Combined Task and Motion Planning as Classical AI Planning

Jonathan Ferrer-Mestres

TESI DOCTORAL UPF / 2018

Thesis advisor

Prof. Dr. Hector Geffner,
Department of Information and Communication Technologies

Universitat Pompeu Fabra Barcelona

To my family

Acknowledgements

This thesis has been possible thanks to the help and support of many people. First of all, I would like to thank to my PhD advisor Hector Geffner. His help, guidance and patience made this thesis possible. He has inspired and encouraged me to always go one step further. Since I started as an undergraduate student, until the moment of writing this dissertation, he has always been an exceptional role model for me of what a professional researcher should be. I feel privileged to be his student.

I also want to thank to Professor Vladimir Estivill-Castro. His passion for robotics was an inspiration for me. I will always remember the first time that I saw him doing a demo with robots. At that moment, I knew what my professional career will be.

I would like to thank the people of the Artificial Intelligence and Machine Learning Group at Universitat Pompeu Fabra. Nir Lipovetzky, Miquel Ramírez, Alexandre Albore and Hector Palacios, who I met when I was an undergraduate student. They always made me feel comfortable and welcome when I started in the group as a scholar and they always inspired me. I also want to thank to Anders Jonsson, Sergio Jiménez, Dimitri Ognibene, Gergely Neu and Vicenç Gomez. We all have shared interesting and motivating moments during group seminars thanks to their talks and experiences. I also want to thank the department staff. All of them have always helped me and they had a lot of patience dealing with me.

During these almost five years, I had the luck to meet amazing and friendly people with whom we shared a lot of experiences and good moments. Pre-deadline nights, discussions, stress, frustrations, trips to conferences, programming competitions, lunches, coffees and all the things we shared together, would not be possible without the presence of my fellow PhD candidates: Javier Segovia-Aguas, Guillem Frances and Oussam Larkem.

During this time, I have met some incredible people, with whom I have established a friendship and who will always be in my heart. Filippos Kominis and Damir Lotinac. We shared a lot of feelings and experiences. I will always remember our talks, our nights before a deadline, our coffees downstairs the office, our meetings after work and all the ups and downs that we spent together. I cannot imagine these years without you. Even now, that all of us are living in different countries, you are always by my side.

I owe my gratitude to my wife Eva Adarve, for all her support during all these years. She always encouraged me and made me believe in myself. Her love and support made me stronger. We started this life stage together, and we have always walked side by side. Her strengths have always helped me. Also I want to thank my dog Caprica, who has always waiting for me to return home, happy, no matter the situation.

Finally, a truly thank goes to my parents, for all their unconditional support during my whole live. They have been always by my side, no matter the situation. They always gave me everything they had and they encouraged me to be what I want.

I can only say one thing to all these people. Thank you to all of you!

Abstract

Planning in robotics is often split into task and motion planning. The task planner decides what needs to be done, while the motion planner fills up geometric details. However, such a decomposition is not effective in general as the symbolic and geometrical components are not independent. This dissertations shows that it is possible to compile combined task and motion planning problems (CTMP) into classical planning problems; i.e., planning problems over finite and discrete state spaces with a known initial state, deterministic actions, and goal states to be reached. Motion planners and collision checkers are used for the compilation, but not at planning time. What makes our approach effective is 1) a fully compilation of CTMP problems into classical planning problems, 2) expressive classical planning languages for representing compiled problems, using functions and state constraints, 3) general planning algorithms capable of finding plans for CTMP problems using domain-independent heuristics.

Resum

La planificació en robòtica es divideix en planificació de tasques i planificació de moviments. El planificador de tasques decideix que és el que s'ha de fer, mentre el planificador de moviments s'encarrega dels detalls geomètrics. Aquesta descomposició no és efectiva, ja que els components simbòlics i geomètrics no són independents. En aquesta tesi, demostrem que és possible compilar problemes de planificació de tasques i moviments a problemes de planificació clàssica, és a dir, problemes sobre un espai d'estats finit, amb coneixement de l'estat inicial i accions deterministes. En aquesta proposta, els planificadors de moviments només s'utilitzen durant la compilació, no durant la cerca. El que fa aquesta tesi robusta és: 1) un procés de compilació de problemes de planificació de tasques i moviments a problemes de planificació clàssica, 2) uns llenguatges expressius per representar problemes compilats, utilitzant funcions i restriccions d'estats, 3) algorismes de cerca amb heurístiques independents del domini.

Preface

Classical planning is the problem of finding a sequence of actions that maps the initial state to the goal state. Classical planning assumes complete knowledge of the environment, deterministic actions and a finite state space. It belongs to what is known as task planning or symbolic reasoning. Classical planning can be seen as the problem of searching in an implicit defined direct graph, where nodes are states and edges represent actions. Modern classical planners rely on the use of heuristics search [8, 48, 45, 94], which have proven to perform well on recent planning competitions. Heuristics are usually computed from a relaxation of the original planning problem. Another class of search is width-based search, which does not compute heuristics from the problem representation, but instead performs a novelty test in every reached state. Some forms to represent planning problems in a standard form have been developed in the past years. One of them, STRIPS, is a widely used language to represent classical planning problems, which are encoded using the Planning Domain Definition language (PDDL).

Motion planning is the problem of finding a collision-free trajectory, which respects the kinematics of the robot, in a continuous space. Task planning assumes a discretized search space. However, search on a exact representation of the continuous space is not feasible, as the space, referred as configuration space, is large to scale up. For this reason, motion planning requires for methods to sample the configuration space. While these methods sacrifice the notion of completeness, motion planning algorithms based on sampling has demonstrated to perform well in problems that involve manipulation tasks. This type of planning is also referred as continuous planning or geometric reasoning.

Task planning works well for large state spaces and motion planning do also for continuous spaces. However, task planning cannot deal with a continuous representation of the space, required for robot motions, and motion planners cannot support high level goals. Combined Task and Motion Planning (CTMP) is the problem of finding a feasible sequence of actions for solving long term goals that involve motions in the continuous space. Feasible actions are actions which do not produce any collision. The problem of interleaving task and motion planning is well known in robotics literature. Although there have been several efforts for efficiently integrate task and motion planners [53, 105, 78, 41], there is not a general approach for solving CTMP problems and neither a common or standard language to represent them.

In Part I of this dissertation we introduce what is Combined Task and Motion Planning and the problems it presents, and the motivation to propose an approach that integrates task and motion planning. We review classical planning, its semantics and syntax. We then analyze the main computational approaches that we use for classical planning: heuristic search and width-based search. These computational approaches are relevant for the methods we propose. We then define Functional STRIPS, which is an expressive language which we use to show how to encode CTMP problems in a compact form. We review a second type of planning which is crucial for this dissertation: Motion planning. We define the geometric model and we review some state-of-the-art motion planning algorithms.

In Part II we introduce a novel integration of task and motion planning, where both types of planning are addressed in combination. We introduce a formulation of task and motion planning in Functional STRIPS with state constraints. This integration and model is the starting point of this dissertation. In the first chapter of this part we extend the FSTRIPS language with the ability to include state constraints, which are not a standard feature of classical planners. We show how state constraints are used to compute heuristics. We present how to model 2D CTMP problems in this expressive language. We then present two different computation approaches: The first one is a translation of CTMP problems in FSTRIPS with state constraints, to STRIPS. The second computation approach is width-based search. We then present and model *Pick-and-Place* problems in 3D. We make use of a simulated robot with a 7 degrees of freedom manipulator. We propose a way to fully compile CTMP problems in FSTRIPS with state constraints. The modeling assumes this crucial preprocessing stage. We then present a planning algorithm that computes heuristics which are not fully domain-independent, but the approach consolidate the base for a general approach for representing and solving CTMP problems.

In Part III we address the limitations of previous part and we propose a general and fully automatic preprocessing stage which assumes an input language to fully compile different types of CTMP problems. We propose a formal language to represent CTMP problems which involves objects of different shapes. The preprocessing stage computes a base and arm graph for representing the robot base and arm motions, and a set of tables encoding overlaps constraints and grasping poses, for fast lookups. These data structures are used during planning time and avoid the external calls to motion planners and collision checkers, which are only used during the preprocessing stage. We then show how we model different types of problems that involve symbolic and geometric reasoning. These problems cannot be solved using only one type of planning. Finally, we present a general algorithm which is domain-independent and makes use of state constraints to compute useful heuristics. For this, two different sets of atoms R and C are computed on a $IW(k)$ preprocessing while relaxing the state constraints. The set R contains the relevant atoms obtained from the initial state to each goal. The set C contains the no good atoms, which are the atoms that violate a state constraint.

Overview of Dissertation

Chapter 1 introduces what is combined task and motion planning and the motivation to develop an efficient and general approach.

Chapter 2 introduces classical planning and the Functional STRIPS planning language. We also review some state-of-the-art computational approaches.

Chapter 3 introduces motion planning as planning in continuous space. It presents the problem of combining task and motion planning for robotics manipulation.

Chapter 4 extends the FSTRIPS language with the ability to handle state constraints to avoid collisions. It shows how to model CTMP problems in FSTRIPS and a translation to STRIPS.

Chapter 5 presents a compilation of CTMP problems into FSTRIPS with state constraints and a general width-based search algorithm capable of finding plans over a huge combinatorial state space by computing weak heuristics.

Chapter 6 introduces a formal language to fully and automatically compile CTMP problems into classical planning problems. A general algorithm which compute domain-independent heuristics for handling state constraints is presented and results evaluated on a number of experiments.

Chapter 7 describes the planner, the whole architecture and the different modules used of the shelf.

Chapter 8 presents a summary of the contributions of this dissertation and discuss the current and future work in CTMP.

The different approaches and results presented in this dissertation appear in the following articles:

- Ferrer-Mestres, Jonathan, Guillem Frances, and Hector Geffner. *"Planning with state constraints and its application to combined task and motion planning."* Proc. of Workshop on Planning and Robotics (PLANROB) 2015 [28]. [Chapter 4].

- Ferrer-Mestres, Jonathan, Guillem Francès, and Hector Geffner. *"Combined Task and Motion Planning as Classical AI Planning."* arXiv preprint arXiv:1706.06927 (2017) [29] [Chapter 5]

- Ferrer-Mestres, Jonathan and Hector Geffner. *"General Architecture and Use Cases for Combined Task and Motion Planning"*. In preparation. [Chapter 6]

Contents

List of Figures

List of Tables

PART I

Background

CHAPTER 1

Introduction

1.1 Combined Task and Motion Planning

Combined Task and Motion Planning (CTMP) is a well known problem in robotics. CTMP problems involve robots that can move and rotate, objects that have geometrical dimensions and constraints for avoiding collisions. These problems are thought to be outside the scope of standard AI planners and are normally addressed through a combination of two types of planners: task planners that handle the high-level, symbolic reasoning part, and motion planners that handle motion and geometrical constraints [64, 105, 65, 78, 99].

Task planning performs well for large state spaces but it cannot deal with a continuous representation of the space, which is required for robot motions. On the other hand, motion planning performs well for continuous spaces, but it cannot support high level goals. Hence, an integration of task and motion planning is required for solving these types of problems. Combined Task and Motion Planning (CTMP) is the problem of finding a feasible sequence of actions for solving long term goals that involve motions in the continuous space. Feasible actions are actions which do not produce any collision.

Consider an autonomous robot that can move around and manipulate objects located on top of a table. In order to achieve high level tasks like picking up objects and placing them on top of another table, the robot must be able to interleave high level planning for long term goals and motion planning to compute arm and base motions and to check for motion feasibility. Motion planners work well for large continuous spaces and task planners do also for large discretized state spaces.

However, task planning cannot deal with geometric details and motion planners cannot support symbolic tasks. For this reason, we require to combine both types of planning for solving large manipulation high level tasks. Task planning can decide where each object must be placed, while motion planning decides how to approach to these objects and how to grasp them based on their geometric details. The symbolic and geometrical components, however, are not independent. Hence, by giving one of these two parts the secondary role of verifying feasibility and filling up the geometric details, approaches based on task and motion decomposition tend to be ineffective, resulting in lots of backtracking. The problem of excessive backtracks is well known in constraint satisfaction when constraints are used *passively* to prune the search for solutions [79, 18].

1.2 Example

Let's suppose the problem of Fig. 1.1. There are two tables: one with three different boxes on top of it and the second table is empty. There is a robot which can move its base and arms. The goal is to place the blue box on top of the empty table. However, the red and yellow boxes are obstructing the path to the blue box and they have to be moved out of the way before grasping the blue box. If we rely on a classical planning representation, we can have high-level actions like *move*, *pick-up* and *place*. The task planner reasons at the symbolic level, so it cannot deal with the geometric representation of this problem. A possible solution given by a task planner could be: 1) *move* to a discretized position close to the table where the boxes are, 2) *pick-up* the blue box, 3) *move* to a position close to the left-most table, 4) *place* the blue box on the left table. However, when the robot tries to pick-up the blue box, the motion planner tries to compute a collision-free trajectory, which is not possible since the yellow box is obstructing the path to the blue box. After the task planner has been notified, it can replan to move the yellow box out of the way. However, the robot cannot pick-up the blue box yet, as the motion planner will find again a collision, this time with the red box. The fact of using the motion planner to verify feasibility causes lots of backtracking. On the other hand, a motion planner cannot deal with high-level goals like *place* the blue box on the left-most table. Finding a solution for this problem requires for a combination of task and motion planning. A possible and good solution could be:

- *move* to a position close to the table containing the boxes.
- *pick-up* the yellow box.
- *place* the yellow box in another no-obstructing location.

- *pick-up* the red box.
- *place* the red box in another no-obstructing location.
- *pick-up* the blue box.
- *move* to the left-most table.
- *place* the blue box in the left-most table.

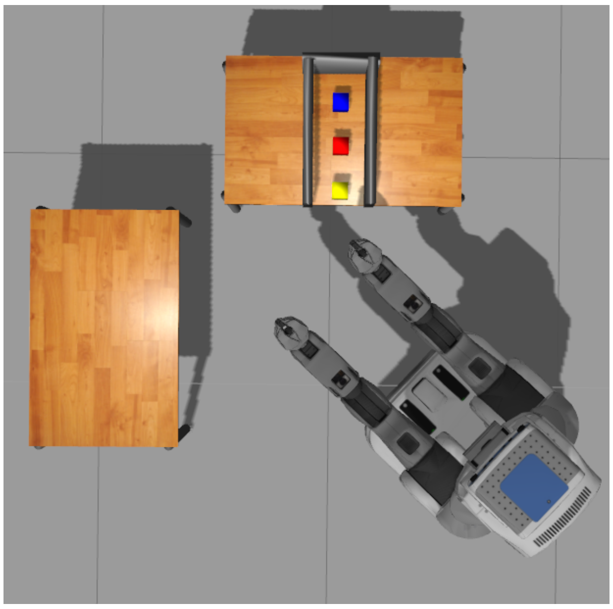


Figure 1.1: An instance of a problem which requires task and motion integration. The goal is to place the blue box in the left-most empty table.

1.3 Motivation

The problem of interleaving task and motion planning is well known in robotics literature. Although there have been several efforts for efficiently integrate task and motion planners [11, 105, 78, 41], there is not a general approach for solving CTMP problems and neither a common or standard language to represent them. An efficient combination is crucial for problems which require symbolic and geometric reasoning.

In this dissertation we propose a novel approach where CTMP problems are fully compiled into classical planning problems and state constraints are used to prevent collisions. We define a formal input language to describe CTMP problems. Moreover, we avoid the use of calls to motion planners and collision checkers during planning time and we propose a flexible planning algorithm which computes domain-independent heuristics to deal with state constraints.

1.4 Thesis Outline

In chapter 1 of this thesis we have already explained the problem of combining task and motion planning. We presented an example which requires a robust integration of both types of planning in order to avoid backtracking. We have introduced our motivation to develop a novel and general method to efficiently combine task and motion planning by, given an input language, compile CTMP problems into classical planning problems with state constraints.

In chapter 2 we review classical planning and planning problems in STRIPS and FSTRIPS. We also mention some of the computational approaches for classical planning: heuristic search and width-based search algorithms, which are extensively used in this work.

Chapter 3 defines what is motion planning. We review different methods used for planning in continuous space, focusing on sampling-based motion planning. We describe some sampling methods. We then explain some state-of-the-art motion planning algorithms, such as Rapidly Exploring Random Tree (RRT) and Probabilistic Roadmap (PRM), and we show some examples of motion planning for solving geometric problems.

Chapter 4 introduces a Functional STRIPS model with state constraints to avoid collisions. We also introduce heuristics to handle state constraints and we show how the computation of heuristics is affected by the introduction of such constraints. Further, we demonstrate the modeling of 2D CTMP problems with FSTRIPS and state constraints. We introduce two computational approaches for solving these CTMP problems. The first approach is a translation of a CTMP problem with state constraints from FSTRIPS to STRIPS and we use state-of-the-art heuristic search to finding plans. The second computational approach does not use heuristics at all, but width-based search algorithms based on a novelty test. The experiments presented in chapter 4 show a robot arm in a 2D world and are far from trivial.

In Chapter 5 we scale up to more challenging problems, involving *Pick-and-Place* tasks in a 3D world. We present a compilation which is independent of the number of objects and allows us to compute suitable tables for fast lookups during search process. Computing such tables avoids expensive calls to motion planners and collision checkers, which are only used at compilation time and not during planning time. We then present the model of pick-up and place CTMP problems in FSTRIPS using state constraints to avoid collisions, and we present an approach which uses such state constraints to compute heuristics.

In Chapter 6 we develop a general approach for combined task and motion planning, which makes use of recent advances of modern classical AI planning algorithms. We show how to model a number of CTMP problems in FSTRIPS, moving away from the classical *Pick-and-Place* domain, and we present two new problems which require symbolic and geometric reasoning: *Blocks World* and *Structures Building*. Finally, the main contributions of this chapter are 1) the definition of an input formal language, 2) a fully and automatically compilation of task and motion planning problems to classical planning problems and 3) a flexible and general planning algorithm which is domain-independent and computes heuristics from the FSTRIPS and state constraints encoding.

In Chapter 7 we present the planner that we have developed which combines task and motion planning, the pipeline of our architecture, the modules used off the shelf and the implementation details.

Finally, in Chapter 8 we present the conclusions along with the main contributions of this work. Finally, we review ongoing work and future possibilities.

CHAPTER 2

Classical Planning

2.1 Introduction

Classical planning is the problem of finding a plan, which is a squence of deterministic actions that when applied, maps an initial state into a goal state. The environment is only affected by the actions performed by a single agent, which has full knowledge of the current state of the environment. The challenge of classical planning is computing such plans efficiently. Classical AI planners are currently able to solve problems over large state spaces. State-of-the-art methods rely on the use of heuristics, which are derived automatically in order to guide a state-space search or on translations into propositional satisfiability. [96, 40, 38]. Classical planning techniques have proven to be efficient, as we can see in the International Planning Competition, as well as in translation approaches, where a problem of a different planning model is translated to a classical planning problem that allows the use of approaches developed for classical planning. For example, translations from conformant planning to classical planning [86, 87], contingent planning to classical planning [1] or temporal planning to classical planning [52].

2.2 Model

A classical planning problem can be seen as a search problem in a directed graph where nodes represent states and edges between nodes are actions. A plan is a path from a node representing the initial state, to a node representing a goal state. The classical planning model can be described as follows:

Definition 2.2.1 *(Classical Planning Model). The Classical Planning Model is a tuple* $\Pi = \langle S, s_0, S_G, A, f, c\rangle$ *where:*

- S is a finite and discrete state space
- $s_0 \in S$ is the initial know state
- $S_G \subseteq S$ is the set of goal states
- $A(s) \subseteq A$ is the set of actions A that are applicable in each state $s \in S$
- $f(s, a)$ is a deterministic transition function where applying an action a in a state s results in state s'
- $c(s, a)$ is the positive cost of applying action a in state s

A plan π for a classical planning model, is a sequence of actions $a_0, a_1, \dots, a_n$, such that, when applied in the initial state s_0, produces a sequence of states $s_0, s_1, \dots, s_{n+1}$ such that a_i is an action applicable in state s_i and s_{n+1} is a goal state.

In this work we assume that costs are uniform. Meaning, the cost of applying an action a to some state s is always the same. The total cost of a plan is the sum of action costs

$$c(\pi) = \sum_{i=1}^{n} c(a_i).$$

Thus, the cost is the plan length $|\pi|$. A plan is said to be *optimal* (π^*) if the total cost is the minimum, among all possible plans. In other words, a plan π is optimal if there is not shorter plan than π.

2.3 Syntax

The most extensively used language to describe classical planning models is STRIPS[30], which consists of boolean variables called fluents and atoms. The states are described through these variables, which domain is $\{$*true,false*$\}$. Then, a state is defined by the conjunction of the truth values of all the variables in that state.

Definition 2.3.1 *A classical planning Problem in STRIPS is a tuple* $P = \langle F, O, I, G\rangle$ *where:*

- F is the set of all boolean variables or atoms

- O is the set of all actions or operators
- $I \subseteq F$ is the initial situation
- $G \subseteq F$ is the goal situation

Every action $a \in O$ is represented by three lists:

- A *Precondition* list $Prec(a) \subseteq F$. It represents the set of atoms that must be true in order to apply the action a.
- An *Add* list $Add(a) \subseteq F$. It represents the set of atoms that will become true after applying the action a.
- A *Delete* list $Del(a) \subseteq F$. It represents the set of atoms that will become false after applying the action a.

