000]`
`20.002: (move_robot r1 b1 a6)  [20.000]`
`40.003: (assemble r1 m2 m3 a6 b6 b7)  [10.000]`

in the adjacent location $b7$. The total plan cost is 4 and the total planning time was 0.13 seconds.

By running the task planner alone with this domain we cannot have an idea of the path that the robot has to take in each `move_robot` action. However, by running the task planner using the modified domain, as table 4.2 shows, the output plan discriminates all the locations that the robot goes through and thus having a motion plan. As the plan indicates, to get from $a0$ to $c2$ the robot goes through locations $a1$, $a2$ and $b2$ and in the second `move_robot` action the robot starts in its current position $c2$ and it goes through locations $c3$, $c4$, $b4$ to finally reach $c5$. The total plan cost is 10 and the total planning time was 0.35 seconds.

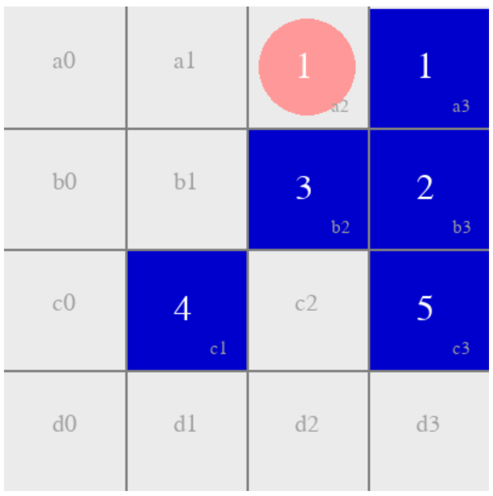
Table 4.2: POPF-TIF output after running the problem defined in figure 4.1 with the modified domain that forces robot movement between adjacent locations only

Actions
`0.000: (move_robot r1 a0 a1)  [20.000]`
`20.001: (move_robot r1 a1 a2)  [20.000]`
`40.002: (move_robot r1 a2 b2)  [20.000]`
`60.003: (move_robot r1 b2 c2)  [20.000]`
`80.004: (load r1 m2 c2 c1)  [5.000]`
`80.005: (move_robot r1 c2 c3)  [20.000]`
`100.006: (move_robot r1 c3 c4)  [20.000]`
`120.007: (move_robot r1 c4 b4)  [20.000]`
`140.008: (move_robot r1 b4 b5)  [20.000]`
`160.009: (assemble r1 m2 m3 b5 b6 b7)  [10.000]`

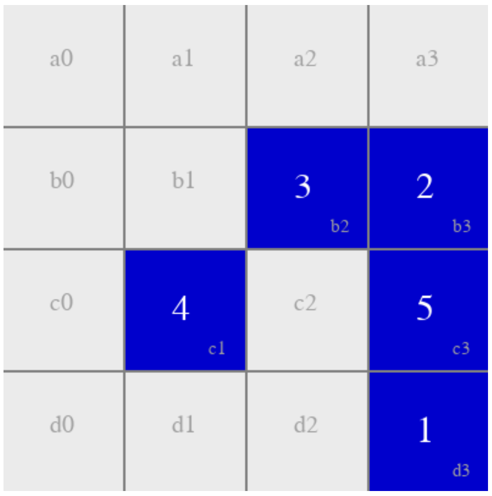
Comparing both results we can see that using the domain definition to force this kind of motion plan implies a higher cost: the original domain had a cost of 4 while the one that forced a motion discrimination had a cost of 10; and it also implies a higher execution time: the original domain had an execution time of 0.13 seconds while the other one was 0.35 seconds which is almost 3 times higher. In problems that require a larger amount of move actions the cost would increase exponentially, as well as the execution time, proving that making the task planner do a motion plan is not an efficient approach compared to using a domain that implies less actions and combining it with an external motion planner.

4.2 Motion Plan With Waypoint Removal

In order to see the difference between the motion plan produced by the original MPTP approach and the motion plan produced by MPTP with waypoints update we define a problem where a robot needs to go around some modules in order to get to its goal location. For that purpose we define the problem that is visually represented in figure 4.2.



(a) Initial state



(b) Goal state

Figure 4.2: Visual representation of a problem in a 4×4 map with five modules and one mobile robot where the blue squares represent the module positions and the red circle represents the initial robot position

Figure 4.2(a) shows the initial configuration of the five modules and the initial robot position which is $a2$. The goal configuration is shown in figure 4.2(b) where the only module that changed position was module $m1$ which started in location $a3$ and whose goal location is $d3$. So the goal of this problem is for the robot to get module $m1$ from its initial to its goal location.

After running POPF-TIF with both the original MPTP external solver and MPTP with the waypoint removal approach, both output plans were the same and the one shown in table 4.3.

Table 4.3: POPF-TIF output after running it with both the original MPTP external solver and MPTP with the waypoint removal approach - the output task plan was the same

Actions
`0.000: (load r1 m1 a2 a3)   [5.000]`
`0.001: (move_robot r1 a2 d2)   [20.000]`
`20.002: (assemble r1 m1 m5 d2 d3 c3)   [10.000]`

Both plans include one `load` action where the robot loads module $m1$, followed by one `move_robot` action where the robot goes from $a2$ to $d2$ and then an `assemble` action where the robot assembles module $m1$ to $m5$. The difference between the two plans is in the motion path and consequently in the total plan cost.