2.4 Semantics

Given a classical planning problem $P = \langle F, O, I, G \rangle$, the corresponding state model is $S(P) = \langle S, s_0, S_G, A, f, c \rangle$ consists of:

- Each state $s \in S$ is a subset of atoms from F
- The initial state s_0 is I
- The goal states s are such that $G \subseteq s$
- The actions a applicable in s, $A(s)$ are operators in O such that $A(s) = o|Prec(o) \subseteq s$.
- The transition function $f(s,a)$ results in new state $s' = (s \bigcup Add(a)) \backslash Del(a)$.
- Action costs are always uniform. The total cost of a plan π is the sum of the cost of all applied actions.

2.5 Example

Consider a problem in which an $N \times N$ grid, with $N = 3$, contains a robot. The robot R is at some position p_i. There is a box B at position p_j with $i \neq j$. The goal is for the box B to be at position p_k, where $i \neq j \neq k$. Assume that initially the robot is at position p_1 and the box is at position p_5, and the goal is to have the box at position p_9. The Figure 2.1 illustrates the problem. We can model this problem as a classical planning Problem using STRIPS as $P = \langle F, O, I, G \rangle$:

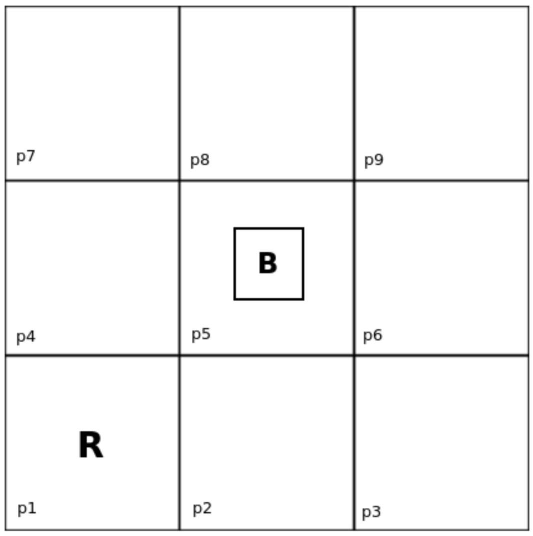


Figure 2.1: A simple classical planning problem representation, where a robot **R** located in position p_1 has to pick-up a box **B** , which is position p_5, and place it in position p_9.

- Fluents F: {*at_R*(p_i), *at_B*(p_i), *holding*, *adj*(p_i, p_j)}, for any i, j in $[1, N \times n]$, with *at_R*(p_i), *at_B*(p_i) denoting the position of the robot and the box respectively, *holding* denoting whether the robot has the box or not, and *adj*(p_i, p_j), denoting if a connection (adjacency) exists between two any positions.

- Initial I: *at_R*(p_1), *at_B*(p_5), *adj*(p_1, p_2), *adj*(p_2, p_3), *adj*(p_1, p_4), *adj*(p_2, p_5), *adj*(p_3, p_6), *adj*(p_4, p_5), *adj*(p_5, p_6), *adj*(p_4, p_7), *adj*(p_5, p_8), *adj*(p_6, p_9), *adj*(p_7, p_8), *adj*(p_8, p_9), *adj*(p_2, p_1), *adj*(p_3, p_2), *adj*(p_4, p_1), *adj*(p_5, p_2), *adj*(p_6, p_3), *adj*(p_5, p_4), *adj*(p_6, p_5), *adj*(p_7, p_4), *adj*(p_8, p_5), *adj*(p_9, p_6), *adj*(p_8, p_7), *adj*(p_9, p_8).

- Goal G: *at_B*(p_9).

- Operators O:

  - *move*(p_i, p_j) for any $i, j \in [1, N \times N]$.
    - *Precondition(o): at_R*(p_i), *adj*(p_i, p_j)
    - *Add(o): at_R*(p_j)
    - *Delete(o): at_R*(p_i)
  - *pick-up*(p_i) for any $i \in [1, N \times N]$.
    - *Precondition(o): at_R*(p_i), *at_B*(p_i)
    - *Add(o): holding*

        - *Delete(o): at_B(p_i)*
    - *place(p_i)* for any $i \in [1, N \times N]$.
        - *Precondition(o): at_R(p_i), holding)*
        - *Add(o): at_B(p_i)*
        - *Delete(o): holding*

A possible plan π for this problem is the following:

$$\pi = \{move(p_1, p_2), move(p_2, p_5), pick\_up(p_5), move(p_5, p_8), move(p_8, p_9), place(p_9)\}$$

In this example, it does not exist a shorter plan than π. Then, we say that the plan is optimal $\pi*$. The total cost of this plan is given by the number of applied actions, which is the plan length $|\pi| = 6$.

2.6 Planning Domain Definition Language

Classical Planning Problems in STRIPS, are usually encoded using the Planning Domain Definition Language (PDDL) [81]. The planning community has standardized planning with the introduction of PDDL and its extensions [31]. The PDDL support for the International Planning Competition (IPC) allowed for benchmark planning problems, proposed to be addressed by the planning community. The original fully deterministic planning challenges, have been complemented by planning and learning domains, as well as uncertainty challenges (where the effects of the actions are uncertain and usually modeled by some probability distribution of the outcomes).

For the purpose of this work, it is sufficient to show a PDDL encoding in STRIPS for a fully deterministic problem. Figure 2.2 shows the encoding in PDDL for the problem domain defined in the previous section, while figure 2.3 shows the encoding of a problem instance. PDDL encodings are usually defined in two separate files, called *domain* and *instance*.

2.7 Complexity

Determining if there exists a valid plan π for an arbitrary problem instance, in the propositional STRIPS planning model is decidable and has been shown to be PSPACE-complete [9]. This is the class of problems that can be solved using memory which is polynomial to the input size and has no restrictions in the

```
(define (domain Pick-up-Place)
  (:requirements :strips :typing)
  (:types position)
  (:predicates (at_R ?p - position) (at_B ?p - position)
               (adj ?p1 ?p2 - position) (holding)))

(:action Move
  :parameters (?p1 ?p2 - position)
  :precondition (and (at_R ?p1) (adj ?p1 ?p2))
  :effect (and(not(at_R ?p1) (at_R ?p2)))

(:action Pick-Up
  :parameters (?p - position)
  :precondition (and(at_R ?p)(at_B ?p))
  :effect (and(not(at_B ?p) (holding)))

(:action Place
  :parameters (?p - position)
  :precondition (and(at_R ?p)(holding))
  :effect (and(not(holding) (at_B ?p)))
)
```

Figure 2.2: PDDL domain encoding for illustrating a Classical Planning Problem in STRIPS.

amount of time. Other case, is undecidable for infinite state spaces. In the case of planning with costs, the problem of finding an optimal plan $\pi*$ for an arbitrary problem instance is also PSPACE-complete. These classes of problems can be solved in polynomial space with no restrictions on running time. As in the worst-case planning problems are intractable, planning approaches are measured in terms of performance regarding time and space, on a set of benchmarks.

2.8 Computational Approaches for Classical Planning

Classical AI planners are currently able to solve problems over large state spaces. In classical planning, the initial state is fully known and actions have deterministic effects. State-of-the-art methods rely on the use of heuristics that are derived automatically in order to guide a state-space search or on translations into propositional satisfiability.

```
(define (problem pickup-place)
(:domain Pick-up-Place)
(:objects p1 p2 p3 p4 p5 p6 p7 p8 p9 - position)
(:init (at_R p1) (at_B p5) (adj p1 p2) (adj p2 p3) (adj p1 p4)
       (adj p2 p5) (adj p3 p6) (adj p4 p5) (adj p5 p6) (adj p4 p7)
       (adj p5 p8) (adj p6 p9) (adj p7 p8) (adj p8 p9) (adj p2 p1)
       (adj p3 p2) (adj p4 p1) (adj p5 p2) (adj p6 p3) (adj p5 p4)
       (adj p6 p5) (adj p7 p4) (adj p8 p5) (adj p9 p6) (adj p8 p7)
       (adj p9 p8))
(:goal (at_B p9)))
```

Figure 2.3: PDDL instance encoding of the robot domain.

Given a classical planning problem P and its state model $S(P)$, where in the associate graph nodes represent states and edges represent actions between nodes, a graph-search algorithm can be used to find a plan from an initial state to a goal state. Although any graph-search algorithm could be used among these graphs, blind-search algorithms like *Dijkstra* [19], *Depth First Search* (DFS) or *Breath First Search* (BrFS) do not perform well due the size of the state space of planning problems. On the other hand, there are more effective methods who have been proven to be successful:

2.8.1 Classical Planning as Heuristic Search

Heuristic search algorithms are extensively used in state-of-the-art planners and have demonstrated good results in classical planning benchmarks. A *heuristic* function is an approximation of the solution cost, which is used to guide the search, and is derived directly from the problem representation. One example is a navigation problem. which could be seen as a graph, where nodes represent cities and edges represent a path between cities. A heuristic estimator is used on every reachable state $s \in S$, where a possible heuristic function h could be the straight distance between cities. One search algorithm which relies on a heuristic estimator is $A*$, where node expansions depend on an evaluation function: $f(n) = g(n)+h(n)$, where $g(n)$ is the total accumulated cost from the initial node and $h(n)$ is the output of the heuristic function; Greedy Best First Search (GBFS), where node expansion only depends on the heuristic function $f(n) = h(n)$. Some algorithms like $A*$ are optimal, in the sense that an optimal plan $\pi*$ is guaranteed, only when the heuristic is admissible. A heuristic h is admissible when $h \leq h*$, where $h*$ is the optimal heuristic. The optimal heuristic $h*$ is the exact cost from a state s to a goal state. However, depending on the heuristic function, $A*$ may not feasible for large state space problems. Then, heuristics is not only related to find optimal solutions, but computing informative heuristics, together with heuristic search algorithms have been demonstrated to find optimal and suboptimal solutions for problems with a large state space.

The recent International Planning Competition (IPC) have been dominated by the planners which rely on heuristics such as: `HSP` [8], `FF` [48], `Fast-Downward` [45] and `LAMA` [94]. Among with different versions and improvements of these citet planners [46, 21]. These heuristics are automatically extracted from the problem representation [8, 82], by using different relaxations on the original problem. These heuristics are domain-independent. A relaxation is a simplified form of an original problem. For example, the `FF`planner computes a delete relaxation, where negative effects of actions are not taken into consideration, and it computes a heuristic based on the length of the relaxed plan called h_{FF}. Other heuristics which come from the relaxation of a problem are the well known additive heuristic h_{add} and the maximum heuristic h_{max} [8, 20]. Another well known heuristic planner is `LAMA`, which use landmark heuristics. A landmark is an atom p that is made true by all plans. Then, landmark heuristic just counts the number of landmarks that have not been achieved yet in some state s. Landmarks have been used in several approaches [94, 55].

2.8.2 Classical Planning as SAT

Planning as Satisfiability [56] determines whether there is a truth assignment that satisfies a propositional logical formula or a set of clauses obtained from a classical planning problem translation. In other words, it determines if a Conjunctive Normal Form (CNF) formula is satisfiable or not. If the planning problem is unsolvable, the SAT formula will be unsatisfiable. The logical formula captures the representation of the planning problem like the initial state, goal state and actions. SAT solvers take a set of clauses and a horizon N, starting from $N = 0$. The algorithm proceeds by generating appropriate logical values assignments and increasing N until a plan is found. Given a STRIPS problem P and a horizon N, a CNF formula which include propositions for each atom and actions, is satisfable if there is a plan that solves problem P in at most N steps. SAT problem has been shown to be NP-complete[97], so there is no polynomial time algorithm known yet, but exponential worse-case algorithms. However, SAT solvers manage to solve problems with a huge amount of variables and clauses.

2.8.3 Width-based Search

While heuristic based search relies on extracting heuristics automatically from the problem representation, another approach is based on the structure of the problem.

Pure width-based search algorithms [72, 73] are exploration algorithms and do not rely on goal directed heuristics. The simplest such algorithm is Iterated Width or IW, which is a sequence of calls $IW(i)$ for $i = 1, 2, \ldots$ where $IW(i)$ is a plain breadth-first search with one change: the states generated in the search are pruned when they are not the first state in the search to make true some tuple (set) of i atoms or less. Thus, $IW(i)$ for $i = 1$, i.e., $IW(1)$, is a breadth-first search where a newly generated state s is pruned when there is no new atom $X = x$ made true by s, $IW(2)$ is a breadth-first search where a newly generated state s is pruned when there are no atoms $X = x$ and $Y = y$ such that the *pair* of atoms $\langle X = x, Y = y \rangle$ is true in s and false in all the states generated before s, and so on. More generally, $IW(k)$ is a breath-first search where newly generated states are pruned if they don't pass the novelty test, meaning, if its novelty is greater than k. The novelty of a state s is the size of the smallest tuple of atoms t that is true in s and false in all previously generated states s'.
A key property of the algorithm is that while the number of states is *exponential* in the number of atoms, IW(i) runs in time that is exponential in i only. In particular, IW(1) is linear in the number of atoms, while IW(2) is quadratic. Furthermore, the work seen in [72] define a general *width* measure for classical problems P and prove that IW(i) solves P when the width of P is no greater than i. Moreover in such a case, IW(i) solves P optimally (i.e., it finds a shortest solution). As a result, all such problems can be solved in quadratic time by IW(2) although the number of states is exponential.

$IW(k)$ has proven to perform really well in instances of many of the standard benchmark domains. In most of these instances the algorithm solved them in (low) polynomial time, provided that the goal is atomic. IW is a complete algorithm but it does not perform well with multiple goal atoms [72, 73]. For this reason, another algorithm called *Serialized IW* (SIW) has been developed. SIW achieves a goal atom one at a time, by doing multiple calls to *IW* algorithm. However, it is an incomplete algorithm that can get trapped into dead-ends.

On the other hand, Best-first width search (BFWS) is a best-first search algorithm that combines width-based measures with an implicit form of goal serialization. BFWS tracks two measures: the number of atomic goals that are true in s, and the size of the smallest tuple of atoms t that is true in s and false in all the states generated before s that have the same number of true goals as s. The evaluation function $f(s)$ in BFWS is given by the second measure, called the novelty measure (smaller is better), with ties broken by favoring states with a maximum number of true goals.

Width based search algorithms have been used more recently for playing Atari video games [4], and video games in the General Video Game AI Competition [88], achieving in both cases state-of-the-art results [76, 39].

2.9 Functional STRIPS

Functional STRIPS (FSTRIPS) is a simple but expressive classical planning language based on the variable-free fragment of first-order-logic, where action a have preconditions $Pre(a)$, where $Pre(a)$, as well as goals G, can be variable-free, first-order formulas, and $f(t)$ and t' are terms with f being a fluent symbol. FSTRIPS involves *constant*, *function* and *relational or predicate symbols* but no variable symbols. We review it following [37].

2.9.1 Syntax

Definition 2.9.1 *(FSTRIPS Planning Problem) A problem in FSTRIPS is a tuple* $P = \langle S, I, O, G \rangle$ *where:*

- S is the set of non-standard symbols (fixed and fluent) and their types
- I provides the initial unique denotation s_0 of such symbols
- O stands for the actions
- G is the goal

A plan for a problem $P = \langle F, I, O, G \rangle$ is a sequence of applicable actions from O that maps the unique initial state where I is true into one of the states where G is true.

Functional STRIPS assumes that *fluent* symbols, whose denotation may change as a result of the actions, are all *function* symbols. Constant, functional and relational (predicate) symbols whose denotation does not change are called *fixed* symbols, and its denotation must be given either extensionally by enumeration, or intentionally by means of procedures as in [23]. Among them, there is usually a finite set of object names, and constant, function, and relational symbols such as '3', '$+$' and '$=$', with the standard interpretation.

For example, typical Blocks world atoms like $on(a, b)$ can be encoded in FSTRIPS as $on(a, b) = true$, by making on a functional symbol, or in this case, more conveniently, as $loc(a) = b$ where loc is a function symbol denoting the block location.

Terms, atoms, and formulas are defined from constant, function, and relational symbols in the standard way, except that in order for the representation of states to be finite and compact, the symbols, and hence the terms are typed. A *type* is given by a finite set of fixed constant symbols. The terms $f(t)$ where f is a fluent symbol and t is a tuple of fixed constant symbols are called *state variables*, as the state is actually determined by the value of such "variables".

2.9.2 Semantics

From a semantic point of view, states represent logical interpretations over the language of FSTRIPS. The denotation of a symbol or term t in the state s is written as t^s, while the denotation r^s of terms made up only of fixed symbol, and which does not depend on the state, is written as r^*. The denotation of standard fixed symbols like '3', '+', '=' is assumed to be given by the underlying programming language, while object names c are assumed to denote themselves so that $c^* = c$. The denotation of fixed (typed) function and relational symbols can be provided extensionally, by enumeration in the initial situation, or intensionally, by attaching actual functions (i.e. external procedures) to them [23]. The states s thus just need to encode the denotation f^s of the functional fluent symbols, which as the types of their arguments are all finite, can be represented as the value $[f(c)]^s$ of a finite set of state variables $f(c)$, where f is a functional fluent and c is a tuple of fixed constant symbols. The denotation $[f(t)]^s$ of a term $f(t)$ for an arbitrary tuple of terms t, is then given by the value $[f(c)]^s$ of the state variable $f(c)$ where $c^* = t^s$. The denotation e^s of all terms, atoms, and formulas e in the state s follows in the standard way.

An action a is applicable in a state s if $[Pre(a)]^s = true$, and the state s_a that results from the action a in s satisfies the equation $f^{s_a}(t^s) = w^s$ for all the updates $f(t) := w$ that the action a triggers in s, and otherwise is equal to s. This means that the action a changes the value of the *state variable* $f(c)$ to w^s in the state s iff there is an effect $C \rightarrow f(t) := w$ of action a such that $C^s = true$ and $t^s = c$. For example, if $X = 2$ is true in s, the update $X := X + 1$ increases the value of X to 3 without affecting other state variables. Similarly, if $loc(b) = b'$ is true in s, the update $clear(loc(b)) := true$ is equivalent to $clear(b') := true$.

2.9.3 Example

A simple planning problem involving a set of integer variables X_1, …, X_n and actions that allow us to increase or decrease the value of any variable by one within the $[0, n]$ interval, can be modeled in Functional STRIPS by treating the variables X_i as 0-arity fluent functional symbols with values ranging in the $[0, n]$ interval. These X_i symbols represent the state variables in the problem. If $I = \{X_1 = 0, \ldots, X_n = 0\}$, and $G = \{X_1 < X_2, \ldots, X_{n-1} < X_n\}$, the problem is about changing the value of the X_i variables from 0 to final values that increase monotonically with i. The precondition-free action $increment(X_i)$ has the effect $X_i := \min(X_i + 1, n)$, and $decrement(X_i)$ has the effect $X_i := \max(X_i - 1, 0)$

Another example is the well known Blocks world domain. The action of moving a block b onto another block b' can be expressed by an action $stack(b, b')$ with precondition $clear(b) = true \wedge clear(b') = true$, and effects $loc(b) := b'$ and $clear(loc(b)) := true$. In this case, the terms $clear(b)$ and $loc(b)$ for blocks b stand for state variables; the term $clear(loc(b))$ is not a state variable, as $loc(b)$ is not a fixed constant symbol. The goal of placing block b_1 on top of block b_2 is expressed as $loc(b_1) = b_2$, while more interestingly, the goal of building a tower, any tower, with all the blocks $b_1, \ldots, b_n$ can be expressed with the conjunction of atoms $loc(b_i) \neq loc(b_k)$ for every pair of blocks b_i and b_k, $i \neq k$. A model fragment of Blocks World domain can be seen in Fig. 2.4 and an example of an instance is shown in Fig. 2.5.

2.10 Summary

A classical planning problem can be seen as a search over a directed graph where nodes represent states and edges represent actions between states. A plan is a path in the graph that connects the initial state with a goal state. Classical planning belongs to what is known as task planning, which represent the symbolic reasoning of an agent along a finite state space. There are a number of graph search algorithms. Blind Search algorithms have been proven to not perform well on large state problems, while heuristic search algorithms have demonstrated a good performance on classical planning benchmarks. Deriving heuristics from the problem representation is a crucial process to guide search, not only for optimal plans, but for suboptimal and satisfactory solutions. There are several well known classical planners that compute heuristics with different approaches. Delete relaxation or landmark heuristics are an example of what state-of-the-art classical planners use to solve challenging benchmarks. Heuristics search compute heuristic from the problem representation, another class of algorithms, called width-based search algorithms, like $IW(k)$ and $SIW(k)$, compute a type of pseudo-heuristics from the problem structure. Width-based search algorithms have a relevant importance in this thesis, as they have demonstrated to produce good and quality results on very challenging problems.