We consider a cost of 1 for both the `load` and `assemble` actions. The motion planner cost is the calculated euclidean distance between all the waypoints included in the robot's path from one location to another. In this case we are using a map that has one waypoint for every location which corresponds to its center coordinates. For this reason, the motion cost between every two adjacent locations is going to be 1. The labels of the roadmap points are shown in figure 4.3.

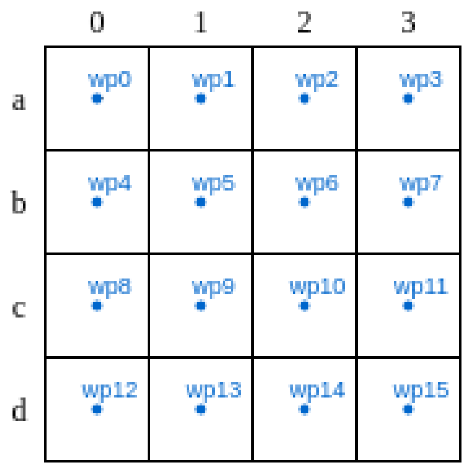
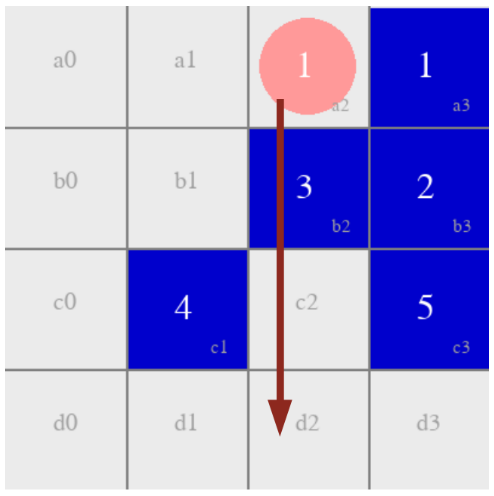


Figure 4.3: Roadmap points used in the problem described in figure 4.2 and their respective labels as well as labeling of grid locations (rows are identified by letters and columns by numbers)

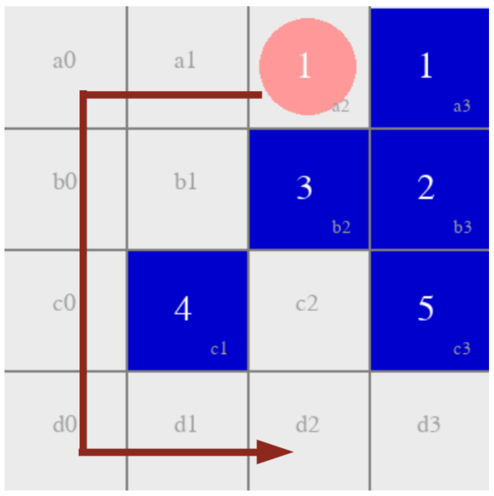
The motion paths outputted by the $output\_motion.txt$ file for both approaches is shown in figure 4.4. For the first approach the cost was 5. This corresponds to a cost of 1 from the `load` action plus 1 from the `assemble` action and a remaining cost of 3 for the `move_robot` action. The output motion path that could be seen by the $output\_motion.txt$ file was the following sequence of waypoints (reading from left to right and taking into account the waypoints nomenclature of figure 4.3): $wp2 \rightarrow wp6 \rightarrow wp10 \rightarrow wp14$. This motion path is highlighted in figure 4.4(a).

For the second approach with waypoints removal the total plan cost was 9. This cost corresponds to the sum of a cost of 1 from the `load` action with another cost of 1 from the `assemble` action with a remaining cost of 7 for the `move_robot` action. This move action has a higher cost than the previous approach since the waypoints from all the locations that contain a module were removed. This includes waypoints $wp6$, $wp7$, $wp9$ and $wp11$ which the robot is not able to use for its path. For this reason, the outputted motion plan for this second approach was a sequence of roadmap points that go around the modules as follows: $wp2 \rightarrow wp1 \rightarrow wp0 \rightarrow wp4 \rightarrow wp8 \rightarrow wp12 \rightarrow wp13 \rightarrow wp14$. A visual representation of this motion path can be seen in figure 4.4(b).

Comparing both results we conclude that using the roadmap points update approach is beneficial in situations like the one described in this section where there are obstacles that are unknown to the task



(a) Original MPTP



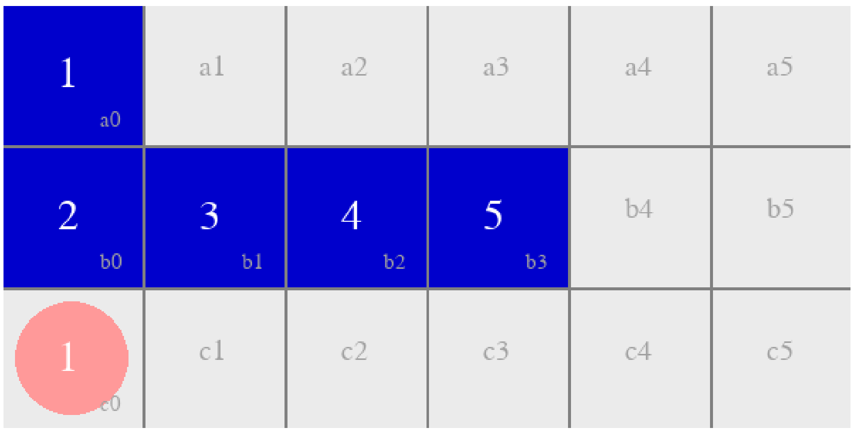
(b) MPTP with waypoint removal

Figure 4.4: Motion plan (represented by the dark red arrows) outputted by MPTP and by MPTP with the waypoint removal approach for the problem in figure 4.2