For representing classical planning Problems, we have mentioned two languages. STRIPS language is the most extended one in the planning community. It has been used in the International Planning Competition to represent well known benchmarks. On the other hand, we propose the use of another language, called FSTRIPS, which is more expressive and it is based on the variable-free fragment of first order logic. We have defined and example in PDDL of how to represent a simple problem in both STRIPS and FSTRIPS. The next chapter of this thesis, introduces another well known type of planning used in the robotics community, called motion planning.

```
(define (domain blocksworld-fn)
  (:types place - object
          block - place)
  (:constants table - place)
  (:predicates (clear ?b - place))
  (:functions  (loc ?b - block) - place)

(:action stack-to-block
  :parameters (?b - block ?from - place ?to - block)
  :precondition (and (clear ?b)
                     (clear ?to)
                     (= (loc ?b) ?from)
                     (not (= ?b ?to))
                     (not (= ?b ?from))
                     (not (= ?from ?to)))
  :effect (and (assign (loc ?b) ?to)
               (clear ?from)
               (not (clear ?to)))

(:action stack-to-table
  :parameters (?b ?from - block)
  :precondition (and (clear ?b)
                (not (= ?b ?from))
                (= (loc ?b) ?from))
  :effect (and (assign (loc ?b) table)
               (clear ?from))
```

Figure 2.4: PDDL model fragment for illustrating Blocks world classical planning Problem in FSTRIPS.

```
(define (problem two-blocks)
  (:domain blocksworld-fn)
  (:objects b1 b2 - block)
  (:init (clear b2) (= (loc b1) table)
         (= (loc b2) b1) (clear table) )
  (:goal (= (loc b1) b2))
)
```

Figure 2.5: PDDL instance encoding for illustrating the Blocks world classical planning Problem in FSTRIPS.

CHAPTER 3

Motion Planning

3.1 Introduction

Motion Planning is widely used in robotics community [66, 58, 57]. A robot navigation problem involves a robot moving in an environment with possible presence of obstacles. But finding a collision free trajectory between an initial configuration q_I and a goal configuration q_G does not only involve planar navigation. Manipulating objects with a robot arm, with a number of degrees of freedom, requires efficient motion plans. For instance, a manipulation problem with a robot arm with 7 degrees of freedom (7-DOF), that has to interact in a large continuous configuration space to pick-up and place objects. Problems that involve long term tasks, such as picking up objects, tidying-up a room, or the classical planning problem Blocks world, requires both task and motion planning.

In chapter 2 we discussed the problem of classical planning, which falls into category of task planning or symbolic reasoning. Task planning answers the question of, *what should be done?*. While motion planning, also known as continuous planning or geometric reasoning answers the question of, *how should be done?* As represented in Fig. 1.1, the robot can compute a trajectory for how to pick up the box, but not what to do with that box. Task planner can be used to specify symbolic goals such as *place box blue on top of table 2*. To accomplish this, we rely on classical planning. However, to compute a collision-free trajectory for a robot manipulator we require for motion planning. In this chapter we define what is motion planning, we discuss some of the methods to sample the continuous space and we review some motion planning algorithms.

3.2 Planning in Continuous Space: Motion Planning

Motion planning determines what motions a robot has to do in order to reach a goal configuration state without colliding into obstacles, and respecting the properties of the robot itself. Motion planning is also referred to as planning in continuous space, where the state space is referred as the configuration space.

One of the best known methods for addressing the motion planning problem is *sampled-based motion planning*. This method avoids the construction of the entire configuration space and instead, it relies on defining a set of samples along the space. It has been successful in recent years for robotics manipulation. There are several techniques for sampling which will be reviewed in following sections. Sampling a configuration space requires the use of collision detector, which is considered as a black box. Sampling based methods sacrifice completeness by the notion of probabilistic completeness. This will be covered more extensively in the corresponding section, where we will describe the state-of -the-art sampling-based motion planning algorithms.

Another method is known as *combinatorial motion planning*, which leads us to completeness, building and exactly and discrete representation of the configuration space. This means that for any problem instance a combinatorial motion planning algorithm will either find a solution or will report that no solution exists. However, sampling-based approach is more used in the robotics motion planning research, due to the complexity and difficulties of the combinatorial motion planning. One of the reasons of using combinatorial motion planning is solving specific planning problems. For example a simple 2D world with a robot that can only perform simple actions.

Another current approach is the *search-based planning* [12, 13], where they use heuristic search-based algorithms based on sets of motion primitives that have been pre-defined, and the search is done over a constructed graph. This method has been demonstrated in mobile manipulation problems using 7-DOF robots. It is a complete approach but not necessary optimal. One of the advantages of this approach, is the fact that they can use the same motion primitives in similar cases, where, on the other hand, in the sampling-based motion algorithms, different outputs can be obtained from the same, or similar input. However, in high dimensional spaces, the search can become slow.

3.2.1 Geometric Model

In this section we define each element that compose a geometric model. We use the notation defined by [68].

Let's define the *world* as $\mathcal{W}$, either in 2D, in which $\mathcal{W} = \mathbb{R}^2$ or 3D world, in which $\mathcal{W} = \mathbb{R}^3$.

The obstacle region $\mathcal{O}$ is a static portion of the world defined as the set of all points in $\mathcal{W}$ that lie in one or more obstacles: $\mathcal{O} \subseteq \mathcal{W}$. There are several ways to represent $\mathcal{O}$. For example as polygonal models, semi-algebraic models, 3D triangles meshes or bitmaps represented as occupancy grids. For example an obstacle region $\mathcal{O}$, can be represented as a polygon composed by a set of vertices and edges between these vertices. And obstacle polygon can be represented as a sequence of points (x_1, y_1), (x_2, y_2), $\ldots$, (x_m, y_m) and edges between them. In order to check if a robot configuration lies in an obstacle region we need to use what is called *collision checkers*. These methods are used for motion planning but are not included as a motion-planning algorithm itself. There are several choices for collision detection and $\mathcal{O}$ representation.

Let's denote a rigid robot as $\mathcal{A}$. A robot is transformed in $\mathcal{W}$ mapping every point of $\mathcal{A}$ into $\mathcal{W}$ preserving the distance between each pair of points of $\mathcal{A}$ and its orientation θ. If we have a robot $\mathcal{A}$ occupying a 2D region in $\mathcal{W}$, the translation of $\mathcal{A}$ is done by translating each point (x_i, y_i) for $(x_i + x_t, y_i + y_t)$. The same for the rotation movements, where $\mathcal{A}$ can be rotated a θ value. With this representation, in a 2D world, $\mathcal{A}$ has three degrees of freedom corresponding to:

- Two for translation (x, y).
- One for rotation θ.

In a 3D world, $\mathcal{A}$ has six degrees of freedom corresponding to:

- Three for translation (x, y, z).
- Three for rotation (θ, ψ, ϕ), known as pitch, roll and yaw.

This definition is valid for a rigid body composed by a boundary representation. We can have the case of a robot composed by a set of links: $\mathcal{A}_1$,$\mathcal{A}_2$,$\ldots$,$\mathcal{A}_m$, where m is the total number of links. A link A_i is attached to a link A_{i+1} by a joint j. Each joint between links has motion constrains, that must be specified, in order to represent the kinematics of the robot and to avoid self-collision problems.

The state space S term used in planning literature, is named as configuration space $\mathcal{C}$ in motion planning [77]. The configuration space $\mathcal{C}$ is the set of all possible configurations that could be applied to a robot.

So, let's suppose a world $\mathcal{W}$, a configuration space $\mathcal{C}$, and a robot $\mathcal{A}$. The robot $\mathcal{A}$ is at configuration $q \in \mathcal{C}$, where $q = (x_t, y_t, \theta)$. In addition, if $\mathcal{W}$ contains an obstacle region $\mathcal{O}$, this is represented as: $\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid A(q) \cap \mathcal{O} \neq \emptyset\}$. So, $\mathcal{C}_{free}$ is defined as $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$. Fig. 3.1 represents a configuration space C with the obstacle region $\mathcal{C}_{obs}$. Now, we can formally define a Motion Planning Problem, also called the Piano Mover's Problem 3.2:

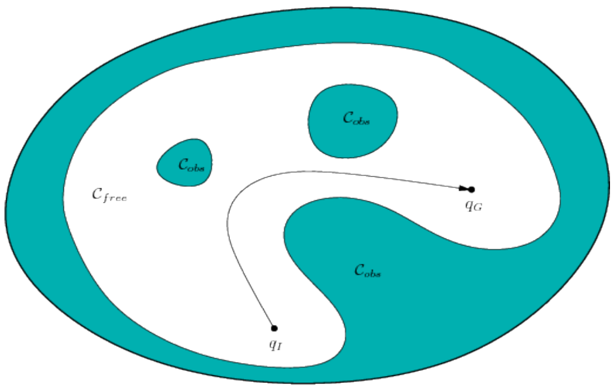


Figure 3.1: A configuration space $\mathcal{C}$ with a trajectory connecting q_I and q_G while avoiding $\mathcal{C}_{obs}$. The image has been extracted from [68].

Definition 3.2.1 *(Motion Planning Problem) A Motion Planning Problem is defined as a tuple* $\mathcal{P} =< \mathcal{W}, \mathcal{O}, \mathcal{A}, \mathcal{C}, \mathcal{C}_{free}, \mathcal{C}_{obs}, q_I, q_G >$*, where:*

- $\mathcal{W}$ is a world either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.
- $\mathcal{O}$ is the obstacle region such that $\mathcal{O} \subset \mathcal{W}$
- $\mathcal{A}$ is a robot. A robot can also be a collection of m links, such that: $\mathcal{A}_1$, $\mathcal{A}_2$, $\ldots$, $\mathcal{A}_m$.
- $\mathcal{C}$ is the set of all possible configurations of $\mathcal{A}$.
- $\mathcal{C}_{obs} \subseteq \mathcal{C}$ is the set of configurations which collide with the obstacle region.
- $\mathcal{C}_{free} \subseteq \mathcal{C}$ is the set of configurations which does not collide with the obstacle region.
- $q_I \in \mathcal{C}_{free}$ is the initial configuration.

- $q_G \in \mathcal{C}_{free}$ is the goal configuration.

A solution to a Motion Planning Problem is a collision-free continuous path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, where $\tau(0) = q_I$ and $\tau(1) = q_G$.

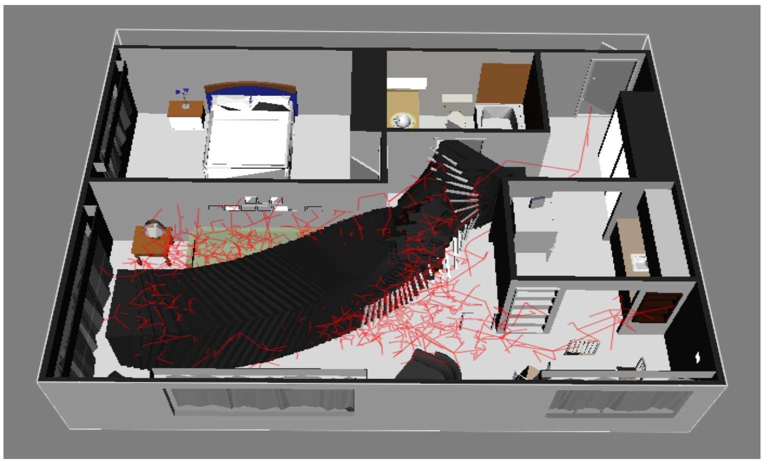


Figure 3.2: Piano Mover's Problem using RRT. The image has been produced using OMPL [104].

3.2.2 Sampling-based Motion Planning

Sampling-based motion planners approximate the connectivity of a configuration search space with a graph structure, avoiding the construction of $\mathcal{C}_{obs}$. The configuration space is sampled using sampling-based methods that we will review in following sections. Edges between sampled configurations represent collision free trajectories. For this reason, sampling-base motion planning requires a collision detector that checks if a sampled configuration or a trajectory yields in $\mathcal{C}_{obs}$. In case that a sampled configuration collides with the obstacle region, then q is not taken into consideration. The same happens with trajectories connecting two configurations q and q'. For checking a collision along a trajectory,

Sampling-based motion planners sample the configuration space with a determined resolution, sacrificing the notion of completeness, which says that for any input, the algorithms reports if there is a solution in a finite time. On the other hand, with sampling-based motion algorithms, we move into the notion of *probabilistic complete*, which means that with enough sampling points, the probability to find an existing solution converges to one if a solutions exists. Sampling-based motion planning is composed by these following components:

- A **Configuration Space** $\mathcal{C}$. For example, for a flying robot, the configuration space consist of all posible translations and rotations.
- **Control space**: Required only for systems with dynamics and system changes. For example The game of Koules [62].
- **State sampling**: It is done by a sampler. Configurations need to be selected in an infinite space in order to search over them.
- **Collision checkers**: Check if a configuration is colliding with the obstacle region. Also check if a trajectory between a pair of configurations is collision-free. For this purpose, the trajectory is divided into m intermediate configurations. If not one yields in the obstacle region, then the trajectory is collision-free.

We have defined what is the configuration space, so now, we are going to review some sampling methods and what are collision checkers.

3.2.3 Sampling Methods

The configuration space $\mathcal{C}$ is infinite, so a method to sample $\mathcal{C}$ is required in order to search over $\mathcal{C}$. Some of these sampling methods are:

- **Random sampling:** Where samples are chosen over $\mathcal{C}$ randomly.
- **Obstacle-based sampling:** It consist to interpolating configurations between invalid and valid samples and take the last valid configuration before an invalid one (very useful for narrow passages or problems where motions have to be done close to obstacles).
- **Gaussian sampling:** It starts sampling a random configuration and continues following a Gaussian distribution.
- **Low-dispersion sampling:** It samples large uncovered areas in order to make these uncovered areas as small as possible.

3.2.4 Collision Checkers or State Validation

Sampling-based motion planning avoids the construction of $\mathcal{C}_{obs}$. Over the generated samples, a collision detection module is used in order to validate these samples. A sampled configuration q is valid if $q \in \mathcal{C}_{free}$.A collision detection can be done by a logical predicate $\o : \mathcal{C} \rightarrow \{TRUE, FALSE\}$. So, if we sample a configuration q such that, $q \in \mathcal{C}_{obs}$, then $\o(q) = FALSE$. In other words, configuration$q$ is an invalid configuration. In addition motion validation has to check if the interpolation between two configurations q_1 and q_2 is valid, although both of them are located in the $\mathcal{C}_{free}$ space. Meaning, we have to check if $\tau \subset \mathcal{C}_{free}$. This is called motion validation or path checking. Collision checkers are a crucial part of motion planning, as the aim of a motion planners is to find a collision free trajectory that respect the robot kinematics.

3.2.5 Single-query and multiple-query models

To describe some state-of-the-art motion planning algorithms, it is important to distinguish between single-query and motion-query models.

Single-Query Models

The single-query models are greedy algorithms that don't explore all the $\mathcal{C}_{free}$ space, only the parts that are relevant or that depend on some bias. A valid bias could be to explore the configuration space towards a goal configuration. These algorithms can take a long time to build a search structure, but then can deeply dive into a solution. Given a graph $G(V, E)$, and starting with a vertex v that represents an initial configuration q_I, the usual process is as follows:

1. Select a vertex $q \in V$.
2. For a sampled vertex $q_{new} \in \mathcal{C}_{free}$, try to construct and edge τ from q to q_{new}, using the required collision detection methods.
3. Insert τ to E and q_{new} in V (if it was not already).
4. Check if solution exists.
5. If solution not found, then come back to point 1.

Algorithm 1 illustrates the basic procedure to build a *Rapidly-Random Exploring Tree* (RRT) [67]. This algorithm is one of the most well known and extended tree build process. Given an initial configuration q_I, a number of iterations K and a metric Δ_q it builds a graph G where each node is a collision-free sampled configuration and edges between nodes are collision-free trajectories. Per each iteration, the procedure selects a random configuration $q \subseteq \mathcal{C}$. Step 4 selects the nearest vertex q_{near} in G to q_{rand}; Then, on step 5, it selects a new configuration q_{new} which is computed by moving from q_{near} to q_{rand} using a metric Δ_q. Finally, The new configuration q_{new} and the edge from q_{near} to q_{new} are added to the graph G. After a number of K iterations the tree has been built. One may notice that there is no a collision checking process along the tree building. Procedures $RAND\_CONF()$

Algorithm 1 Rapidly-Exploring Random Tree.

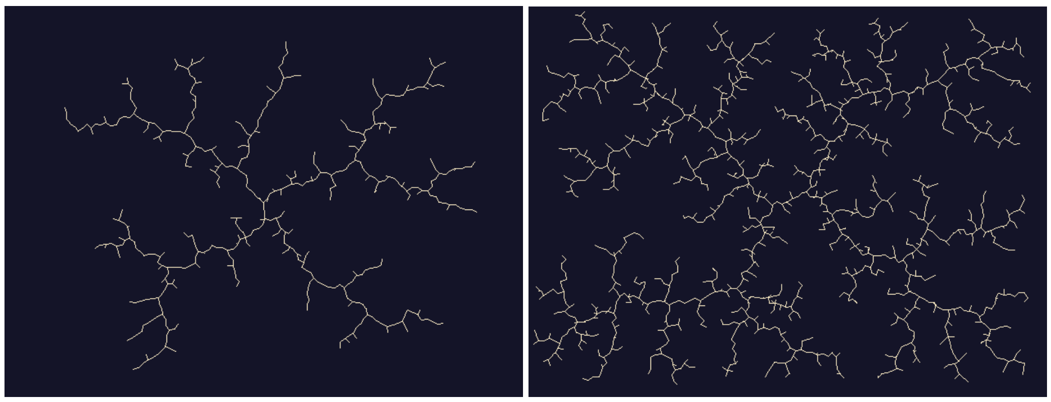
```
1: procedure BUILD_RRT($q_I$, $K$, $\Delta_q$)
2:     $G.init(q_I)$;
3:     for $i = 1$ to $K$ do
4:         $q_{rand} \leftarrow RAND\_CONF()$;
5:         $q_{near} \leftarrow NEAREST\_VERTEX(q_{rand}, G)$;
6:         $q_{new} \leftarrow NEW\_CONF(q_{near}, \Delta_q)$;
7:         $G.add\_vertex(q_{new})$;
8:         $G.add\_edge(q_{near}, q_{new})$;
       return $G$;
```

and $NEW\_CONF(q_{near}, \Delta_q)$ could be replaced by $RAND\_FREE\_CONF()$ and $NEW\_FREE\_CONF(q_{near}, \Delta_q)$. However, collision checkers are usually represented decoupled to motion planning algorithms, so we follow the same convention. Fig. 3.3 illustrates the process of building a RRT after a number of iterations, being $K = 500, 1500, 2500, 500$ and the sampling method is *random sampling*. These images have been obtained using the code in [69].

For the experiments related in this thesis, we are going to use a variation of RRT. This variation is an algorithm called Rapidly-Exploring Random Tree Connect (RRT-Connect) [60]. This algorithm 2 works generating two trees one towards the other, attempting to connect both trees. This algorithm has demonstrated to be useful for environments with narrow passages. The algorithm starts building two separates trees. The first tree called T_a is constructed from the initial configuration q_I and the other tree T_b from the goal configuration q_G; In each iteration, T_a is grown as a normal *RRT*. If a new vertex q_s is added to T_a, then the algorithm attempts to extend T_b towards q_s. This means that T_b tries to be extended towards T_a. If both trees T_a and T_b are finally connected, then there is a path from q_I to q_G through both trees, which now are represented as a single graph.

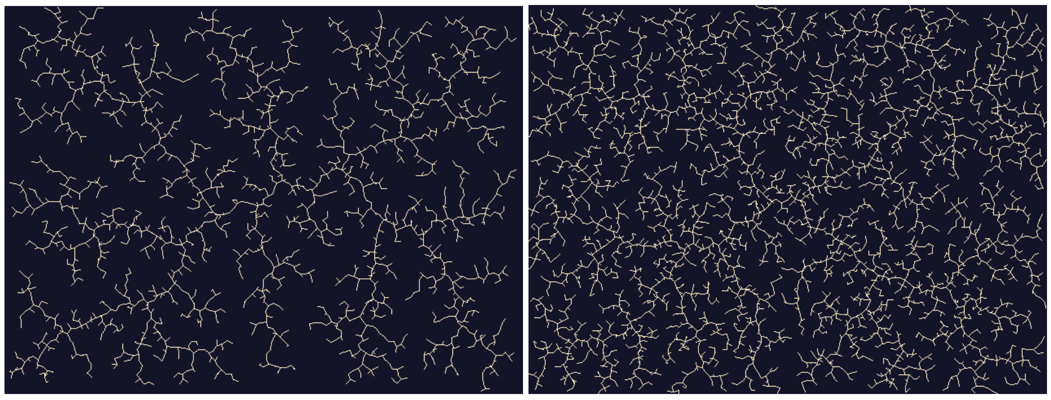
Multiple-Query Models

On the other hand, multiple-query models work by sampling $\mathcal{C}$ as much as they can, constructing what is known as a *roadmap*. Once the roadmap is built, any query represented as a pair (q_I, q_G) can be used on the same roadmap, without the requirement of constructing a new tree or graph, as in RRT family algorithms. These algorithms cover a wide space with uniformity, but needs lots of sampling to cover the space. The most well known example of a multi-query algorithm is the *Probabilistic Roadmap* (PRM) [58]. The basic process to build a roadmap is



(a) RRT with 500 iterations.