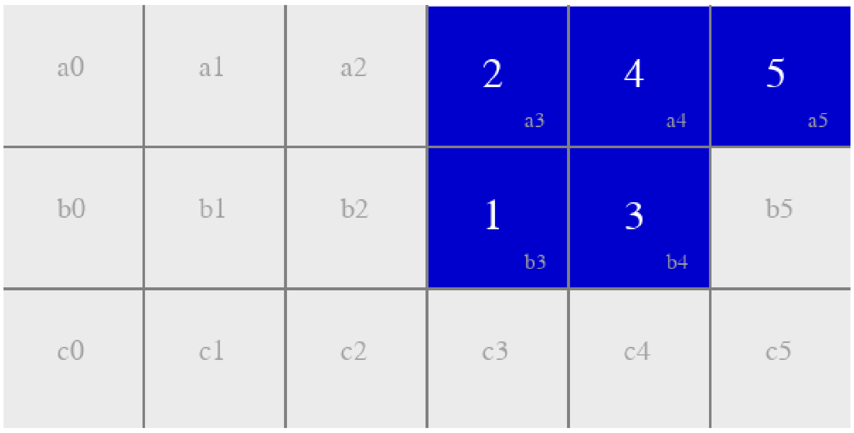
planner due to the way the domain is defined (which is a domain that implies less actions and whose advantages were explained in section 4.1). Even though results from figure 4.4(b) imply a higher cost than the results from figure 4.4(a), the motion plan is better and more accurate since the one in figure 4.4(a) would fail at the robot's execution phase due to physical motion constraints that were not taken into account during the planning phase.

4.3 Columbus Lab Scenario

The Columbus lab scenario consists in a simple environment which was specifically designed as a proof-of-concept to compare and study the advantages of using an integrated task-motion planning approach in contrast with using task and motion planning separately. This scenario consists of the domain described in section 3.1.1 with a map that is similar to a 2D approximation of the ISS Columbus laboratory (hence the name Columbus lab scenario), which is the European laboratory. The map used in this test can be seen in figure 3.2 and uses the location's names and the exact roadmap points pictured in figure 3.3, which means that each location contains a roadmap point that corresponds to its center coordinates. The problem that was defined for this test is depicted in figure 4.5 where there is a mobile robot and five modules that are to be all moved to a different configuration.



(a) Initial state



(b) Goal state

Figure 4.5: Visual representation of the Columbus lab corridor problem in a 3×6 map with five modules and one mobile robot where the blue squares represent the module positions and the red circle represents the initial robot position

The modules are labeled from 1 to 5 and their initial positions are the ones shown in figure 4.5(a) which are $a0$, $b0$, $b1$, $b2$ and $b3$, respectively. The robot's initial location is $c0$, close to module $m2$. The robot has to move all modules to their goal locations which are $b3$, $a3$, $b4$, $a4$ and $a5$, respectively.

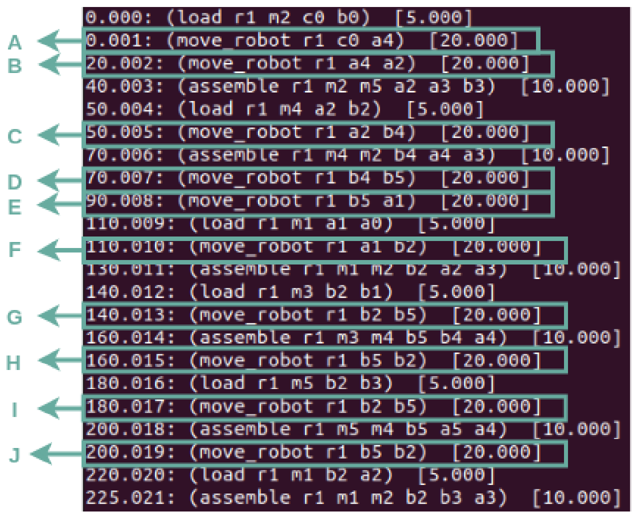
After running this problem using only the task planner POPF-TIF we get the output solution plan detailed in table 4.4. The table shows all the output task plan actions in order. This is a valid plan and all `move_robot` actions have at least one possible path with no modules blocking it, which means that if we were to do motion planning after this task plan it would be possible to find a valid motion plan with no modules blocking the way.

We proceeded to manually build the motion plan with the smallest cost possible by following the given task plan and taking into account all the module positions upon the execution of each `load` and `assemble` action. The result was the motion plan described in figure 4.6.

Taking the motion plan in figure 4.6 we then calculated the euclidean distances of all the paths that make up the motion plan in order to know the total plan cost of solving this problem using task and motion planning separately. The cost of each action is shown in table 4.4 in front of the respective actions as well as the total plan cost at the bottom of the table. The cost of the solution by doing task planning separately from motion planning is 55.

Table 4.4: Output task plan obtained by running the problem described in figure 4.5 using only the task planner (POPF-TIF) and the cost associated with each action (the costs of the move_robot actions were calculated using the motion plan described in figure 4.6)

Actions	Cost
`0.000: (load r1 m2 c0 b0) [5.000]`	1
`0.001: (move_robot r1 c0 a4) [20.000]`	6
`20.002: (move_robot r1 a4 a2) [20.000]`	2
`40.003: (assemble r1 m2 m5 a2 a3 b3) [10.000]`	1
`50.004: (load r1 m4 a2 b2) [5.000]`	1
`50.005: (move_robot r1 a2 b4) [20.000]`	5
`70.006: (assemble r1 m4 m2 b4 a4 a3) [10.000]`	1
`70.007: (move_robot r1 b4 b5) [20.000]`	1
`90.008: (move_robot r1 b5 a1) [20.000]`	7
`110.009: (load r1 m1 a1 a0) [5.000]`	1
`110.010: (move_robot r1 a1 b2) [20.000]`	2
`130.011: (assemble r1 m1 m2 b2 a2 a3) [10.000]`	1
`140.012: (load r1 m3 b2 b1) [5.000]`	1
`140.013: (move_robot r1 b2 b5) [20.000]`	5
`160.014: (assemble r1 m3 m4 b5 b4 a4) [10.000]`	1
`160.015: (move_robot r1 b5 b2) [20.000]`	5
`180.016: (load r1 m5 b2 b3) [5.000]`	1
`180.017: (move_robot r1 b2 b5) [20.000]`	5
`200.018: (assemble r1 m5 m4 b5 a5 a4) [10.000]`	1
`200.019: (move_robot r1 b5 b2) [20.000]`	5
`220.020: (load r1 m1 b2 a2) [5.000]`	1
`225.021: (assemble r1 m1 m2 b2 b3 a3) [10.000]`	1
TOTAL:	55



```
0.000: (load r1 m2 c0 b0)  [5.000]
0.001: (move_robot r1 c0 a4)  [20.000]
20.002: (move_robot r1 a4 a2)  [20.000]
40.003: (assemble r1 m2 m5 a2 a3 b3)  [10.000]
50.004: (load r1 m4 a2 b2)  [5.000]
50.005: (move_robot r1 a2 b4)  [20.000]
70.006: (assemble r1 m4 m2 b4 a4 a3)  [10.000]
70.007: (move_robot r1 b4 b5)  [20.000]
90.008: (move_robot r1 b5 a1)  [20.000]
110.009: (load r1 m1 a1 a0)  [5.000]
110.010: (move_robot r1 a1 b2)  [20.000]
130.011: (assemble r1 m1 m2 b2 a2 a3)  [10.000]
140.012: (load r1 m3 b2 b1)  [5.000]
140.013: (move_robot r1 b2 b5)  [20.000]
160.014: (assemble r1 m3 m4 b5 b4 a4)  [10.000]
160.015: (move_robot r1 b5 b2)  [20.000]
180.016: (load r1 m5 b2 b3)  [5.000]
180.017: (move_robot r1 b2 b5)  [20.000]
200.018: (assemble r1 m5 m4 b5 a5 a4)  [10.000]
200.019: (move_robot r1 b5 b2)  [20.000]
220.020: (load r1 m1 b2 a2)  [5.000]
225.021: (assemble r1 m1 m2 b2 b3 a3)  [10.000]
```