(b) RRT with 1500 iterations.



(c) RRT with 2500 iterations.

(d) RRT with 5000 iterations.

Figure 3.3: RRT execution after 500, 1500, 2500 and 5000 iterations.

Algorithm 2 Rapidly-Exploring Random Tree Connect

```
 1: procedure BUILD_RRT-CONNECT($q_I$, $q_G$, $K$)
 2:     $T_a.init(q_I)$;
 3:     $T_b.init(q_G)$;
 4:     for $i = 1$ to $K$ do
 5:         $q_{rand} \leftarrow RAND\_CONF()$;
 6:         $q_{near} \leftarrow NEAREST\_VERTEX(q_{rand}, T_a)$;
 7:         $q_s \leftarrow STOPPING\_CONFIGURATION(q_{near}, q_{rand})$;
 8:         if $q_s \neq q_n$ then
 9:             $T_a.add\_vertex(q_s)$;
10:             $T_a.add\_edge(q_{near}, q_s)$;
11:             $q'_{near} \leftarrow NEAREST\_VERTEX(T_b, q_s)$;
12:             $q'_s \leftarrow STOPPING - CONFIGURATION(q'_{near}, q_s)$
13:             if $q'_s \neq q'_{near}$ then
14:                 $T_b.add\_vertex(q'_s)$;
15:                 $T_b.add\_edge(q'_{near}, q'_s)$;
16:             if $q'_s = q_s$ then return $SOLUTION$
17:         if $|T_a| > |T_b|$ then $SWAP(T_a, T_b)$
      return $FAILURE$
```

shown in algorithm 3. A PRM starts sampling a number n of configurations in the configuration space. Only good configurations are taken, where good configurations those ones which are collision-free. Once the state space has been sampled, the algorithm builds the map attempting to connect each configuration q in V with the k-nearest configurations q_{near} in V. If the new pair (q, q_{near}) does not exist in E and both configurations q and q_{near} can be connected, then this edge is added to the graph. At the end, the procedure returns the whole constructed graph. A Probabilistic Roadmap Planner consists of two phases:

- **Preprocessing phase:** It builds the graph G following the algorithm 3
- **Query phase:** Given a pair (q_I, q_G), a search algorithm tries to find a path from q_I to q_G;

It is easy to see that in single-query algorithms, solutions can be very different for the same or similar input. On the other hand, in multiple-query algorithms, the solution for the same input will be the same, unless the roadmap is not changed in any way. There are a number of other motion planning algorithms like KPIECE [103], PDST [62], EST [50] and variations of well know algorithms like RRT* and PRM* [54], Lazy-RRT and Lazy-PRM [7].

Algorithm 3 Probabilistic Roadmap

```
1: procedure BUILD_PRM($n$, $K$)
2:     $G.init(\emptyset)$;
3:     $V \leftarrow \emptyset$
4:     $E \leftarrow \emptyset$
5:     while $|V| < n$ do
6:         $q_{rand} \leftarrow RAND\_CONF()$;
7:         $V \leftarrow V \bigcup q_{rand}$
8:     for all $q \in V$ do
9:         $Q_{nearest} \leftarrow K\_NEAREST(q, V, K)$
10:        for all $q_{near} \in Q_{nearest}$ do
11:            if $edge(q, q_{near} \notin E$ and $CONNECT(q, q_{near})$ then
12:                $E \leftarrow E \bigcup edge(q, q_{near})$
       return $G$
```

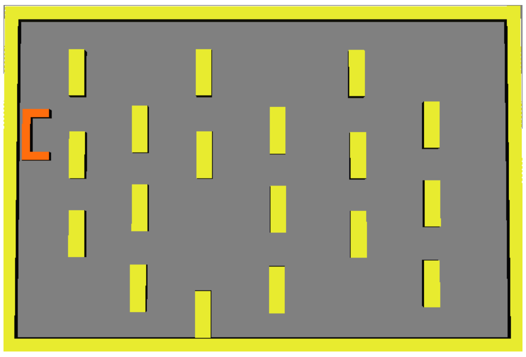
3.2.6 Example

Consider a motion planning problem where a rigid robot $\mathcal{A}$ must be moved from an initial configuration to a goal configuration q_G in a 2D world $\mathcal{W}$ whose dimension are $N \times M$, where $N = 660$ and $M = 440$. The obstacle region $\mathcal{O}$ is denoted by the yellow marks. The configuration space $\mathcal{C}$ represents the set of possible robot configurations as triplets $\langle x, y, theta \rangle$. $\mathcal{C}_{free}$ denote the configurations in $\mathcal{C}$ which are collision free, while $\mathcal{C}_{obs}$ denote those configurations which collides with $\mathcal{O}$. The initial configuration as stated in Fig. 3.4a is $q_I = \langle 35, -169, 0 \rangle$ and the goal configuration is $q_G = \langle 604, -312, 90 \rangle$ as represented in Fig. 3.4b. The goal is to find a collision-free trajectory τ between q_I and q_G.

Fig. 3.5 shows the results of applying RRT and RRT-Connect to the motion planning problem described before. Notice that the trajectories are different, where the RRT-Connect is the shorter one. Both algorithms are single query, so the tree needs to be rebuilt, after each execution.

Fig. 3.6 illustrates the results after applying the PRM algorithm. Fig. 3.6a shows the valid configuration samples while Fig. 3.6b contains the whole roadmap with edges connecting the samples. Notice that preprocessing phase for PRM, samples the configuration $\mathcal{C}$ substantially more than RRT or RRT-Connect.

A possible solution to the problem depicted by Fig. 3.4 using RRT-Connect is a collision-free trajectory of the form:


(a) Rigid body with its initial configuration.

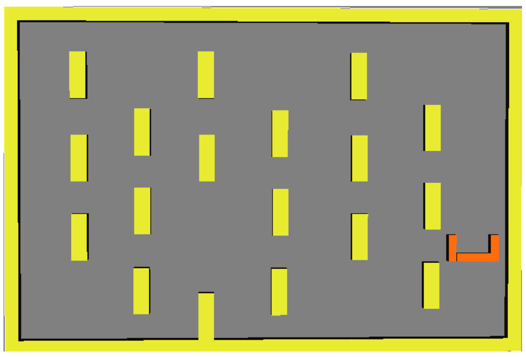
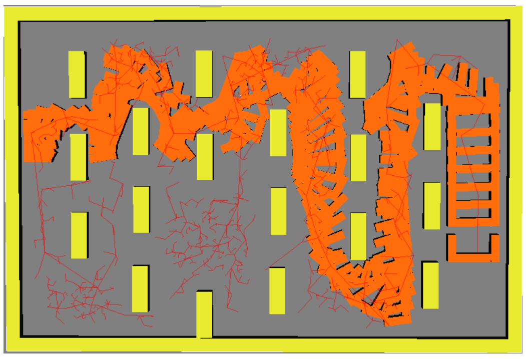

(b) Rigid body with its goal configuration.

Figure 3.4: A simple 2D environment with the initial and goal configurations. The OMPL [101] has been used to generate these environments.


(a) RRT trace.

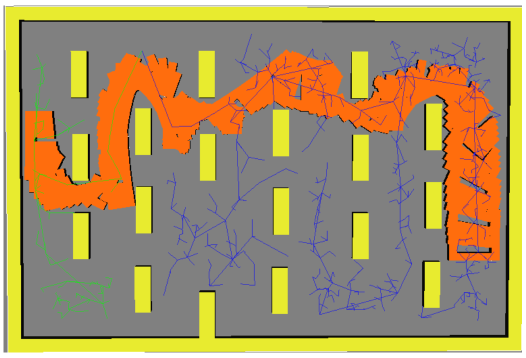
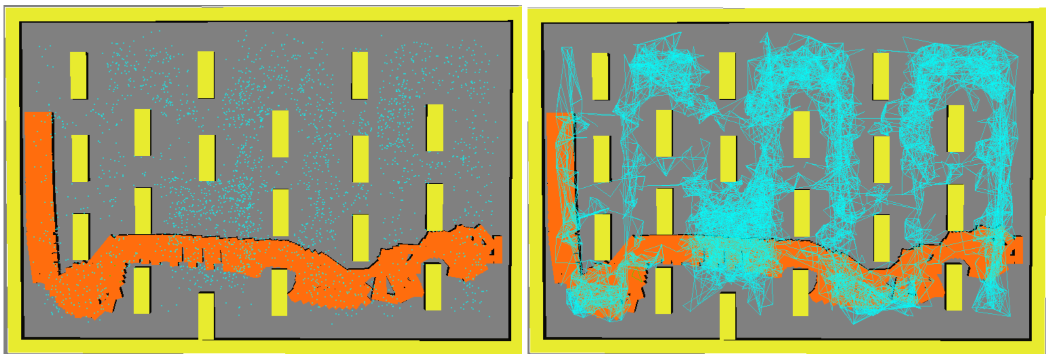

(b) RRT-Connect trace.

Figure 3.5: RRT and RRT-Connect traces. In Fig. 3.5a the red lines represent the tree as it has been grown from q_I. In Fig. 3.5b green lines represent the tree build from q_I while blue lines represent the tre built from q_G. The orange trace represents the intermediate configurations along the trajectory. These plans have been obtained using OMPL [101].



(a) Configuration samples using PRM. (b) The whole PRM.

Figure 3.6: PRM trace. The orange trace represents the intermediate configurations along the trajectory. These plans have been obtained using OMPL [101].

$$\tau = \{\langle 35.2, -169.4, 0.0\rangle, \langle 37.2, -163.7, -0.008\rangle, \ldots, \langle 604.2, -312.4, 1.6\rangle\},$$

where each tuple represents a configuration of the form $\langle x, y, \theta\rangle$.

3.3 Summary

Combined Task and Motion Planning is the problem of integrated symbolic reasoning with geometric reasoning. While task planning can specify long term tasks and high-level goals, motion planning can reason over the geometry on the continuous space. It can solve manipulation problems like moving an arm to place an end effector in a feasible pose for grasping an object, or where to move the base to have a feasible grasping pose.

Sampling-based motion planning is the most extended method, which relies on sampling the configuration space using different methods. Motion planning algorithms are divided into single-query and multiple-query algorithms. The first ones build a tree for a single problem, attempting to connect the initial configuration and the goal configuration. The second ones try to sample as much of the configuration space and creates a graph where basic search algorithms can be applied to find a trajectory for different queries. The aim of a motion planner is to find a collision-free trajectory from the initial to the goal configurations. Collision checkers must be used in order to avoid collisions during sampling and connection phase. In both cases, these algorithms cannot solve per se, the different problems proposed in this thesis. CTMP problems are challenging and require a robust and efficient integration to avoid backtracking. For the rest of this work, motion planners and collision checkers are used off-the-shelf to compute both, base and arm trajectories and to avoid collisions.

PART II

Combined Task and Motion Planning

CHAPTER **4**

Basic Model for Combined Task and Motion Planning in 2D

4.1 Introduction

The aim of this chapter is to provide a novel integration of task and motion planning where the symbolic and geometrical components in robot planning are addressed in combination, with both parts playing an active role in the search for plans. Unlike recent works that have a similar motivation [100, 34], our integration results in problems that can be solved with classical planners off the shelf. For this, we discretize the configuration space and introduce first a formulation of task and motion planning in Functional STRIPS with state constraints.[1] Functional STRIPS is an expressive first-order planning language that accommodates constraints, functions, and numerical variables for which a powerful heuristic-search planner has been developed recently [32]. State constraints are used to prevent collisions and while they are not supported in Functional STRIPS, we show how they can be handled.

The state constraints are formulas that must be true in all states encountered throughout the execution of a plan, and are used to prevent collisions. The functions are used in turn for encoding the changes, taking into account the geometrical dimensions of objects and their poses. State constraints are not a standard feature of planning languages, although their convenience has been thoroughly

[1]State constraints should not be confused with state invariants: a state constraint *enforces* a formula to be true in all reachable states, while a state invariant is a formula that holds in all reachable states.

discussed in the literature on reasoning about actions [71, 98]. They are not part of Functional STRIPS, but We will show that it is not difficult to add them to the language and computational model.

The combined problem of task and motion planning expressed in FSTRIPS with state constraints can be solved with the Functional STRIPS planner FS0 [32], that performs a greedy best-first search from the initial state, using an informed heuristic that is a variation of the relaxed planning graph heuristic introduced by FF [49]. In addition, since the Functional STRIPS heuristic is expensive, we consider two alternatives. The first is to compile these Functional STRIPS representations into standard STRIPS for enabling the use of state-of-the-art STRIPS planners. The second is the use of width-based planning algorithms [72], that have been shown to be effective and do not use any heuristic. Width-based algorithms have been used to achieve state-of-the-art results in classical planning [72], in the Atari video-games [76], and the games of the general-video game AI competition [39]. As we will see, they also appear to be competitive for combined task and motion planning.

4.2 Computation of the Heuristics

The FS0 planner solves problems in Functional STRIPS under two restrictions: first, precondition, condition, and goal formulas are limited to be conjunctions of literals; second, terms featuring nested fluents are excluded. These problems are solved using a greedy best-first search from the initial state, guided by a heuristic that is a departure from generalizations of relaxed planning graph heuristic in FF [49] that follow the so-called *value-accumulating semantics* [47, 42, 51]. In such an interpretation, each fact layer k of the relaxed planning graph (RPG) keeps for each state variable X a set X^k of possible values that grow monotonically with k. A conjunction of atoms in a precondition, condition, or goal, is satisfied in a layer, potentially triggering new effects, if each atom in the conjunction appears in the layer.

This interpretation is adequate for propositional settings, but too weak for the first-order setting of Functional STRIPS where for example, a conjunction like $(X_1 < X_2) \wedge (X_2 < X_3)$ will be deemed as satisfied in a layer k if the set of possible values in that layer are $X_1^k = X_2^k = X_3^k = \{0, 1\}$. Indeed, there are then logical interpretations that can make *each* of the two atoms true, but no logical interpretation that can make *both* atoms true at the same time.

The main departure in the computation of the relaxed planning graph in Functional STRIPS is that the sets X^k of possible values for the state variables X in layer k are used to determine whether a condition, precondition, or goal formula A is *satisfiable* in layer k by following the rules of *first-order logic*; namely, A is satisfiable if there is at least one state s compatible with the variable domains, i.e., where $X^s \in X^k$ for each state variable X, such that $A^s = true$. If A is a conjunction of atoms $p_1(t_1) \wedge \cdots \wedge p_n(t_n)$ and fluents do not appear nested, the problem of determining the satisfiability of A in layer k maps into a constraint satisfaction problem (CSP) where the variables are the state variable X with domains X^k, and the constraints are given by atoms. The problem can be solved in FS0 either exactly or approximately. In the first case, the CSP is solved fully, which can be fast, although it is exponential in the worst case; in the second, effective but polynomial constraint propagation algorithms are used instead, and A is taken to be satisfiable if the propagation does not prove inconsistency.

To complete the construction of the relaxed planning graph in FS0, a value $c' \notin X^k$ is added to the domain X^{k+1} of state variable $X = f(c)$ in the next layer $k+1$, supported by effect $C \rightarrow f(t) := t_1$ of action a, iff there is state s compatible with the possible values of the variables in layer k such that $Pre(a)^s = C^s = true$, $t^s = c$, and $t_1^s = c'$. This last condition actually amounts to testing the satisfiability of a CSP with variables X, domains X^k, and the constraints $Pre(a) = true$, $C = true$, $t = c$, and $t_1 = c'$. Likewise, the graph construction ends in layer k if the goal G is then satisfiable. Like in FF, the heuristic value $h(s)$ stands for the number of actions in the plan extracted backward from the first layer where G is satisfiable.

4.3 Functional STRIPS Model with State Constraints

Due to the ability of Functional STRIPS to handle functions and constraints, it is rather simple to express problems where the *goal* involves geometrical constraints. It is less direct, however, to enforce such constraints *throughout the execution* of plans, which is critical in robotics for avoiding collisions. We show how to achieve this by extending Functional STRIPS with state constraints [71, 98].

4.3.1 Syntax and Semantics

Definition 4.3.1 *(FSTRIPS Planning Problem with* state constraints*) A FSTRIPS planning problem with* state constraints *is a tuple $P = \langle S, I, O, G, C\rangle$ where:*

- S is the set the non-standard symbols (fixed and fluent) and their types.
- I provides the initial unique denotation s_0 of such symbols.

- O stands for the actions.
- G is the goal.
- C stands for a set of formulas expressing the constraints.

The syntax for these formulas is the same as for those encoding the goal G but their semantics are different. State constraints are used for encoding *implicit preconditions*. Namely, an action a is deemed applicable in a state s when *both* $[Pre(a)]^s = true$ *and* the state s_a that results from applying a to s is such that $c^s = true$ for every state constraint $c \in C$. In other words, an action a is *non-executable* in s if its execution leads to a state s_a that violates a state constraint.

4.4 Heuristics for Handling State Constraints

State constraints affect the construction of the relaxed planning graph and the resulting heuristic, but the required changes are minor. We had before that a value $c' \notin X^k$ is added to the domain X^{k+1} of state variable $X = f(c)$ in the next layer $k+1$ of the planning graph, supported by an effect $C \rightarrow f(t) := t_1$ of action a, iff there is state s compatible with the possible values X^k of the variables x in layer k such that $Pre(a)^s = C^s = true$, $t^s = c$, and $t_1^s = c'$. The change in the presence of state constraints is that such states s must satisfy all the state constraints as well. Indeed, the states that are possible according to the sets X^k of possible values of each state variable X but which do not satisfy a state constraint are pruned, and never considered in the construction of the planning graph. In particular, the goal G is satisfied in layer k if an interpretation s compatible with the possible values X^k satisfies both G and the state constraints. As an example, an action effect like $X > 2 \rightarrow Y := X + 1$ can support the values $Y = 4$ and $Y = 5$ in layer $k + 1$ if the set of possible values for X in layer k is $X^k = \{3, 4\}$. On the other hand, if $X < 4$ is a state constraint, only the first of the two Y values will get support, as the interpretations where $X = 4$ is true will be pruned. Likewise, a goal like $X > 3$ will not be satisfiable then in layer k either.

4.5 Example with State Constraints

We illustrate next the usefulness of state constraints both in terms of modeling and computation with a couple of examples, reporting their running times as well.

The **Missionaries and Cannibals** (M&C) problem has received wide attention since the early days of AI [2] as a toy problem that is nevertheless representative of a wider class of transportation-under-constraints problems.

In its standard version, the problem places three missionaries and three cannibals on the left bank of a river which they all want to cross. A single boat is available that can hold only two people at a time, regardless of whether they are missionaries or cannibals. Apparently, the missionaries do not want to be outnumbered by the cannibals, be it on either bank of the river or inside of the boat, for fear of the cannibals exercising their defining inclination.[2] The goal is to find an appropriate schedule of river crossings that transports everyone to the right bank of the river in a safe manner.

We model a generalization of the problem for n missionaries and n cannibals ($n \geq 3$) on a complete graph. There are fixed symbols $l_1, \ldots, l_m$ representing the locations, and state variables $nc(l)$ and $nm(l)$ representing the number of cannibals and missionaries at each location l. In addition, the 0-ary functional symbol X represents the current location of the boat. The actions $move(c, m, l)$, with $0 \leq c, m \leq 2$, $c + m \in \{1, 2\}$, move c cannibals and m missionaries in one boat trip from the current location to l. Their preconditions are $c \leq nc(X)$ and $m \leq nm(X)$, and their effects are $X := l$, $nc(l) := nc(l) + c$, $nc(X) := nc(X) - c$, and analogously for the missionaries. Finally, the restriction on cannibals not outnumbering missionaries in a location l is modeled by the binary state constraint $nm(l) \geq nc(l) \vee nm(l) = 0$. The "inside of the boat" restriction is encoded as part of the $move$ action.

As a second example, consider a simple **navigation problem** with geometrical obstacles in which an $n \times m$ grid contains a robot that has to reach a goal cell while avoiding obstacles. For simplicity, let us assume for now a point robot with no geometry, and rectangular shape obstacles o which can be represented by a couple of coordinate points (x_o, y_o) and (x'_o, y'_o), with $x_o < x'_o$ and $y_o < y'_o$ (obstacles having other shapes can be thought of as a combination of smaller rectangles). The location of the robot is represented by two 0-arity fluent functional symbols x and y with values in $[1, n]$ and $[1, m]$. The actions $move(dx, dy)$ with $dx, dy \in [-1, 1]$ move the agent to adjacent locations, including diagonals, with effects $x := max(0, min(x + dx, n))$ and $y := max(0, min(y + dy, m))$. Avoidance of obstacles o in any plan can then be represented succinctly through the *state constraints* $\neg(x_o \leq x \leq x'_o \wedge y_o \leq y \leq y'_o)$ that say that the robot cannot be inside one of the rectangles. The problem has as many state constraints as obstacles, and

[2] A historically more accurate version of the problem has it that it is the cannibals that do not want to be outnumbered by the missionaries for fear of being converted, but we restrict our discussion to the first version for the sake of tradition.

each constraint involves the coordinates of a rectangle and the coordinates of the robot. The extension of the FS0 planner with state constraints can handle problems such as this, with a linear number of geometrical obstacles, in less than 0.1 seconds for 10×10 grids, and in less than 15 seconds for 50×50 grids with 2500 cells.

We have actually tested the planner on a number of randomly generated instances of increasing size for each of these two domains.[3] In the case of the M&C domain, the planner scales up pretty well, handling problem sizes of 20-node location graphs and 12 missionaries plus 12 cannibals with relative ease: an instance with a 10-node graph and $9 + 9$ missionaries and cannibals is solved in 80 sec. by finding a plan of length 49 after expanding 637 nodes. An instance with 20 nodes and $12 + 12$ individuals takes 379 sec. and 183 node expansions to find a plan of length 45. Similarly, the planner handles navigation problems with a linear number of geometrical obstacles in less than 0.1 sec. for 10×10 grids and less than 15 sec. for 50×50 grids.

4.6 Modeling Task and Motion Planning

We consider a more general type of problems where there is a robot (gripper) that can translate, rotate, pick up objects, and drop them. The domain encoding in Functional STRIPS with constraints is shown in Figure 4.2, and the instances to be considered are shown in Figure 4.1. For convenience, the encoding distinguishes actions for translations and rotations with an object being held and without, as in the former case the action changes the configuration of both the robot and the object being held. The main state variables in the problem represent the configurations $conf(r)$ and $conf(o)$ of the robot and the different movable objects o. Symbols like $translated$ and $rotated$ denote fixed functions.

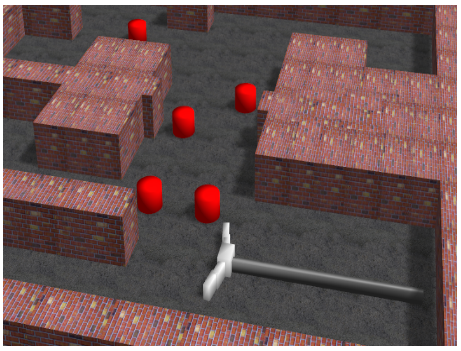
Symbol $conf$ is a functional fluent, and r and o are constant symbols representing the robot and each one of the objects. The other state variable represents the status of the gripper: $hand = o$ means that the object o is being held, and hence that it will move with the robot, and $hand = free$ that the gripper is empty. The pick up and drop actions just change the state of the gripper, and for the robot to pick up an object, the gripper must be empty and the tip of the gripper must be sufficiently close to the object, something that is captured by a fixed boolean symbol $graspable$ whose denotation is an actual function that determines if the robot and the object are in a configuration where the robot can grasp the

[3] Problem encodings for which empirical results are discussed are available at `https://goo.gl/67Zibk`.

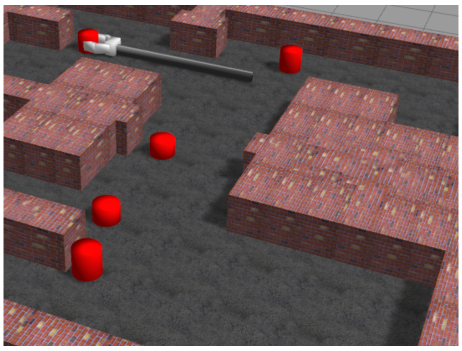
object. The other fixed function symbols used are $translated$, $rotated$, $orotated$, $no\_overlap$, and $valid$. They are used to capture the configuration changes that result from the actions, and the state constraints. The function denoted by the symbol $translated$ is used to update the robot configuration when the robot translates in a certain direction, and to update an object configuration when the robot translates while holding an object. The symbol $rotated$ denotes a function that captures the robot configuration that results when the robot rotates in an angular direction, and $orotated$ represents a function that captures the object configuration that results when the robot rotates while holding the object. The difference between $rotated$ and $orotated$ arises because the robot, which is actually a robot gripper, rotates over its center of mass which is not the center of mass of the object being held, that is closer to the tip of the gripper. Likewise, the fixed symbol $no\_overlap$ denotes a boolean procedure that accepts as arguments a robot and a movable object, or two objects, along with their configurations, and determines whether they overlap in space. Finally, the boolean function denoted by the symbol $valid$ takes a configuration as an input and check if it is valid. Actually, there is a single non valid configuration that is the null configuration, denoted $\perp$. The functions that compute updated configurations output the null configuration when the resulting configuration is not physically possible (part of the object gets out of the grid or overlaps a fixed obstacle). While the $no\_overlap$ state constraints take care of collisions among *pairs* of movable objects (including the robot), the $valid$ state constraints take care of collisions of *one* movable object with the fixed environment.

In the 2D grid worlds that we consider, configurations are triplets $\langle x, y, \theta \rangle$ where x and y capture the center-of-mass position of the object or robot, and θ its orientation. The 2D space is discretized according to a varying resolution parameter r into a grid of size $r \times r$ for $r \in \{10, 30, 50\}$, while angles are discretized into $r_a = 8$ values. Thus the possible number of configurations for each object and robot is in the order of $\#c = r_a \times r^2$, and since the gripper can be in $n + 1$ states, where n is the number of objects, the total size $|S|$ of the state space is $(n+1) \times \#c^{n+1}$, which for $n = 5$ and $r = 50$ is $|S| = 6 * (8 * 50^2)^6 = 384 * 10^{25}$. This is pretty large, but precisely the appeal of classical planners is that their performance is not tied to the size of the state space, as they can often search for plans by considering a tiny fraction of the states.

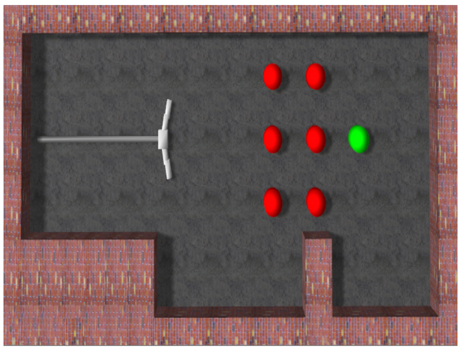
Given the discretization of the configuration space, the fixed functions (procedures) $translated$ and $rotated$, accept a robot or object configuration in the discretized space, and a direction (angular for rotations), and return the resulting configuration that is also in the discretized space. There are 8 possible transla-



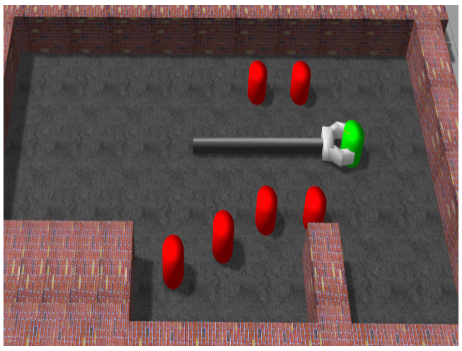
(a)



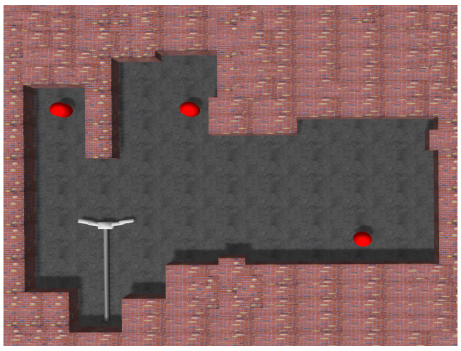
(b)



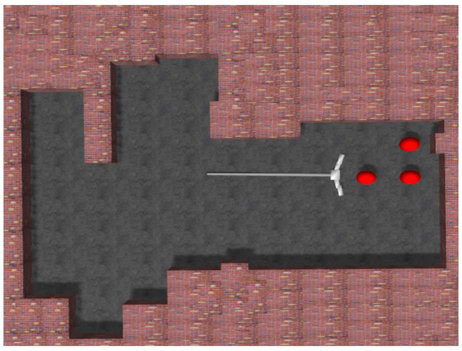
(c)



(d)



(e)



(f)

Figure 4.1: Three task and motion instances considered with the initial state shown on the left and the final state where goal is true shown on the right. *MOVING (M)* domain in which a robot has to reach a target configuration, and a number of obstacles need to be picked up and moved to get them out of the way. The other two domains considered are *CLUTTER (C)*, where robot has to pick up an object obstructed by other objects, and *TIDYING-UP* (T) where objects have to be transported to a target area. Tables below show features of the encodings and results obtained by running six planners.

tion directions and each one moves the robot (and the object being held if any) to the corresponding neighboring cell in the $r \times r$ grid, by adding $+1$, 0, or -1 to the x and y coordinates. The angular directions are just two: clockwise and counter-clockwise, and they add or substract 45 degrees to the orientation angle. The procedure $orotated$ that updates the configuration of the object being held after a rotation is the only one that may result in a configuration that is not in the discretized space. In the planning encoding, this is avoided by mapping the resulting object configuration into the nearest discretized configuration.

The configuration triplets are represented by suitable identifiers, and for efficiency purposes, all the functions operating on configurations are precompiled so that applying a function is retrieving an entry from a table. This means that a function invocation like $translated(r, conf(r), d)$ becomes a table lookup that returns a configuration id, so that an effect like $conf(r) := translated(r, conf(r), d)$ simply sets the state variable $conf(r)$ to id. Similarly, the boolean procedures $no\_overlap$ and $graspable$ take configuration id's as arguments, and return a boolean value by a table lookup. In addition, $no\_overlap$ takes also the object id's, as it needs to have access to the object and robot geometries, that can be arbitrary. The robot and object geometries (shapes) are given as tables that are visible only to these fixed procedures. The fixed physical environment (walls and non-movable obstacles) are compiled into these procedures, and actions that overlap fixed objects result in the null configuration. The function $valid$ actually just checks if its argument is the null-configuration or not.

```
types:  dir, adir, conf, obj, robot, thing

fixed function symbols:
  translated(t:thing, c:conf, d:dir): conf
  rotated(r:robot, c:conf, d:adir): conf
  orotated(o:object, c1,c2:conf, d:adir): conf
  graspable(c1, c2: conf): bool
  no_overlap(t1, t2:thing, c1, c2:conf): bool
  valid(c1:conf): bool

fluents:
  conf(t: thing): conf
  hand: {free} U object

action translate(d: dir)
  prec: hand = free
  eff conf(r) := translated(r,conf(r),d)

action rotate(d: adir)
  prec: hand = free
  eff conf(r) := rotated(r,conf(r),d)

action translate-with-obj(o: obj, d: dir)
  prec: hand = o
  eff conf(r) := translated(r,conf(r),d)
  eff conf(o) := translated(o,conf(o),d)

action rotate_with_obj(o: obj, d: adir)
  prec: hand = o
  eff conf(r) := rotated(r,conf(r),d)
  eff conf(o) := orotated(o,conf(o),conf(r),d)

action pickup(o: obj)
  prec: graspable(conf(r),conf(o))= true
  prec: hand = free
  eff  hand := o

action drop(o: obj)
  prec: hand = o
  eff:  hand := free

state constraints:
 no_overlap(t, t': thing, conf(t), conf(t'))
 valid(c: conf)
```

Figure 4.2: Task and motion planning in Functional STRIPS. The six fixed function symbols denote actual functions defined through external procedures that are sensitive to the fixed environment and discretization used.

We will consider three families of task and motion planning instances all based on this domain encoding: one where the robot has to move to a target destination by moving objects that are on the way, one where objects have to be moved to destinations, and a third, where an object has to be grasped by moving blocking objects. All of the problems combine elements from task and motion planning. Some instances are shown in Figure 4.1.

4.7 Computation 1: Translations to STRIPS

Instances of the domain above can be solved by running a Functional STRIPS planner. Yet, the encodings do not exploit the full power of the language, and suggest a simple compilation into STRIPS. Since there are very good STRIPS planners, we describe the compilation below and report the results obtained by running state-of-the-art STRIPS planners too. The compilation into STRIPS involves two parts: compiling the functions away, and compiling the state constraints away. The first part is simple: since the effects of STRIPS actions are state-independent, we make the configurations mentioned in an action schema into arguments of the action schema, and ground these arguments to each of their possible values. Then one must check that these values are in the right relation, and use them to establish the action precondition, add, and delete lists. This allows us to compile away the fixed functions. In addition, atoms $X = f(Y)$ and assignments $X := f(Y)$ where f is a fluent, like $hand$ and $config$, are converted into relations $f(X, Y)$. For example, the Functional STRIPS schema $rotate\_with\_obj(o : obj, d : adir)$ maps into the STRIPS ground actions:

```
action rotate_with_obj(c1r, c2r, c1o, c2o, o, d)
precondition: hand(o), conf(r,c1r), conf(o,c1o)
add: conf(r,c2r), conf(o,c2o)
del: conf(r,c1r), conf(o,c1o)
```

Figure 4.3: PDDL domain encoding for illustrating the Classical Planning Problem in STRIPS of the previous example.

where o is an object, d is an angular direction (clockwise, counter-clockwise), and the c's are configurations in the table that must obey the relations $c2r = rotated(r, c1r, d)$ and $c2o = orotated(o, c1o, c1r, d)$. Since $graspable(cr, co)$ must be true when $hand(o)$ becomes true, and remains true as long as the object is being held, the arguments of the action must also satisfy $graspable(c1r, c1o) = true$. The number of ground actions $rotate\_with\_obj(c1r, c2r, c1o, c2o, o, d)$ ends up being linear in the total number of configurations and not quartic.

Additional action preconditions and effects result from the compilation of state constraints. The compilation of the $valid$ constraint is direct: the grounding is limited to object and robot configurations that are valid (i.e., that do not overlap walls or fixed obstacles). For compiling the $no\_overlap$ constraint, new atoms $free(cell)$ are added and used in the *preconditions* and *delete effects* of all (ground) actions where the robot or an object would end up occupying $cell$ after the action. Likewise, $free(cell)$ atoms are included in the add list of actions that make $cell$ free after the action. Recall that state constraints encode action preconditions implicitly: an action a is not deemed to be applicable in a state s when the resulting state violates a state constraint. For compiling these constraints into STRIPS, we need to convert such implicit preconditions into explicit preconditions. Propositional state constraints A, are actually, a particular type of temporal extended goal or LTL formula of the form "always A" [3], and methods have been devised for compiling such goals into STRIPS and variations [26]. In our setting, the ground formulas A take the form $\neg(conf(r, o1) \wedge conf(o2, c2))$ where $o1$ is an object (or robot) and $o2$ is an object.

4.8 Computation 2: Heuristic and Width-based Search

We consider a second computational approach, width-based search [72] that has been developed in the context of classical planning and uses no heuristic at all.

Width-based algorithms [72, 73] are simple but not that well known, so it's worth reviewing them briefly. An expression $X = x$ where X is a state variable and x a value, is called an atom, and a state s is said to make the atom $X = x$ true if X has value x in the state s. Similarly, s is said to make a tuple t (set) of atoms true if s makes each atom in t true.

Iterated Width or IW is a sequence of calls $\text{IW}(i)$ for $i = 1, 2, \ldots$ where $\text{IW}(i)$ is a plain *breadth-first search* with one change: the states s generated in the search are pruned when they are not the *first* state in the search to make true some tuple (set) of i atoms or less. Thus, $\text{IW}(i)$ for $i = 1$, i.e., IW(1), is a breadth-first search where a newly generated state s is pruned when there is no *new atom* made true by s, IW(2) is a breadth-first search where a newly generated state s is pruned when there is no *new atom pair* made true by s, and so on.

Best-first width search (BFWS) is a best-first search algorithm that combines width-based measures with an implicit form of goal serialization. BFWS tracks two measures: the number of atomic goals that are true in s, and the size of the smallest tuple of atoms t that is true in s and false in all the states generated before s that have the same number of true goals as s. The evaluation function $f(s)$ in BFWS is given by the second measure, called the novelty measure (smaller is better), with ties broken by favoring states with a maximum number of true goals. We use the implementation of these algorithms in the planning toolkit LAPTK that supports width-based search for problems expressed in STRIPS, and more recently in Functional STRIPS [92].

4.9 Experimental Results

We evaluated 6 planners on 5 instances combining task and motion planning aspects that are beyond the scope of motion planners alone. Using three resolutions that partition the 2D square grid into 10x10, 30x30, and 50x50 cells, the 5 instances result into 15 planning problems. The instances correspond to*MOVING* (M) and *CLUTTER*, where a robot (gripper) must reach a target or grasp an object by moving objects out of the way, and *TIDYING-UP* (T) where objects must be transported to a destination (set of configurations).

Reach a target configuration, is a common motion planning problem, but the inclusion of objects which can be moved by the robot introduce a task component which is no trivial since the possible obstructions must be captured by the motion planner while the decision about which object must be moved, by the task planner, producing the commented backtracking. Our approach shows that state constraints allows to produce plans which no overlap in space in a reasonable time, even with a huge number of possible overlaps between objects, being more than 500k in some instances and even more than a million in T with 50x50 and 3 objects.

The instances are referred to as X-r-k where X is the instance class, r the resolution, and k the number of movable objects. The STRIPS planners are FF, FD with lazy evaluation, and LAMA [49, 45, 94], while the Functional STRIPS planners are FS0 [32], IW, and BFWS [72]. The initial and goal configurations of one of the problems is shown in Figure 4.1. We do not report empirical comparisons with other approaches to combined task and motion planning due to the lack of shared languages, benchmarks, and software. The planners that we use are all available, while our Functional STRIPS encodings, STRIPS translations, and videos illustrating the resulting plans can be found on the web

Problem				**FSTRIPS**			**STRIPS**	
Name	#C	R-O	O-O	#A	#Cs	#V	#A	#F
M-10-4	244	1.5k	896	58	15	10	4.3k	3k
M-10-5	244	1.5k	896	58	21	12	5.3k	3.1k
M-30-4	1.3k	86k	82.6k	58	15	10	33.5k	5.3k
M-30-5	1.3k	86k	82.6k	58	21	12	41.2k	6.2k
M-50-4	3.3k	553.4k	608k	58	15	10	–M–	
M-50-5	3.3k	553.4k	608k	58	21	12	–M–	
C1-10-7	166	1320	512	94	36	16	5.2k	3421
C1-30-7	1k	107k	66.4k	94	36	16	109k	7.5k
C1-50-7	2.7k	762k	536k	94	36	16	–M–	
C2-10-5	158	1.2k	996	70	21	12	3729	3209
C2-30-5	1k	94.1k	62.8k	70	21	12	56.8k	5.6k
C2-50-5	2.5k	656k	502.5k	70	21	12	–M–	
T-10-3	222	1.8k	704	64	10	11	4.3k	4.2k
T-30-3	1.4k	167.5k	92.4k	64	10	11	96.2k	6k
T-50-3	3.5k	1.2m	761.8	64	10	11	–M–	

Table 4.1: Data of the 15 planning instances. Columns show number of valid object configurations, configuration pairs that overlap (robot-object and object-object), ground actions, state constraints, and state variables. –M– means that STRIPS planners died at preprocessing before reporting the numbers.

Problem	FD			LAMA			FF		
Instance	L	E	T	L	E	T	L	E	T
M-10-4	19	20	0.08	19	41	0.04	20	24	0.0
M-10-5	20	22	0.1	20	37	0.04	21	32	0.1
M-30-4	39	46	3.4	39	59	2.82	57	464	11.73
M-30-5	44	55	2.36	44	68	3.74	75	2K	92.97
M-50-4		-M-			-M-			-M-	
M-50-5		-M-			-M-			-M-	
C-10-7	18	42	0.08	18	62	0.04	17	221	0.08
C-30-70	51	470	55.76	52	702	176.04		-T-	
C-50-7		-M-			-M-			-M-	
C2-10-5	9	16	0.06	10	22	0.02	11	33	0.01
C2-30-5	16	16	2.56	17	20	1.66	16	26	1.35
C2-50-5		-M-			-M-			-M-	
T-10-3	58	186	0.16	53	411	0.36	68	637	0.29
T-30-3		-M-			-M-			-M-	
T-50-3		-M-			-M-			-M-	

Table 4.2: Performance of STRIPS planners on the 15 instances from Table 4.1. The STRIPS planners are FD, FF, and LAMA. In most instances, the memory failure occurs at preprocessing.

`https://goo.gl/67Zibk`.

Table 4.1 shows data for the planning instances determined by the resolution parameters. As expected, the Functional STRIPS encodings are compact, while the STRIPS compilation tend to be large, involving in some cases thousand of atoms and tens of thousands of ground actions.

Aside from the resolution parameter and number of movable objects, the table shows the number of valid configurations for an object, and the number of robot-object and object-object configuration pairs that overlap in space. It then shows the number of ground actions, state constraints, and state variables in the Functional STRIPS encodings, and the number of ground actions and fluents in the STRIPS encodings. The symbol –M– is used to show cases where the compilation in STRIPS failed due to memory. As expected, the encoding in Functional STRIPS are compact, while those for STRIPS can be very large, resulting in thousand of atoms and tens of thousands of ground actions (more than 100k ground actions in C-30-7).