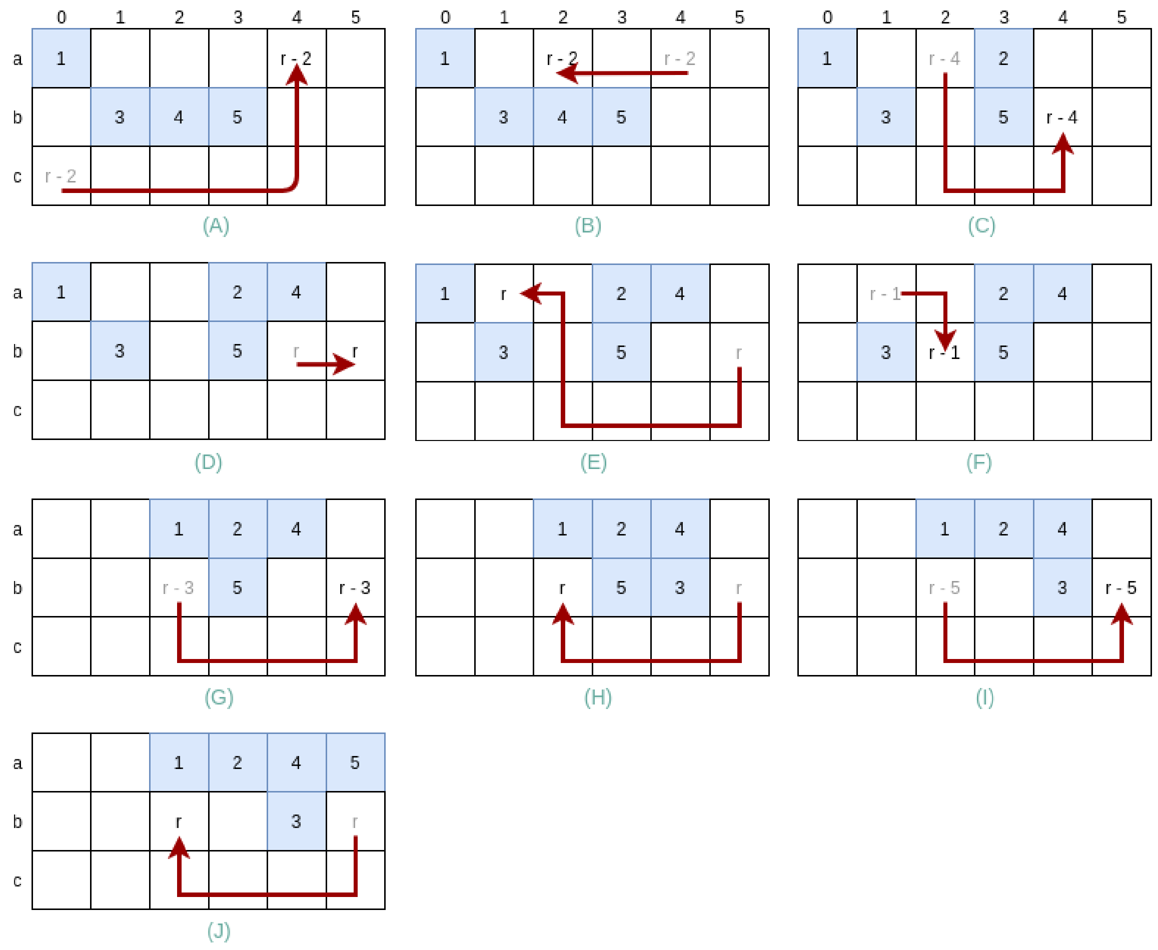


Figure 4.6: Task plan outputted by running the problem from figure 4.5 using only the task planner (POPF-TIF) as well as the correspondent visual representation of the minimum cost manually built motion plan for every move_robot action (labeled with capital letters A-J); r represents the robot and $r - n$ stands for the robot loaded with module n

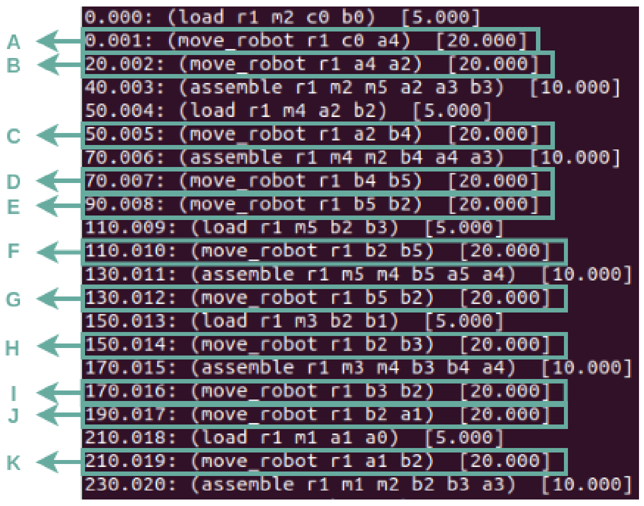
Then we ran the Columbus lab scenario using MPTP along with the waypoint removal approach (in order to get an accurate motion plan for this domain) and we obtained the solution task plan detailed in table 4.5. This table also shows the cost associated with each action, being the `move_robot` action cost the one calculated by the external solver with the roadmap points update.

This test's motion plan was printed in the *output_motion.txt* file and its graphical representation is shown in figure 4.7 by the red arrows. There is a path representation for each `move_robot` action. The total solution plan cost for this approach is the one shown at the bottom of table 4.5 which is 41.

Table 4.5: Output task plan obtained by running the problem described in figure 4.5 using a mixed task and motion planning approach (MPTP) with the waypoint removal approach and the cost associated with each action

Actions	Cost
`0.000: (load r1 m2 c0 b0) [5.000]`	1
`0.001: (move_robot r1 c0 a4) [20.000]`	6
`20.002: (move_robot r1 a4 a2) [20.000]`	2
`40.003: (assemble r1 m2 m5 a2 a3 b3) [10.000]`	1
`50.004: (load r1 m4 a2 b2) [5.000]`	1
`50.005: (move_robot r1 a2 b4) [20.000]`	5
`70.006: (assemble r1 m4 m2 b4 a4 a3) [10.000]`	1
`70.007: (move_robot r1 b4 b5) [20.000]`	1
`90.008: (move_robot r1 b5 b2) [20.000]`	5
`110.009: (load r1 m5 b2 b3) [5.000]`	1
`110.010: (move_robot r1 b2 b5) [20.000]`	3
`130.011: (assemble r1 m5 m4 b5 a5 a4) [10.000]`	1
`130.012: (move_robot r1 b5 b2) [20.000]`	3
`150.013: (load r1 m3 b2 b1) [5.000]`	1
`150.014: (move_robot r1 b2 b3) [20.000]`	1
`170.015: (assemble r1 m3 m4 b3 b4 a4) [10.000]`	1
`170.016: (move_robot r1 b3 b2) [20.000]`	1
`190.017: (move_robot r1 b2 a1) [20.000]`	2
`210.018: (load r1 m1 a1 a0) [5.000]`	1
`210.019: (move_robot r1 a1 b2) [20.000]`	2
`230.020: (assemble r1 m1 m2 b2 b3 a3) [10.000]`	1
TOTAL:	41