Table 4.2 shows the results of the three STRIPS planners and table 4.3 shows the results of the three FSTRIPS planners. For each instance, the columns show

Problem	**FS0**			**IW**			**BFWS**		
Instance	L	E	T	L	E	T	L	E	T
M-10-4	24	42	2.11	18	1.5k	0.076	24	645	0.01
M-10-5	21	36	2.43	18	1.2k	0.072	24	534	0.02
M-30-4		-T-		38	30k	1.9	38	14k	0.59
M-30-5		-T-		42	19k	1.73	52	12k	0.69
M-50-4		-T-		59	64k	5.6	74	30k	1.49
M-50-5		-T-		66	51k	5.08	79	32k	2.03
C-10-7	23	115	1.82	15	1.6k	0.12	15	1.2k	0.07
C-30-70	35	308	34.33	37	55k	5.95	37	39k	2.9
C-50-7		-T-			-T-			-T-	
C2-10-5	12	23	0.2	8	262	0.01	8	86	0.0
C2-30-5	35	308	35.52	16	2.5k	0.2	16	349	0.03
C2-50-5	124	2778	416.4	23	9.6k	1.02	23	1.3k	0.12
T-10-3	62	538	18.2	41	42k	1.81	38	37k	1.12
T-30-3		-T-		65	1.2m	80.7	75	224k	8.73
T-50-3		-T-			-M-			-M-	

Table 4.3: Performance of Functional STRIPS planners on the 15 instances from Table 4.1. The Functional STRIPS planners are FS0, IW, and BFWS. In most instances, the timeout occurs with the heuristic planner FS0. Width-based search planners performs well in most of the problem instances.

plan length, number of expanded nodes, and runtime in seconds. Time and memory outs shown as –T– and –M–. Time and memory bounds are 1800 seconds and 8GB respectively. The STRIPS planners solve 9 out of the 15 planning instances, with runtimes that do not go beyond 1 minute except in one case. This is quite remarkable as some of these problems, like C-30-7, involve a huge number of ground actions (more than 100k). There are no significant differences among the STRIPS planners, with FF missing one more instance than FD and LAMA (precisely C-30-7). It turns out that the Functional STRIPS planner FS0 doesn't do better: it solves 8 of the problems, the same as FF and one less than FD and LAMA. Since the heuristic of FS0 is informative but expensive, however, FS0 doesn't run out memory (8Gb) but out of time (1800 seconds). FS0 solves one instance that STRIPS planners do not: C2-50-5. *Width-based algorithms* IW and BFWS come on top: they solve almost all of the instances, 13 out of 15, in a few seconds, with the exception of T-30-3 that is solved by IW in 80 seconds. Moreover, the plans computed by IW tend also to be the shortest. Width-based algorithms expand many more nodes than the heuristic search algorithms, but they do so much faster. Actually, in these problems, STRIPS planners expand few nodes and the extra overhead in the computation of the heuristic in FS0 does not pay off.

4.9.1 Validation and Use of Plans

In a different set of experiments, using the robot simulator Gazebo with the physics on and a bump detector, we checked the *validity* of the plans. The encodings ensure that the resulting plans are collision-free *after* each of the actions in the plan but this doesn't ensure the absence of collisions *during* the execution of the actions. We thus wanted to test if, as expected, fine discretizations exclude collisions during the execution of actions, which we call internal collisions, as for fine discretizations internal collisions result in "external" collisions; i.e., collisions at the end of the action execution.

Focusing on the plans obtained by IW and BFWS only, there are 28 plans to check as there are 5 task and motion problems and 3 resolutions, which results in 15 problems for each of the two algorithms, with one problem, T-50-3, not solved by either one. Interestingly, the resolutions 10x10 and 30x30 result in plans that produce an internal collision at some point; indeed, 9 of the 10 plans for the 10x10 resolution and 7 of the 10 plans for the 30x30 resolution feature at least one such collision during an action execution. On the other hand, for the 50x50 resolution just 1 out of the 9 plans found by IW and BFWS was invalid in this sense. This means that fine space resolutions can be used to exclude *all collisions* and that plans for such resolutions can actually be computed, even if the combinatorics is very high.

Actually, from a computational point of view, it is not necessary or convenient to use the same resolution over the whole space to ensure collision-free plans. One can apply a *plan-simulate-refine-replan* cycle where the resolution is increased over the areas of the space where the simulations detect internal collisions until no such collisions are found. It is important to emphasize that this iterative approach is different than both hierarchical approaches that decouple task and geometry planning, and replanning approaches where failed simulated executions are used to revise the symbolic task model [100].

The *key difference* is that our scheme takes geometry into account while computing the plans. The need for replanning is not for checking geometrical feasibility but to make the geometrical reasoning sound by increasing the resolution over the relevant parts of the configuration space.

4.10 Discussion

We have proposed an alternative integration of task and motion planning where the symbolic and geometrical components are addressed in combination, with neither part taking the back seat. For this, we have built on an expressive planning language, Functional STRIPS, that supports constraints, functions, and numerical variables, and on the planner `FS0`, which supports a large fragment of this language in the specification of problems and is crucially able to exploit its expressivity in the computation of heuristics.

We have extended this language and computational model with state constraints: logical formulas that must hold true in every state of a plan. In order to address motion and task planning problems, we use *functions* for encoding the geometrical dimensions of objects and their poses, and *state constraints* to express that no two objects, including the robot, can overlap in space. The experiments reported are preliminary but illustrate the feasibility of the approach.

There is a lot of room for improving performance and for exploring the possibilities that are afforded by this integration of motion planning into task planning. In particular, scaling up well in the presence of large grids and many objects remains a challenge. In principle, however, there is no need for the grids to be regular: it'd be more natural to use higher resolutions around the current robot configuration and lower resolutions elsewhere. Alternatively, maps obtained from random configuration sampling, as in probabilistic roadmaps, could be used instead. The strength of the integration proposed is that it is very general and independent of these choices.

We have shown that existing STRIPS and Functional STRIPS planners can be used for solving problems that combine task and motion planning. We used the STRIPS planners off-the-shelf, and introduced a domain-independent extension of Functional STRIPS for managing state constraints that are used to prevent collisions. The STRIPS encodings are obtained from a simple compilation.

The problems considered are not fully realistic but are far from trivial. The state space in some of the problems is huge. With a 50x50 discretization, there are 2500 cells, and $8 \times 2500 = 20,000$ object configurations which in some problems result in more than 10^{25} states. These are problems that are solved by width-based algorithms in a few seconds. While the approach, does not exclude the possibility of collisions during the execution of actions, as opposed to collisions at the end, we have seen that such collisions can be avoided by using fine discretizations, and used a robot simulator to prove that only 1 of the 9 plans found by either IW or BFWS for the 50x50 resolution was invalid in this sense. We discussed also ways for using the approach in a *plan-simulate-refine-replan* cycle to create non-uniform discretizations.

CHAPTER 5

Pick and Place Tasks in 3D

5.1 Introduction

Planning problems in robotics involve robots that move around, while manipulating objects and avoiding collisions. These problems are thought to be outside the scope of standard AI planners, and are normally addressed through a combination of two types of planners: task planners that handle the high-level, symbolic reasoning part, and motion planners that handle motion and geometrical constraints [11, 41, 10, 105, 78, 53]. These two components, however, are not independent, and hence, by giving one of the two planners a secondary role in the search for plans, approaches based on task and motion decomposition tend to be ineffective and result in lots of backtracks [65].

In recent years, there have been proposals aimed at addressing this combinatorial problem by exploiting the efficiency of modern classical AI planners. In one case, the spatial constraints are taken into account as part of a goal-directed replanning process where optimistic assumptions about free space are incrementally refined until plans are obtained that can be executed in the real environment [100]. In another approach [35], geometrical information is used to update the heuristic used in the FF planner [49]. Other recent recent approaches appeal instead to SMT solvers suitable for addressing both task planning and the geometrical constraints [85, 14].

The work done in this chapter aims at exploiting the efficiency of modern classical AI planning algorithms but departs from prior work in two ways. First, task and motion problems are *fully compiled* into classical planning problems so that the classical plans are valid robot plans. Motion planners and collision checkers [68] are used in the compilation but not in the solution of the classical problem. The compilation is thus sound, and probabilistically complete in the sense that robot plans map into classical plans provided that the number of sampled robot configurations is sufficient. In order to make the compiled problems compact, we move away from the standard PDDL planning language and appeal instead to Functional STRIPS [37], a planning language that is expressive enough to accommodate *procedures* and *state constraints*. State constraints are formulas that are forced to be true in every reachable state, and thus represent implicit action preconditions. In the CTMP planning encoding, state constraints are used to rule out spatial overlaps. Procedures are used in turn for testing and updating robot and object configurations, and their planning-time execution is made efficient by precompiling suitable tables. The size and computation of these tables is also efficient, and allows us to deal with 3D scenarios involving tens of objects and a PR2 robot simulated in Gazebo [59].

The second departure from prior work is in the classical planning algorithm itself. Previous approaches have built upon classical planners such as FF and LAMA [49, 94], yet such planners cannot be used with expressive planning languages that feature functions and state constraints. The Functional STRIPS planner FS [32] handles functions and can derive and use heuristics, yet these heuristics are expensive to compute and not always cost-effective to deal with state constraints. For these reasons, we build instead on a different class of planning algorithm, called best-first width search (BFWS), that has been recently shown to produce state-of-the-art results over classical planning benchmarks [74]. An advantage of BFWS is that it relies primarily on exploratory novelty-based measures, extended with simple goal-directed heuristics. For this work, we adapt BFWS to work with Functional STRIPS with state constraints, replacing a Functional STRIPS heuristic that is expensive and does not take state constraints into account by a fast and simple heuristic suited to *Pick-and-Place* tasks.

Given that classical AI planning is planning over finite and discrete state spaces with a known initial state, deterministic actions, and a goal state to be reached [38], it is not surprising that the combined task and motion planning can be fully compiled into a classical planning problem once the continuous configuration space is suitably discretized or sampled [68]. Moreover, modern classical planners scale up very well and like SAT or SMT solvers are largely unaffected by the size of the state space. If this approach has not been taken before, it is thus not due to the lack of efficiency of such planners but due to the limitations of the languages that they support [83]. Indeed, there is no way to compile non-overlap physical constraints into PDDL in compact form. We address this limitation by using a target language for the compilation that makes use of *state constraints* to rule out physical overlaps during motions, and *procedures* for testing and updating physical configurations. This additional expressive power prevents the use of standard heuristic search planning algorithms [49, 94] but is compatible with a more recent class of width-based planning methods that are competitive with state-of-the-art heuristic search approaches [75, 74].

5.2 Compiling Abstract Model in Functional STRIPS

The planning encoding shown in Fig. 5.1 assumes a crucial preprocessing stage where the base and arm graphs are computed, and suitable tables are stored for avoiding the use of motion planners and collision checkers during planning time. This preprocessing is efficient and does not depend on the number of objects, meaning it can be used for several problem variations without having to call collision checkers and motion planners again. Indeed, except for the overlap tables, the rest of the compilation is local and does not depend on the possible robot base configurations at all.

To achieve this, we consider the robot at a *virtual base* $B_0 = \langle x, y, \theta \rangle$ with $x = y = \theta = 0$ in front of a *virtual table* whose height is the height of the actual tables, and whose dimensions exceed the (local) space that the robot can reach without moving the base. By considering the robot acting in this local virtual space without moving from this virtual base B_0, we will obtain all the relevant information about object configurations and arm trajectories, that will carry to the real robot base configurations B through a simple linear transformations that depend on B. The computation of the overlap tables is more subtle and will be considered later.

First of all, the x, y space of the virtual table is discretized regularly into D position pairs x_i, y_i. If the height of the objects is h' and the height of the tables is h, then the virtual object configurations are set to the triplets $\langle x_i, y_i, z \rangle$ where $z = h + h'/2$. Each virtual object configuration represents a possible center of mass for the objects when sitting at location x_i, y_i over the virtual table. For each such configuration $C = \langle x_i, y_i, z \rangle$, k grasping poses A^j_C are defined from which an object at $\langle x_i, y_i, z \rangle$ could be grasped, and a motion planner (MoveIt) is called to compute k' arm trajectories for reaching each such grasping pose A^j_c through k' different waypoints from a fixed resting pose and the robot base fixed at B_0. This means that up to $k \times k'$ arm trajectories are computed for each virtual object configuration, resulting in up to $D \times k \times k'$ arm trajectories in total and up to $k \times D$ grasping poses. For each reachable grasping pose A^j_C, we store the pair $\langle A^j_C, C \rangle$

in a hash table. The table captures the function $vplace$ that maps grasping poses (called arm configurations here), into virtual object configurations. The meaning of $vplace(A) = C$ is that when the robot base is at B_0 in front the virtual table and the arm configuration is A, an object on the gripper will be placed at the virtual object configuration C.

The *arm graph* has as nodes the arm configurations A that represent reachable grasping poses $A = A^j_C$ in relation to some virtual object configuration C, in addition to the resting arm configuration. The arm trajectories that connect the resting arm configuration A_0 with an arm trajectory A provide the edge in the arm graph between A_0 and A. The graph contains also the inverse edges that correspond to the same trajectories reversed. Grasping configurations that are not reachable with any trajectory from the resting arm configuration are pruned and virtual object configurations all of whose grasping poses have been pruned, are pruned as well.

The *base graph* is computed by sampling a number of configurations N_B near the tables and calling the MoveIt motion planner to connect each such configuration with up to k_B of its closest neighbours. The number of *robot configurations* results from the product of the number of arm configurations $k \times D$ and the number of base configurations N_B. In the experiments we consider numbers that go from tens to a few hundred and which thus result into thousands of possible robot configurations. The computation of the base and arm graphs defines the procedures used in the $MoveBase$ and $MoveArm$ actions that access the source and target configuration of each graph edge.

The set of (real) *object configurations* are then defined and computed as follows. The virtual object configuration $C = \langle x, y, z\rangle$ represents the 3D position of the object before a pick up or after a place action, with the arm at configuration A and the robot base at the virtual base configuration $B_0 = \langle 0, 0, 0\rangle$. As the robot moves from this "virtual" base to an arbitrary base B in the base graph, the point C determined by the same arm configuration A moves to a new point C' that is given by a transformation $T_B(C)$ of C that depends solely on B. Indeed, if $B = B_0 + \langle \Delta_X, \Delta_Y, \Delta_\theta\rangle$ with $\Delta_\theta = 0$, then $T_B(C) = \langle x + \Delta_X, y + \Delta_Y, z\rangle$. More generally, for any Δ_θ:

$$T_B(C) = \langle x', y', z\rangle. \tag{5.1}$$

Where x' and y' are computed as follows:

$$x' = \Delta_X + (x - \Delta_X)cos(\Delta_\theta) - (y - \Delta_Y)sin(\Delta_\theta). \tag{5.2}$$

$$y' = \Delta_Y + (x - \Delta_X)sin(\Delta_\theta) + (y - \Delta_Y)cos(\Delta_\theta). \tag{5.3}$$

The set of *actual object configurations* is then given by such triplets $T_B(C) = \langle x', y', z \rangle$ for which 1) B is a node of the base graph, 2) C is a virtual object configuration, and 3) the 2D point x', y' falls within a table in the actual environment. That is, while the virtual object configurations live only in the virtual table with the base fixed at B_0, the actual object configurations depend on the virtual object configurations, the base configurations, and the real tables in the working space. We will write $T_B(C) = \perp$ when C and B are such that for $T_B(C) = \langle x', y', z' \rangle$, the 2D point x', y' does not fall within a table in the actual environment. In such a case, $T_B(C)$ doesn't denote an actual object configuration.

Given the linear transformation T_B and the function $vpose(A)$ defined above, that maps an arm configuration into a virtual object configuration that is relative to the virtual base B_0, the procedures denoted by the symbols $@graspable$, $@placeable$, and $@pose$ in the planning encoding are defined as follows:

$$\begin{aligned} @pose(B, A) = C' &\quad \text{iff } C' = T_B(vpose(A)) \\ @graspable(B, A, C') = true &\quad \text{iff } C' = @pose(B, A) \\ @placeable(B, A) = true &\quad \text{iff } @pose(B, A) \neq \perp . \end{aligned}$$

We are left to specify the compilation of the tables required for computing the $@nonoverlap$ procedure without calling a collision checker at planning time. This procedure is used in the state constraints *@nonoverlap(B,Traj,Conf(o),Hold))* for ruling out actions that move the arm along a trajectory $Traj$ such that for the current base configuration B and content of the gripper $Hold$, will cause a collision with some object o in its current configuration $Conf(o)$. For doing these tests at planning time efficient, we precompile two additional tables, called the holding and non-holding overlap tables (HT, NT), which are made of pairs $\langle Tr, C \rangle$ where Tr is a trajectory in the arm graph, and C is what we will call a *relative object configuration* different than both the virtual and real object configuration. Indeed, the set of relative object configurations is defined as the set of configurations $T_B^{-1}(C)$ for all bases B and all real object configurations C, where T_B^{-1} is the inverse of the linear transformation T_B above. If C is a real 3D point obtained by mapping a point C' in the virtual table after the robot base changes from B_0 to B, then $C'' = T_{B'}(C)$ for $B' = B$ is just C' but for $B' \neq B$, it denotes a point in the "virtual" space relative to the base B_0 that may not correspond to a virtual object configuration, and may even fail to be in the space of the virtual table (the local space of the robot when fixed at base B_0). Relative object configurations C'' that do not fall within the virtual table, are pruned. The *holding overlap table (HT)* contains then the pair $\langle Tr, C\rangle$ for a trajectory Tr and a relative object configuration, iff the robot arm moving along trajectory Tr will collide with an object in the virtual configuration C when the robot base is at B_0 and the gripper is carrying an object. Similarly, the pair $\langle Tr, C\rangle$ belongs to the *non-holding overlap table (NT)* iff the same condition arises when the gripper is empty. Interestingly, each of these two tables is compiled by calling a collision checker (MoveIt) a number of times that is given by the total number of arm trajectories. Indeed, for each trajectory T, the collision checker tests in one single scan which relative configurations C are on the way.

The procedure *@nonoverlap(B,Tr,Conf(o),Hold)* checks whether trajectory Tr collides with object o in configuration $Conf(o)$ when the robot base is B. If $Hold$ is $None$, this is checked by testing whether the pair $\langle Tr, T_B^{-1}(Conf(o))\rangle$ is in the NT table, and if $Hold$ is not $None$, by testing whether the pair is in the HT table. These are lookup operations in the two (hash) tables NT and HT, whose size is determined by the number of trajectories and the number of relative object configurations. This last number is independent of the number of objects but higher than the number of virtual configurations. In the worst case, it is bounded by the product of the number N_B of robot bases and the number of real object configurations, which in turn is bounded by $N_B \times N_C$, where N_C is the number of virtual object configurations. Usually, however, the number of entries in the overlap tables NT and HT is much less, as for most real object configurations C and base B, the point $T_B^{-1}(C)$ does not fall into the "virtual table" that defines the local space of the robot when fixed at B_0. The size of the hash table $\langle Tr, C\rangle$ precompiled for encoding the function $vpose(Tr)$ above is smaller and given just by the number of arm trajectories Tr, to the number of edges in the arm graph, which in turn is equal to $2 \times D \times k \times k'$, where D is the number of virtual object configurations, k is the number of grasping poses for each virtual object configuration, and k' in the number of trajectories for reaching each grasping pose.

5.3 Planning for Pick-and-Place Tasks: Modeling and Computation

We consider CTMP problems involving a robot and a number of objects located on tables of the same height. The tasks involve moving some objects from some initial configuration to a final configuration or set of configurations, which may require moving obstructing objects as well. The model is tailored to a PR2 robot using a single arm, but can be generalized easily.

The main state variables *Base*, *Arm*, and *Hold* denote the configuration of the robot base, the arm configuration, and the content of the gripper, if any. In addition, for each object o, the state variable *Conf(o)* denotes the configuration of object o. The configuration of the robot base represents the 2D position of the base and its orientation angle. The configuration of the robot arm represents the configuration of the end effector: its 3D position, pitch, roll, and yaw. Finally, object configurations are 3D positions, as for simplicity we consider object that are symmetric, and hence their orientation angle is not relevant. There is also a state variable $Traj$, encoding the last trajectory followed by the robot arm, which is needed for checking collisions during arm motions. All configurations and trajectories are obtained from a *preprocessing stage*, described in the next section, and are represented in the planning encoding by symbolic ids. When plans are executed, trajectory ids become motion plans; i.e. precompiled sequences of base and arm join vectors, not represented explicitly in the planning problem.

The encoding assumes *two finite graphs*: a *base graph*, where the nodes stand for robot base configurations and edges stand for trajectories among pairs of base configurations, and an *arm graph*, where nodes stand for end-effector configurations (relative to a fixed base), and edges stand for arm trajectories among pairs of such configurations. The details for how such graphs are generated are not relevant for the planning encoding and will be described below. As a reference, we will consider instances with tens of objects, and base and arm graphs with hundreds of configurations each, representing thousands of robot configurations.