The main difference between both approaches is that the first one does task and motion planning separately by doing the task planning first and only later the motion planning, while the second approach does task-motion planning combined using the MPTP approach. Starting by comparing both results in terms of the task actions we can observe that the actions as well as the sequence of actions differs from the first approach to the other. This is because the second approach is aware of the motion constraints as the robot loads and assembles modules into different positions of the map and with the incorporation of the motion planner along with that information, the task planner is able to choose the actions in a more informed way, so it will switch the order in which the robot moves modules in order to avoid producing motion constraints that will imply a higher cost when doing the motion plan, which is what happened in



```
0.000: (load r1 m2 c0 b0)  [5.000]
0.001: (move_robot r1 c0 a4)  [20.000]
20.002: (move_robot r1 a4 a2)  [20.000]
40.003: (assemble r1 m2 m5 a2 a3 b3)  [10.000]
50.004: (load r1 m4 a2 b2)  [5.000]
50.005: (move_robot r1 a2 b4)  [20.000]
70.006: (assemble r1 m4 m2 b4 a4 a3)  [10.000]
70.007: (move_robot r1 b4 b5)  [20.000]
90.008: (move_robot r1 b5 b2)  [20.000]
110.009: (load r1 m5 b2 b3)  [5.000]
110.010: (move_robot r1 b2 b5)  [20.000]
130.011: (assemble r1 m5 m4 b5 a5 a4)  [10.000]
130.012: (move_robot r1 b5 b2)  [20.000]
150.013: (load r1 m3 b2 b1)  [5.000]
150.014: (move_robot r1 b2 b3)  [20.000]
170.015: (assemble r1 m3 m4 b3 b4 a4)  [10.000]
170.016: (move_robot r1 b3 b2)  [20.000]
190.017: (move_robot r1 b2 a1)  [20.000]
210.018: (load r1 m1 a1 a0)  [5.000]
210.019: (move_robot r1 a1 b2)  [20.000]
230.020: (assemble r1 m1 m2 b2 b3 a3)  [10.000]
```

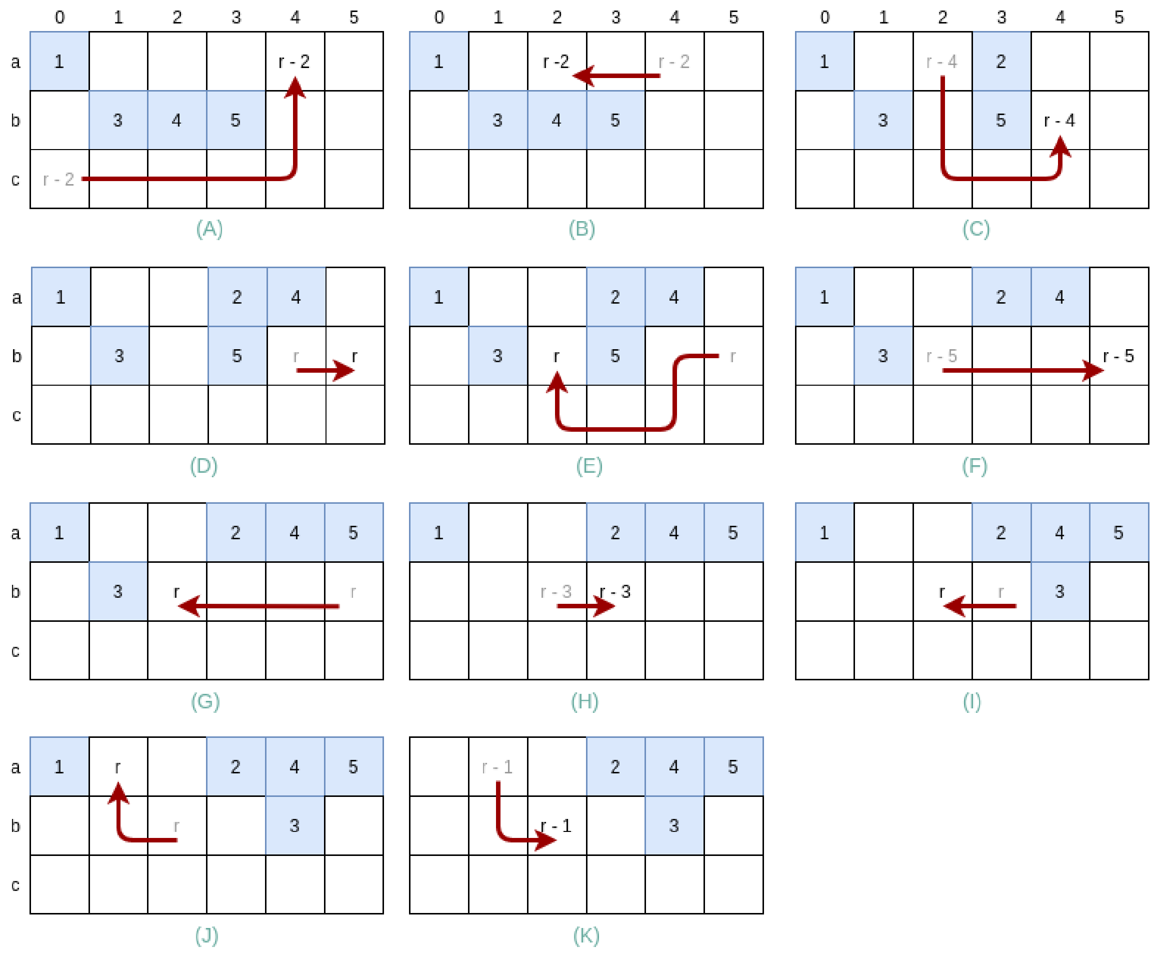


Figure 4.7: Task plan outputted by running the problem from figure 4.5 using a mixed task-motion planing approach (MPTP) with the waypoint removal approach as well as the correspondent visual representation of the outputted motion plan (as outputted in the $output_motion.txt$ file) for every move_robot action (labeled with capital letters A-K); r represents the robot and $r - n$ stands for the robot loaded with module n

the case of the first approach.

From this comparison we can conclude that an informed mixed task-motion planning approach allows the motion planner to be aware of the physical constraints imposed by the task planner’s expanded actions while planning, which will help produce a solution plan with a smaller cost, and consequently with higher quality than the one produced by the task planner working separately from the motion planner. We can also conclude that the task plan, i.e. the discrete actions that are executed as well as their order, may change when using this mixed informed approach in order to accommodate a feasible, realistic and smaller cost motion plan, which is highly beneficial in scenarios like the space assembly domain that involve robot navigation.

5

Conclusion

Contents

5.1 Conclusions

Planning in domains like the space assembly domain where a mobile robot navigates through the environment to move and assemble parts can be challenging since it involves task-motion planning. We have seen that it is possible to define the domain in a way that we can obtain some sort of motion plan by only using the task planner if we force the robot to only move between adjacent locations. We can even divide the map into smaller areas that we would define as locations in order to have a more accurate motion plan. However, by performing the tests described in section 4.1 we can conclude that this is not a very efficient way of performing task and motion planning combined since this implies a very high cost which only gets higher for problems that involve more robot movement actions. It is also a very time consuming approach which might make it difficult to find a solution plan within a reasonable amount of time.

We introduced an improvement into the MPTP approach that enabled the external solver to inform the motion planner of the availability of each roadmap point, updating this information according to the actions expanded by the task planner, while the planner is working to find a solution. After running some tests, we were able to show that using this informed task-motion planning approach was highly beneficial in the space assembly domain in comparison to running the task planner and then the motion planner separately. This is due to the fact that the mixed TMP approach is able to produce a more realistic task plan since it takes into account some motion constraints. By doing task and motion planning separately the task planner can in some situations produce a plan that may be unfeasible due to motion constraints or it can also be a task plan that implies a higher motion cost than other possible task plans.

In section 4.3 we showed a problem that we defined as the Columbus Lab scenario where we made a 2D approximation of the ISS European laboratory and defined a problem with a mobile robot and five modules to assemble into a goal configuration. With this example we were able to test both TMP approaches and conclude that the mixed informed TMP approach produced better results as some physical constraints imposed by the task planner's expanded actions were taken into account during planning, which led to a sequence of high-level task actions that produced a lower cost motion plan and an overall lower total cost for the task-motion plan. This shows that using a mixed TMP approach in scenarios that involve robot navigation is very beneficial in terms of task-motion plan quality.

5.2 System Limitations and Future Work

This work could be extended to a multi-robot environment where the task-motion planning could be informed of each robot's location while expanding the actions during the planning phase. This will allow the planner to build a solution plan that is more likely to be feasible at the execution level due to taking more motion constraints into consideration early on. This would allow some sort of collision avoidance mechanism for the robots not to collide due to the produced task-motion plan.

It could also be extended to take into account a micro-gravity environment, making it into a more accurate approximation of the in-space autonomous assembly. Future work should also be improved to a 3D simulation of the environment and include the 3D free-flying motion range of in-space assembler robots like Astrobee.

Another possible improvement would be to use a different task planner to study how the solution plan's quality would change as well as the time required to plan in more complex problems and how it would change in terms of plan optimality.

Bibliography

[1] D. Thomas, M. P. Snyder, M. Napoli, E. R. Joyce, P. Shestople, and T. Letcher, “Effect of acrylonitrile butadiene styrene melt extrusion additive manufacturing on mechanical performance in reduced gravity,” in *AIAA SPACE and astronautics forum and exposition*, 2017, p. 5278.

[2] C. Jewison, D. Sternberg, B. McCarthy, D. W. Miller, and A. Saenz-Otero, “Definition and testing of an architectural tradespace for on-orbit assemblers and servicers,” 2014.

[3] T. Smith, J. Barlow, M. Bualat, T. Fong, C. Provencher, H. Sanchez, E. Smith *et al.*, “Astrobee: A new platform for free-flying robotics on the international space station,” in *Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*, 2016.

[4] P. R. Norvig and S. A. Intelligence, *A modern approach*. Prentice Hall Upper Saddle River, NJ, USA:, 2002.

[5] M. Ghallab, D. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.

[6] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson *et al.*, “Pddl— the planning domain definition language,” Technical Report, Tech. Rep., 1998.

[7] D. M. McDermott, “The 1998 ai planning systems competition,” *AI magazine*, vol. 21, no. 2, pp. 35–35, 2000.

[8] H. A. Kautz, B. Selman *et al.*, “Planning as satisfiability.” in *ECAI*, vol. 92. Citeseer, 1992, pp. 359–363.