A fragment of the planning encoding featuring all the actions and the state constraints is shown in Figure 5.1. Actions $MoveBase(e)$ take an edge e from the base graph as an argument, and update the base configuration of the robot to the target configuration associated with the edge. The precondition is that the source configuration of the edge corresponds to the current base configuration, and that the arm is the resting configuration *ca0*. Actions $MoveArm(t)$ work in the same way, but the edges t of the arm graph are used instead.

```
(:action MoveBase
 :parameters (?e - base-graph-traj-id)
 :prec (and (= Arm ca0)
            (= Base (@source-b ?e))
 :eff (and (:= Base (@target-b ?e))))

(:action MoveArm
 :parameters (?t - arm-graph-traj-id)
 :prec (and (= Arm (@source-a ?t))
 :eff (and (:= Arm (@target-a ?t))
           (:= Traj  ?t)))

(:action Grasp
 :parameters (?o - object-id)
 :prec (and  (=  Hold  None)
  (@graspable Base Arm (Conf ?o)))
 :eff (and (:= Hold ?o)
           (:= (Conf ?o) c-held)))

(:action Place
 :parameters (?o - object-id)
 :prec  (and (= Hold ?o)
    (@placeable Base Arm)
 :eff (and (:= Hold None)
   (:= (Conf ?o)(@place Base Arm)))

(:state-constraint
  :parameter (?o - object-id)
   (@non-overlap Base Traj (Conf ?o) Hold))
```

Figure 5.1: CTMP Model Fragment in Functional STRIPS: Action and state constraint schemas. Abbreviations used. Symbols preceded by "@" denote procedures. All objects assumed to have the same shape. Initial situation provides initial values for the state variables *Base*, *Arm* (resting), $Traj$ (dummy), and $Conf(o)$ for each object. Goals describe target object configurations. State constraints prevent collisions during arm motions. Motion planners and collision checkers used at compilation time, not at plan time, as detailed in the Preprocessing section.

There are also actions *Grasp(o)* and *Place(o)* for grasping and placing objects o. *Grasp(o)* requires that the gripper is empty and *@graspable(Base,Arm,Conf(o))* is true, where the procedure denoted by the symbol *@graspable* checks if the robot configuration, as determined by its base and (relative) arm configuration, is such that object o in its current configuration can be grasped by just closing the gripper. Likewise, the atoms $Hold = o$ and *@placeable(Base,Arm,Conf(o))* are preconditions of the action *Place(o)* .

The total number of ground actions is given by the *sum* of the number of edges in the two graphs and the number of objects. This small number of actions is made possible by the planning language where robot, arm, and object configurations do not appear as action arguments. The opposite would be true in a STRIPS encoding where action effects are determined solely by the action (add and delete lists) and do not depend on the state. The number of state variables is also small, namely, one state variable for each object and four other state variables. Atoms whose predicate symbols denote procedures, like *@graspable(Base,Arm,Conf(o))*, do not represent state variables or fluents, as the denotation of such predicates is fixed and constant. These procedures play a key role in the encoding, and in the next section we look at the preprocessing that converts such procedures into fast lookup operations.

The only subtle aspect of the encoding is in the state constraints used to prevent collisions. Collisions are to be avoided not just at beginning and end of actions, but also during action execution. For simplicity, we assume that robot-base moves do not cause collisions (with mobile objects), and hence that collisions result exclusively from arm motions. We enforce this by restricting the mobile objects to be on top of tables that are fixed, and by requiring the arm to be in a suitable resting configuration (ca0) when the robot base moves. There is one state constraint *@nonoverlap(Base,Traj,Conf(o),Hold)* for each object o, where $Traj$ is the state variable that keeps track of the last arm trajectory executed by the robot. The procedure denoted by the symbol *@nonoverlap* tests whether a collision occurs between object o in configuration *Conf(o)* when the robot arm moves along the trajectory $Traj$ and the robot base configuration is *Base*. The test depends also on whether the gripper is holding an object or not. As we will show in the next section, this procedure is also computed from two *overlap tables* that are precompiled by calling the MoveIt collision-checker [102] a number of times that is twice the

5.4 Planning Algorithm

The compilation of task and motion planning problems is efficient and results in planning problems that are compact. Yet, on the one hand, standard planners like FF and LAMA do not handle functions and state constraints, while planners that do compute heuristics that in this setting are not cost-effective [32]. For these reasons, we build instead on a different class of planning algorithm, called best-first width search (BFWS), that combines some of the benefits of the goal-directed heuristic search with those of width-based search [72].

Width-based algorithms such as IW and SIW do not require PDDL-like planning models and can work directly with simulators, and thus unlike heuristic search planning algorithms, can be easily adapted to work with Functional STRIPS with state constraints. The problem is that by themselves, IW and SIW are not powerful enough for solving large CTMP problems. For such problems it is necessary to complement the effective exploration that comes from width-based search with the guidance that results from goal-directed heuristics. For this reason, we appeal to a combination of heuristic and width-based search called Best-First Width Search (BFWS), that has been shown recently to yield state-of-the-art results over the classical planning benchmarks [74]. BFWS is a standard best-first search with a sequence of evaluation functions $f = \langle h, h_1, \ldots, h_n \rangle$ where the node that is selected for expansion from the OPEN list at each iteration is the node that minimizes h, using the other h_i functions lexicographically for breaking ties. In the best performing variants of BFWS, the main function $h = w$ computes the "novelty" of the nodes, while the other functions h_i take the goal into account.

For our compiled CTMP domain, we use BFWS with an evaluation function $f = \langle w, h_1, \ldots, h_n \rangle$, where w stands for a standard novelty measure, and $h_1, \ldots, h_n$ are simple heuristic counters defined for this particular domain. The novelty w is defined as in [74]; namely, the novelty $w(s)$ of a newly generated state s in the BFWS guided by the function $f = \langle w, h_1, \ldots, h_n \rangle$ is i iff there is a tuple (conjunction) of i atoms $X = x$, and no tuple of smaller size, that is true in s but false in all the states s' generated before s with the same function values $h_1(s') = h_1(s)$, $\ldots$, and $h_n(s') = h_n(s)$. According to this definition, for example, a new state s has novelty 1 if there is an atom $X = x$ that is true in s but false in all the states s' generated before s where $h_i(s') = h_i(s)$ for all i.

For the tie-breaking functions h_i we consider three counters. The first is the standard goal counter $\#g$ where $\#g(s)$ stands for the number of goal atoms that are not true in s. The second is an slightly richer goal counter h_M that takes into account that each object that has to be moved to a goal destination has to involve two actions at least: one for picking up the object, and one for placing the object. Thus $h_M(s)$ stands for twice the number of objects that are not in their goal configurations in s, minus 1 in case that one such object is being held. The last tie-breaker used corresponds to the counter $\#c(s)$ that tracks the number of objects that are in "obstructing configurations" in the state s. This measure is determined from a set $\mathcal{C}$ of object configurations C computed once from the initial problem state, as it is common in landmark heuristics. The count $\#c(s)$ is i if there are i objects o for which the state variable $Conf(o)$ has a value in s that is in $\mathcal{C}$. The intuition is that a configuration is "obstructing" if it's on the way of an arm trajectory that follows

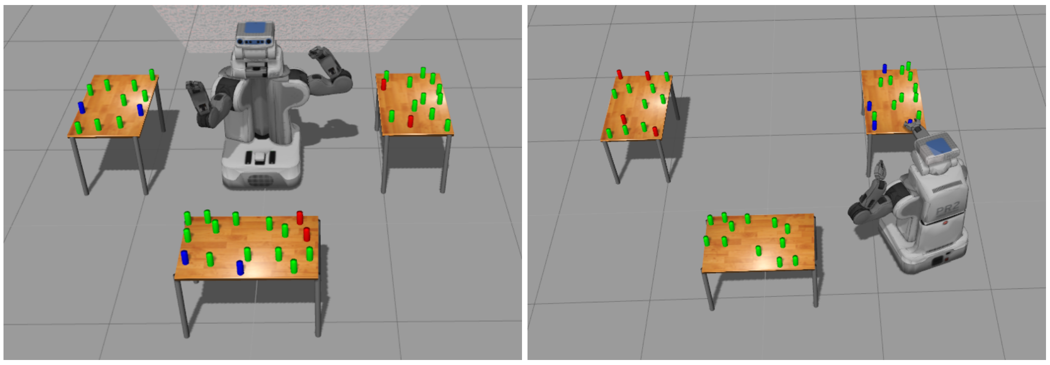


Figure 5.2: Manipulating objects in a 3-table environment, initial (left) and goal (right) situations. The objective is to put the blue objects on the rightmost table and the red objects on the leftmost table.

a suitable relaxed plan for achieving a goal atom. More precisely, we use a single IW(2) call at preprocessing for computing a plan for each goal atom in a problem relaxation that ignores state constraints (i.e., collisions). These relaxed problems are "easy" as they just involve robot motions to pick up the goal object followed by a pick up action, more robot motions, and a place action. The search tree constructed by IW(2) normally includes a plan for each goal atom in this relaxation, and often more than one plan. One such *relaxed plan* "collides" with an object o if a $MoveArm(t)$ action in the plan leads to a state where a state constraint *@nonoverlap(Base,Arm,Conf(o),Hold)* is violated (this is possible because of the relaxation). In the presence of multiple plans for an atomic goal in the relaxation, a plan is selected that collides with a minimum number of objects. For such an atomic goal, the "obstructing configurations" are the real object configurations C such that a state constraint *@nonoverlap(Base,Arm,C,Hold)* is violated in some state of the relaxed plan where $Conf(o) = C$ for some object o. We further consider as obstructing those configurations that in a similar manner obstruct the achievement of the goal of holding any object o that is in an obstructing configuration *in the initial state*, recursively and up to a fixpoint.

The set $\mathcal{C}$ is then a union of the sets of "obstructing configurations" for each atomic goal, and $\#c(s)$ is the number of objects o for which the value C of the state variable $Conf(o)$ in s belongs to $\mathcal{C}$. Note that unlike the other two heuristics $\#g$ and h_M, which must have value zero in the goal, the $\#c(s)$ counter may be different than zero in the goal. Indeed, if a problem involves exchanging the configuration of two objects, $\#c(s)$ will be equal to 2 in the goal, as the two goal configurations are actually obstructing configurations as determined from the initial state. The set $\mathcal{C}$ of obstructing configurations is computed once from the initial state in low polynomial time by calling the IW(2) procedure once. The resulting $\#c(s)$ count provides an heuristic estimate of the number of objects that need to be removed in order to achieve the goal, a version of the minimum constraint removal problem [43] mentioned in [35].

The counters h_M and $\#c$ used in the BFWS algorithm for CTMP planning can be justified on domain-independent grounds. Indeed, h_M corresponds roughly to the cost of a problem where both state constraints and preconditions involving procedures have been relaxed. So the plans for the relaxation are sequences of pickup and place actions involving the goal objects only. The counter $\#c$ is related to landmark heuristics under the assumption that the goals will be achieved through certain motion plans.

The third element in the BFWS algorithm is the extension of the problem states with two extra Boolean features *graspable*(o)* and *placeable*(o)* associated with each object o. The features *graspable*(o)* and *placeable*(o)* are set to true in a state s iff the preconditions of the actions *Grasp(o)* and *Place(o)* are true in s respectively. These features are needed as there are no state variables related to the preconditions *(@graspable B A Conf(o))* and *(@placeable B A)* of those actions, as the predicate symbols of these atoms denote procedures. That is, the terms B, A, and $Conf(o)$ in these atoms denote state variables but the relations themselves, denoted by the symbols *@graspable* and *@placeable*, are static.

Finally, for the experimental results we have found useful to add an extra precondition to the action $MoveArm(t)$. This precondition requires that *@target-a(t)* is the resting configuration *ca0* or that *@placeable(Base,@target-a(t))* is true. In other words, the arm is moved from the resting position to configurations where an object could be picked up or placed. This restriction reduces the average branching factor of the planning problem, in particular when the number of arm motions in the arm graph is large.

5.5 Experimental Results

We test our model on two environments having one and three tables, the characteristics of which are shown in Table 5.1. As explained above, the virtual space of the robot is discretized into $D = 15$ position pairs or *virtual configurations*, with $k = 4$ grasping poses per virtual configuration and $k' = 4$ arm trajectories for each of those grasping poses, obtained from MoveIt!. Thus, the maximum number of (virtual) grasping poses will be $D \times k = 60$, of which those for which no motion

compilation	#traj.	#arms	#bases	#confs	#virt.	#GP	#rel.	#real	T(s)
1-table	268	43	124	5332	15	42	1081	136	5
3-tables	268	43	323	13889	15	42	3379	393	13

Table 5.1: *Compilation data for one and three tables.* Columns show the number of tables, total number of arm trajectories, arms configurations, base configurations, total number of robot configurations, virtual object configurations, number of virtual grasping poses, relative object configurations, total number of real object configurations and overall compilation time.

plan is found get pruned. In our benchmark environments, the total number of virtual grasping poses is 42. In turn, the maximum number of arm trajectories is $D \times k \times k' = 240$ in each direction, i.e. 480, while in both of our environments we have a total of 268 such trajectories, since again no feasible motion plans are found for the rest.

The number of sampled bases is 124 for the one-table environment and 323 for the three-table environment, while each robot base in the base graph is connected to a maximum of 12 closest base configurations.

Importantly, the output of the precompilation phase, which takes 5 min. (13 min.) for the one-table (three-tables) environment, is valid for for all instances with that number of tables, regardless of number of objects, initial robot and object configurations, and particular goals of the problem.

For each environment, we generate a number of semi-random instances with increasing number of objects, ranging from 10 to 40, and increasing number of goals, ranging from 2 to 8, where a problem with e.g. 4 goals might require that 4 different objects be placed in their respective, given target configurations.The initial and goal states of a sample problem instance are shown in Fig. 5.2, where the robot needs to place all blue objects in one table and all red objects in another. Tables 5.2 and 5.3 show the results of the BWFS planner on each generated instance, running with a maximum of 30 minutes and 8GB of memory on an AMD Opteron 6300@2.4Ghz. Each row shows results for one instance. Three leftmost columns report instance characteristics. *#o* denotes number of objects on the table, *#g* number of different goals, *#c* is a proxy for the number of objects that initially obstruct the achievement of the goal, as described in the text. Remaining columns report length of the computed plan (*L*), number of nodes expanded during the search (*E*), and, in seconds, preprocessing, search and total time. `TO` and `MO` denote time- and memory-outs. The planner uses ROS [91], Gazebo [59], and MoveIt [102], in the preprocessing and in the simulations, but not at planning time. Videos showing the execution of the computed plans in the Gazebo simulator, for some selected instances, can be found in `https://goo.gl/67Zibk`, as well as encodings.

#objects	#goals	#c	Length	Expanded	Prep.(s.)	Search(s.)	Total(s.)
10	1	4	38	700	2.4	0.08	2.48
10	2	6	67	5.7k	2.42	0.64	3.06
10	3	8	73	6.1k	2.22	0.72	2.94
15	1	6	49	778	3.4	0.1	3.5
15	2	8	81	9.8k	3.76	1.27	5.03
15	3	10	80	7.7k	4.13	0.97	5.1
20	1	12	86	39k	5.44	4.46	9.9
20	2	14	122	63.3k	5.85	9.42	15.27
20	3	22	159	49.2k	5.66	7.26	12.92
25	1	4	22	206	7.42	0.03	7.45
25	2	4	45	39.1k	7.29	5.54	12.83
25	3	18	MO	-	-	-	-
30	1	4	22	67.6k	9.21	10.16	19.37
30	2	38	MO	-	-	-	-
30	3	38	TO	-	-	-	-

Table 5.2: *Per-instance results* of *Pick-and-Place* problem for one table.

The results show that our approach is competitive and scales well with the number of objects in the table. The length of the obtained plans ranges from 22 to 220 steps. Problems with up to 20 objects, both for one and three tables, for example, are solved in a few seconds and requiring only the expansion of a few thousands of nodes in the search tree.

Problems with a up to 30 and even 40 objects are solved with relative ease in the environment with three tables, but as expected become much harder when we have one single table, because the objects clutter almost all available space, making it harder for the arm robot to move collision-free. Indeed, the results show that the key parameter for scalability is #c, which in a sense indicates how cluttered the space is in the initial situation. When this number is not too high, as in the three-table environment, our approach scales up with relative ease with the number of different specified goals.

Finally, preprocessing times scale up linearly with the number of objects, regardless of the number of goals, thanks to the low-polynomial cost of the IW(2) pass on which the preprocessing is based, as detailed above.

#objects	#goals	#c	Length	Expanded	Prep.(s.)	Search(s.)	Total(s.)
10	2	6	54	1.3k	8.1	0.23	8.33
10	4	2	101	3.9k	8.1	0.8	8.9
10	6	2	121	3.9k	7.18	0.6	7.78
10	8	2	150	4.5k	8.26	0.91	9.17
20	2	4	65	6.2k	19.19	1.32	20.51
20	4	4	89	9.6k	17.9	2.29	20.19
20	6	6	130	3.1k	17.66	0.73	18.39
20	8	8	141	5.9k	18.42	1.26	19.68
25	2	8	46	1.1k	23.74	0.23	23.97
25	4	8	80	2.3	24.44	0.54	24.98
25	6	10	120	3.5k	27.04	0.91	27.95
25	8	12	158	3.4k	23.74	0.69	24.43
30	2	4	MO	-	-	-	-
30	4	2	74	1.6k	30.37	0.4	30.77
30	6	8	123	2.6k	30.09	0.64	30.73
30	8	10	161	3.5k	32.22	0.86	33.08
40	2	4	52	1.3k	45.64	0.33	45.97
40	4	14	114	55.5k	45.65	13.12	58.77
40	6	10	178	166k	47	41.36	88.36
40	8	14	220	201k	46.46	55.57	102.03

Table 5.3: Per-instance results of *Pick-and-Place* problem for three tables.

5.6 Discussion

The presented work is closest to [34, 100]. What distinguishes our approach is that combined task and motion planning problems are fully mapped into classical AI planning problems encoded in an expressive planning language. Motion planners and collision checkers are used at compile time but not at planning time. The approach is sound (classical plans map into valid executable robot plans) and probabilistically complete (with a sufficient number of configurations sampled, robot plans have a corresponding classical plan). For the approach to be effective, three elements are essential. First, an expressive planning language that supports functions and state constraints. Second, a width-based planning algorithm that can plan effectively for models expressed in such a language without requiring the use of accurate but expensive heuristic estimators. Third, a preprocessing stage that computes the finite graphs of robot bases and arm configurations, the possible object configurations, and the tables that allow us to resolve procedural calls into efficient table lookups. We have shown that the compilation process is efficient and independent of the number of objects, that the compiled problems are compact, and that the planning algorithm can generate long plans effectively.

For the experiments, we have considered the type of pick and place problems that have been used in recent work [34, 100]. For these problems, it is sufficient to sample robot base configurations that are close to the physical tables, and arm trajectories that can pick up and place objects in the local space of the robot at a height that corresponds to the height of the tables. This part of the problem is not modeled explicitly in the Functional STRIPS planning encodings, which implicitly assume a finite graph of robot bases and one of robot arm configurations computed at preprocessing. In the future, we want to represent this information explicitly in the planning encoding so that the preprocessing stage can be fully general and automatic. This requires a general representation language for CTMP problems so that the compilation will be a mapping between one formal language and another. Unfortunately, there are no widely accepted and shared formal models and languages for CTMP, which makes it difficult to compare approaches empirically or to organize " CTMP competitions", that in the case of AI planning or SAT solving have been an essential ingredient for progress. We believe that Functional STRIPS can actually serve both roles: as the basis for a general, integrated representation language for CTMP problems and as a convenient target language of the compilation representations. This work is a first step towards this goal where we have shown that the compilation is indeed feasible and effective both representationally and computationally.

PART III

Generality on Combined Task and Motion Planning

CHAPTER **6**

Flexible Algorithms for Combined Task and Motion Planning

6.1 Introduction

On chapter 5 we presented an approach to fully compile CTMP problems into classical planning problems. We have also shown how to model *Pick-and-Place* problems in FSTRIPS using state constraints to avoid collisions. Lastly, we presented a planning algorithm to solve these problems that uses a BFWS search algorithm with an evaluation function $f = \langle h, h_1, \ldots, h_n \rangle$ where tie breaking is done through three different counters: $\#g$, h_M and $\#c$, where h_M and $\#c$ are not fully domain-independent. Although the experiments have shown good results, this approach has some limitations which we address one by one:

1. There is no explicit an specific language for specifying CTMP problems.
2. We have only shown how to model a *Pick-and-Place* problem in FSTRIPS with state constraints.
3. The instances that we reported have only one type of shape: a cylinder.
4. The planning algorithm is not completely general and not fully domain independent. It relies in the computation of some domain-dependent heuristics.

Broadly, there have been several efforts to efficiently combine task and motion planning. However, to the best of our knowledge, these approaches are not general and relies on the continues usage of motion planners, collision checkers and computing domain-dependent heuristics.

The aim of our work is to develop a general approach for combined task and motion planning which can make use of recent advances of modern classical AI planning algorithms. For this purpose, in this chapter we propose the following to address the limitations of chapter 5

1. We present a general language to specify and to automatically compile a CTMP problem into a classical planning problem.