[9] M. Fox and D. Long, “Pddl2. 1: An extension to pddl for expressing temporal planning domains,” *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[10] ——, “Pddl+: Modeling continuous time dependent effects,” in *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, vol. 4, 2002, p. 34.

[11] G. Della Penna, D. Magazzeni, F. Mercorio, and B. Intrigila, “Upmurphi: A tool for universal planning on pddl+ problems,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 19, no. 1, 2009.

[12] D. Bryce, S. Gao, D. Musliner, and R. Goldman, “Smt-based nonlinear pddl+ planning,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[13] L. De Moura and N. Bjørner, “Satisfiability modulo theories: introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[14] E. Ábrahám and G. Kremer, “Smt solving for arithmetic theories: Theory and tool support,” in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2017, pp. 1–8.

[15] M. Cashmore, M. Fox, D. Long, and D. Magazzeni, “A compilation of the full pddl+ language into smt,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 26, no. 1, 2016.

[16] F. Leofante, E. Ábrahám, T. Niemueller, G. Lakemeyer, and A. Tacchella, “Integrated synthesis and execution of optimal plans for multi-robot systems in logistics,” *Information Systems Frontiers*, vol. 21, no. 1, pp. 87–107, 2019.

[17] A. Coles, A. Coles, M. Fox, and D. Long, “Forward-chaining partial-order planning,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 20, no. 1, 2010.

[18] ——, “Popf2: A forward-chaining partial order planner,” *The 2011 International Planning Competition*, vol. 65, 2011.

[19] C. Piacentini, M. Fox, and D. Long, “Planning with numeric timed initial fluents,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[20] S. Bernardini, M. Fox, D. Long, and C. Piacentini, “Boosting search guidance in problems with semantic attachments,” in *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017.

[21] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, S. Sorkin *et al.*, “On finding narrow passages with probabilistic roadmap planners,” in *Robotics: the algorithmic perspective: 1998 workshop on the algorithmic foundations of robotics*, 1998, pp. 141–154.

[22] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, “Incremental task and motion planning: A constraint-based approach.” in *Robotics: Science and systems*, vol. 12. Ann Arbor, MI, USA, 2016, p. 00052.

[23] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 440–448.

[24] A. Thomas, F. Mastrogiovanni, and M. Baglietto, “Mptp: Motion-planning-aware task planning for navigation in belief space,” *Robotics and Autonomous Systems*, vol. 141, p. 103786, 2021.

A

Domain and Problem Code

This appendix contains all the relevant PDDL code that was developed in the context of this thesis. Listing A.1 defines the domain described in section 3.1.1.

Listing A.1: Space assembly domain

```
(define (domain space_assembly)
    (:requirements :typing :conditional-effects :equality :fluents
     :durative-actions :numeric-fluents :negative-preconditions :action-costs)

    (:types module location robot - object)

    (:predicates (on ?m - module ?l - location)
                 (at ?r - robot ?l - location)
                 (loaded ?r - robot ?m - module)
                 (clear ?l - location)
                 (empty ?r - robot)
                 (adjacent ?l0 ?l1 - location))

```

```
  (:functions (act-cost)
              (extern)
              (triggered ?from ?to - location)
              (occupied ?loc - location)
              (unoccupied ?loc - location))

  (:durative-action load
    :parameters (?r - robot ?m - module ?robot_loc ?module_loc - location)
    :duration (= ?duration 5)
    :condition (and (at start (empty ?r))
                    (at start (at ?r ?robot_loc))
                    (at start (on ?m ?module_loc))
                    (at start (adjacent ?robot_loc ?module_loc)))
    :effect (and (at start (increase (unoccupied ?module_loc) 1))
                 (at start (not (empty ?r)))
                 (at end (loaded ?r ?m))
                 (at end (not (on ?m ?module_loc)))
                 (at end (clear ?module_loc))
                 (at end (assign (unoccupied ?module_loc) 0))
                 (at end (increase (act-cost) 1))))

  ; Unload only if there is a module on an adjacent location
  (:durative-action assemble
    :parameters (?r - robot ?m0 ?m1 - module
                ?robot_loc ?goal_module_loc ?adj_module_loc - location)
    :duration (= ?duration 10)
    :condition (and (at start (loaded ?r ?m0))
                    (at start (clear ?goal_module_loc))
                    (at start (at ?r ?robot_loc))
                    (at start (adjacent ?robot_loc ?goal_module_loc))
                    (at start (adjacent ?goal_module_loc ?adj_module_loc))
                    (at start (on ?m1 ?adj_module_loc)))
    :effect (and (at start (increase (occupied ?goal_module_loc) 1))
                (at start (not (loaded ?r ?m0)))
                (at start (not (clear ?goal_module_loc)))
                (at end (empty ?r))
                (at end (on ?m0 ?goal_module_loc))
                (at end (assign (occupied ?goal_module_loc) 0))
                (at end (increase (act-cost) 1))))

  (:durative-action move_robot
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration 20)
    :condition (and (at start (at ?r ?from))
                    (at start (clear ?to)))
```

```
      :effect (and (at start (increase (triggered ?from ?to) 1))
                   (at start (not (at ?r ?from)))
                   (at end (clear ?from))
                   (at end (at ?r ?to))
                   (at end (assign (triggered ?from ?to) 0))
                   (at end (increase (act-cost) (extern)))))

)
```