2. We show how we can model different types of CTMP problems, extending the model of *Pick-and-Place* problem seen in section5.3, where actions *MoveBase* and *MoveArm* are common to all proposed problems. These actions assume the previous computation of a base and arm graph.

3. We show how to define different object shapes, their allowed poses and grasping poses.

4. Finally we propose a general algorithm which is domain-independent and heuristics derive from the FSTRIPS model with state constraints. We introduce a method to deal with state constraints during search time using a computed set of no good atoms. These no good atoms are atoms that violates a state constraint. This set is used to compute the novelty of a state.

The chapter is organized as follows:

1. We propose and define an input language to specify CTMP problems. This language specifies the geometric information of the environment, as well the geometric details of each different 3D-shape object.

2. We review and describe the preprocessing stage without the limitations of chapter 5. We show step by step how based on an input, we construct graphs and tables containing overlaps and grasping poses.

3. We show how to model different CTMP problems. Starting from a general encoding which assumes the compilation of two different graphs: base graph and arm graph. We model problems which fully requires to interleave task and motion planning.

4. We present a general and domain-independent planning algorithm that avoid the specific computation of previous domain-dependent heuristics. Instead, it relies on the problem structure and the state constraints to compute the novelty of a state.

5. We show how search can be improved by explicitly defining boolean features.

6. We show some extensions and optimizations for both the compilation process and the planning algorithm.

7. We report the experimental results of different types of problems involving a number of objects of different shapes.

8. Finally we discuss the general approach.

6.2 Input Language

To compile a CTMP problem into a classical planning problem our method requires a set of parameters. We specify the object geometries, robot kinematics and a set of discretization parameters. The planning encoding is based on a preprocessing stage where graphs and tables are precomputed. Computing these graphs and tables depend on the following specifications: 1) an environment with static elements and their geometric properties (support surfaces and their sizes), 2) movable objects with geometric properties (size, shapes, graspable components), 3) a robot with its kinematic model and geometric properties, 4) a set of parameters used to sample/discretize and 5) a set of constraints. Specifically:

- We specify a static world, which is represented by a continuous 3-dimensional space with objects that cannot be manipulated by the robot. Surfaces where object configurations and end-effector poses are discretized are known as *support surfaces*, like tables. Each support surface must be specified in terms of width, length and height $(w \times l \times h)$ and its absolute position.

- Our method requires as input the description of which objects can be manipulated by the end-effector. For these *movable* objects, our method needs as part of its input the geometric details for each different 3D-shape in the environment. Our current implementation of this input accepts cylinders described by height and radius, and for prism with a square face the side of the square and the height of the prism. For other objects with more complex shapes, our current implementation accepts trays and cups, but for such objects the implementation requires details on how is graspable by the robot: for example a tray is graspable by the edge but not the middle.

- We specify the robot kinematic details, as well as its base size and end-effector size. MoveIt! takes the kinematic model to compute motion plans and to check for collisions.

- We define a set of parameters to discretize the configuration space, object poses, grasping poses and base and end-effector poses.
- We explicitly define constraints expressing joint limits and tolerance.

Our preprocessing process requires as input a world $\mathcal{W}$, a set of manipulable objects $\mathcal{O}$ per each different shape of a movable object, the kinematic model $\mathcal{K}$ of a robot, a set of joint constraints $\mathcal{J}$ and a vector of discretization parameters $\mathcal{D}$. Formally,

Definition 6.2.1 *The input for compiling a CTMP problem into a classical planning problem is $I_p = \langle \mathcal{W}, \mathcal{O}, \mathcal{K}, \mathcal{J}, \mathcal{D} \rangle$, where:*

- A world is defined as $\mathcal{W} = \langle \mathcal{Q}, \mathcal{E} \rangle$, where:
  - $\mathcal{Q}$ defines boundaries of the robot workspace. Is specified as a pair $\langle X, Y \rangle$ where X and Y specifies a rectangle.
  - $\mathcal{E}$ is a finite set of static elements $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$, such that each static element $e_i = \langle i, p_i, \mu_i \rangle$, where
    - i is the unique identifier of e_i.
    - p_i is the absolute position $(x_i, y_i, z_i, \theta_i)$ with respect to the world frame.
    - μ_i are the geometric properties of element e_i, such that $\mu_i = \langle w_i, l_i, h_i \rangle$, where
      - w_i is the width of e_i.
      - l_i is the length of e_i.
      - h_i is the height of e_i.
- $\mathcal{O}$ is a set of manipulable objects, such that each $o_s = \langle s, \mu_s, A_s, G_s \rangle$, where
  - s is the unique identifier of o_s. The identifier represents the type (shape) of the object. For example an object can be a box, a cylinder or a tray.
  - μ_s represents the geometric properties of object o_s, such that $\mu_s = \langle w_s, l_s, h_s, r_s \rangle$, where,
    - w_s is the width of o_s.
    - l_s is the length of o_s.
    - h_s is the height of o_s.
    - r_i is the radius of o_s.

- A_s is the set of allowed rotations for each object type, specified in terms of θ, ψ, ϕ for pitch, roll and yaw. Rotation poses are specified in radians. For example, a block with the allowed rotation poses $(0.7853, 0.0, 0.0)$ and $(1.57, 0.0, 0.0)$ means that for any valid (x, y, z), the object can have the configurations $(x, y, z, 0.7853, 0.0, 0.0)$ and $(x, y, z, 1.5708, 0.0, 0.0)$.
- G_s is a set of grasping poses for o_s, specified in terms of relative end-effector poses, such that $g_s = \langle x, y, z, \theta, \psi, \phi \rangle$. For example, a grasping pose for any o_s could be $g_s = \langle 0.15, 0.0, 0.07, 0.0, 0.0, 1.57 \rangle$, which means that the robot's end-effector pose must be located in g_s to be able to grasp an object o_s. Notice that the pose is relative to the object frame, it is not an absolute pose.

- $\mathcal{K}$ is the definition of the kinematic model of the robot.
- $\mathcal{J}$ is a set of constraints expressing joint limits and joint tolerance. $\mathcal{J}$ can be empty.
- A set of discretization parameters $\mathcal{D} = \{\epsilon, \alpha, \beta, k_B, \sigma, \lambda, \Delta\}$ where,
  - ϵ is the granularity used for sampling bases along x and y coordinates.
  - α is the discretization of bases orientation along the θ coordinate.
  - β is the orientation tolerance for the robot base. This tolerance denotes the maximum absolute orientation towards a support surface.
  - k_B is the k nearest neighbors for each discretized robot base B.
  - σ is the discretization of the support surfaces in terms of possible object configurations.
  - λ denotes the maximum levels on top of the support surfaces. Maximum level values larger than one mean that objects can be stacked.
  - Δ is the set of directions from where the end-effector approaches the object. Each direction is given as a vector where the end point is the grasping pose.

6.3 Preprocessing

First, to encode a CTMP problem we need a crucial compilation process step where the base and arm graphs are computed, as well as tables for lookup operations at planning time. These tables are computed using motion planners and collision checkers, only at preprocessing time. The compilation process is fast and efficient, and does not depend on the number of objects. This means that a single compilation can be used for several instance problems without the requirement to recompute any table or graph. For this reason we avoid the extra calls to motion planners and collision checkers during search.

As described in the previous section, a CTMP problem is compiled to a classical planning problem given a defined input I_p. Once the input is given, our preprocessing method works as follows: we consider the set of support surfaces $\mathcal{E}$, where $\mathcal{E} = \{e1, e_2, \ldots, e_n\}$ and the workspace $\mathcal{Q}$ specified in the world $\mathcal{W}$. Each static element is defined by a tuple $e_i = \langle i, p_i, \mu_i \rangle$ where i is the unique identifier of e_i, p_i is the absolute position $(x_i, y_i, z_i, \theta_i)$ with respect to the world frame, and μ_i are the geometric properties of element e_i, composed by the width w_i, the length l_i and the height h_i.

6.3.1 Constructing the base graph

Constructing the base graph follows the same process as described in section 5.2, but now it depends in the input parameters. The *base graph* is computed by discretizing base configurations (x, y, θ) around support surfaces. x and y are discretized according to ϵ meters and θ according to α radians. Valid base configurations are only those for which θ orients the base towards the closest support surface, the maximum orientation tolerance is given by β. Afterwords, we call the MoveIt! motion planners to connect each such configuration with up to k_B of its closest neighbours. MoveIt! checks for collisions between the discretized base configurations and the static objects. The base graph consists of base configurations and trajectories between them. Given $\langle\langle b_i, e\rangle : b_j\rangle$, where b_i and b_j are the source and target nodes connected by edge e and $i \neq j$. The maximum number of bases is given by

$$N_B = \frac{X \times Y}{\epsilon} \times \frac{2\pi}{\alpha}. \tag{6.1}$$

6.3.2 Constructing the arm graph

Constructing the arm graph is similar to the construction of the arm graph that we have seen in section 5.2. However, there are some subtle differences that makes the preprocessing stage to be more general and independent on object types. The virtual configurations are discretized on top of a virtual table according to the input parameters and each object $o_s \in \mathcal{O}$. Virtual configurations now depend on the objects type o_s, the allowed poses A_s and grasping poses G_s.

To remind the reader, in order to compute the *arm graph* we consider the robot on a *virtual base* $B_0 = \langle x, y, \theta \rangle$ with $x = y = \theta = 0$ in front of a *virtual table* whose height is the height of the actual tables, and whose dimensions exceed the robot arm work space, which is the (local) space that the robot can reach without moving the base. By considering the robot acting in this local virtual space without moving from this virtual base B_0, we will compute all object configurations and feasible grasping poses and arm trajectories.

Computing the virtual object configurations

Virtual object configurations are discretized on top of the virtual table. The x, y space of the virtual table is discretized regularly into D position pairs x_i, y_i. D depends on a discretization parameter σ, which is in meters, on top of the virtual table. If the height of the objects is h' and the height of the tables is h, then the z coordinate of the object's center of mass is $z = h + h'/2$. Notice that problems like *Blocks World* or *Structures Building* involve configurations not only right above the support surfaces, but at different levels l above them. In addition, virtual object configurations are discretized per each different object type.

The discretization of virtual object configurations at different altitudes depends on which are the support object types, which are the objects under the virtual configuration to be discretized. For each object type, we compute the $z_l = h + h'/2$ plus the sum of the heights of each possible permutation of support objects with $1 \leq l \leq \lambda$, where λ is the maximum number of levels in top of the support surface and l is the current level from which we are computing the virtual object configuration. Meaning that a compilation with λ levels will have virtual object configurations per each different object type with a maximum value of z_λ. For example, assume we have two different object types (blocks and cylinders), and $\lambda = 2$. The height of a block is h_b and the height of a cylinder is h_c. Let's suppose we are discretizing virtual object configurations for a block at some position (x, y) and level $l = \lambda = 2$. The possible virtual object configurations at level 2 are given by the possible permutations of the support objects at level 1 . If $l = 2$, then $z_2^1 = h + h_b/2 + h_b$ and $z_2^2 = h + h_b/2 + h_c$. Thus, the possible virtual object configurations for a block at position (x, y) are $C_1 = \langle x, y, z_1, \theta, \psi, \phi \rangle$, $C_2 = \langle x, y, z_2^1, \theta, \psi, \phi \rangle$ and $C_3 = \langle x, y, z_2^2, \theta, \psi, \phi \rangle$, where the virtual object configuration C_1 is a configuration at level 1. Notice that problems like *Pick-and-Place* the maximum level is $\lambda = 1$, so $z_1 = h + h'/2$. Figure 6.1 shows the possible permutations assuming that (x, y) is always the same for each discretized virtual configuration, and assuming that $\lambda = 2$. You can notice that there exists different z values depending on the object type, the level and which is the support object. For each different type of object o_s, the orientation angles are also discretized according to A_s per each x_i, y_i, z_l, where A_s is given as input and it is a set representing the allowed rotations of an object being at any center of mass (x, y, z). Then, each virtual object configuration is a tuple of the form $\langle x_i, y_i, z_l, \theta, \psi, \phi \rangle$.

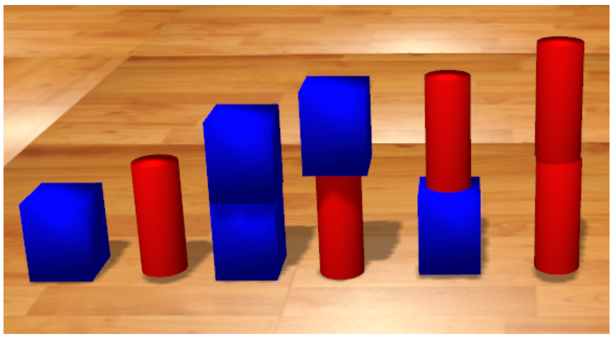


Figure 6.1: All possible permutations assuming that x, y is always the same and $\lambda = 2$.

Let us calculate the maximum number of virtual object configurations N_C. Before that, we have to calculate the maximum number of virtual object configurations of one given object type o_s. If $\lambda = 1$, it means that there are only discretized virtual object configurations right above the table. As an object type o_s can have different rotation poses A_s in one position par (x, y), the number of virtual object configurations of and object of type o_s is $N_s = D \times |A_s|$, where D are the discretized positions pairs (x, y) and $|A_s|$ is the number of possible orientation poses. If $\lambda > 1$, then we have to calculate all the possible virtual object configurations based on which are the support objects. If we refer again to Figure 6.1, we see that a block can be in 3 different virtual configurations per each pair (x, y): One right above the table, the other one on top of another block which is above the table, and the third one on top of a cylinder which is above the table. Let's define the equation to compute the number of virtual object configurations of an object o_s independently on the value of λ:

$$N_s = \begin{cases} D \times |A_s| & \text{if } \lambda = 1 \\ \left[|A_s| + \left(\sum_{l=2}^{l=\lambda} |A_s| \times \frac{(|O| + L - 1)!)}{L! \times (N-1)!} \right) \right] \times D & \text{if } \lambda > 1. \end{cases} \tag{6.2}$$

Where $|O|$ the number of object types, and $L = l - 1$.

Notice that with this equation we avoid repetitions of computed virtual configurations. For example, if $\lambda = 3$, and there are two different types of objects: blocks and cylinders, the discretized virtual object configurations for a block on level $l = \lambda = 3$ are the same if the support objects are a cylinder and a block or a block and a cylinder in levels 1 and 2 respectively.

Thus, the maximum number of virtual object configurations is

$$N_C = \sum_{o_s \in \mathcal{O}} N_s. \tag{6.3}$$

Computing grasping poses and trajectories

Each virtual object configuration represents a possible center of mass for the objects when sitting at location x_i, y_i, z_l over the virtual table. For each such configuration $C = \langle x_i, y_i, z_l, \theta, \psi, \phi \rangle$, k grasping poses G_C^j are computed from which an object at $\langle x_i, y_i, z_l, \theta, \psi, \phi \rangle$ could be grasped, and a motion planner (MoveIt!) is called to compute $k' = |\Delta|$ arm trajectories for reaching each such grasping pose G_C^j through a set of different waypoints from a fixed resting pose and the robot base fixed at B_0. Per each different object type $o_s \in \mathcal{O}$, we define a set of G_s grasping poses, from where the grasping poses G_C^j are obtained. Fig. 6.2 shows five different grasping poses for a block type object. G_s defines the grasping poses relative to an object type s located at $C_0 = \langle x = y = z = \theta = \psi = \phi = 0 \rangle$. When a new virtual object configuration is generated, the grasping poses G_s are transformed to the new virtual object configuration. The k' trajectories depend on the direction approaches to each grasping pose. Δ is the set of direction approaches for the end effector from the resting pose to a grasping pose. A resting pose is an end effector pose with the arm being at some configuration which do not collide with anything while the robot is moving its base.

Formally, let's define the total number of grasping poses per each object type. On equation (6.2) we show how to compute the number of virtual object configurations N_s for an object of type s. Each object type has defined a set of grasping poses G_s when an object located at C_0. The number of grasping poses per each object type s is $N_{gp}^s = N_s \times |G_s|$. Finally, the total number of grasping poses is

$$N_{GP} = \sum_{o_s \in \mathcal{O}} N_{gp}^s. \tag{6.4}$$

Definition 6.3.1 *A virtual object configuration C is valid iff there exists at least one feasible grasping pose G_C^j from where the end effector can grasp the object located at C.*

Definition 6.3.2 *A grasping pose G_C^j for a virtual object configuration C is valid iff there exists at least one feasible trajectory to reach G_C^j.*

N_{GP} is the total number of grasping poses and k' the number of trajectories per each grasping pose resulting in up to $N_{Traj} = N_{GP} \times k'$ arm trajectories in total. Moreover, each grasping pose G_C^j does not depend only on C, but on the object type s of the virtual object. Meaning that two virtual objects with different shapes with the same virtual configuration C could not be grasped with the same grasping pose G_C^j. In addition, grasping poses are not only used to grasp an object, but to place the object in any other configuration. For this purpose, we keep track of how the object o has been grasped. This is the relative pose between the object configuration and the gripper which we call r_o.

A feasible grasping pose is an end effector pose. The *arm graph* has as nodes the end effector poses A. In addition to the resting arm configuration A_0, which is an end-effector pose with the arm joints being at some position which does not interfere the robot movement. The arm trajectories that connect the resting arm configuration A_0 with an end-effector pose A are provided by the edges in the arm graph between A_0 and A. The graph contains also the inverse edges that correspond to the same trajectories reversed. Grasping poses that are not reachable with any trajectory from the resting arm configuration are pruned. Virtual object configurations whose all grasping poses have been removed, are pruned as well. In other words, we cannot have a virtual object configuration without a feasible trajectory to a grasping pose for an object being at that configuration. The graph is encoded as $\langle\langle a_i, e\rangle : a_j\rangle$, where a_i and a_j are the source and target nodes connected by edge e with $i \neq j$.

The number of *robot configurations* results from the product of the number of valid end effector poses N_{GP} and the number of base configurations N_B. In the experiments we consider numbers that go from tens to a few hundred and which thus result into thousands of possible robot configurations. The computation of the base and arm graphs are required by the procedures defined in the $MoveBase$ and $MoveArm$ actions that access the source and target configuration of each graph edge.

6.3.3 Constructing lookup tables

After constructing the base and the arm graph, a number of tables are computed. These tables allow fast lookup operations and avoid the expensive calls to motion planners and collisions checkers during planning time.

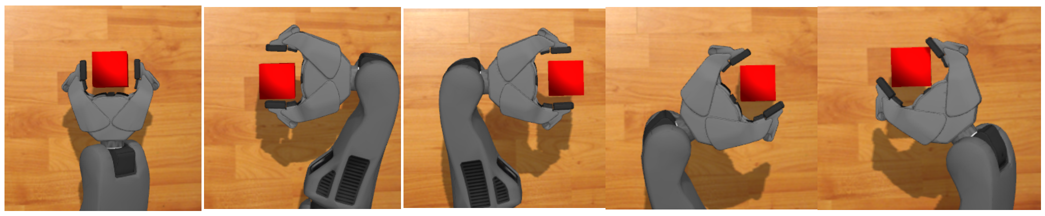


Figure 6.2: Five different grasping poses for a block.

Constructing the Grasping/Placing Table

For each reachable grasping pose G^j_C, we store the tuple $\langle G^j_C, s, r_o, C \rangle$, where s denotes the object type of an object $o_s \in \mathcal{O}$, in a hash table called *grasping_placing_poses*. The table maps end effector poses (grasping poses) into virtual object configurations, depending on the object type and how it has been grasped, which is denoted by r_o.

Being Q the triplet $\langle A, s, r_o \rangle$, a function called $vplace$ maps Q into virtual object configurations. The meaning of $vplace(Q) = C$ is that when the robot base is at B_0, in front of the virtual table, and the end effector pose is A, then an object with type s being grasped with relative pose r_o will be placed at the virtual object configuration C.

Computing Real Object Configurations

The virtual object configurations are those configurations from where an object located on them, can be grasped by the robot being at base B_0. However, virtual object configurations do not represent the total set of configurations where objects can be located. The set of real *object configurations* are then defined and computed as in chapter 6. The configuration C moves to a new pose C' that is given by a transformation $T_B(C)$ as defined in equation (5.1). As now object poses are represented as a six coordinates system , the virtual object configurations $C = \langle x, y, z, \theta, \psi, \phi \rangle$ are now transformed into *actual object configurations* given by such triplets $T_B(C) = \langle x', y', z, \theta', \psi', \phi' \rangle$. Again, we only take as valid those real object configurations which fall within a support surface in the actual environment. Thus, the maximum number of real object configurations is

$$N_R = N_B \times N_C. \tag{6.5}$$

This process produces a table called *real_to_virtual*. This table is composed by tuples of the form $\langle B, T_B(C), C \rangle$ and captures which is the corresponding virtual configuration C given a base B and a real object configuration $T_B(C)$. Being V the tuple $\langle B, T_B(C) \rangle$, where B is the current base configuration and $T_B(C)$ is a real object configuration, the function $real\_virtual$ maps V into virtual object configurations. The meaning of $real\_virtual(V)$ is that when the robot base is at B and some object o_s is at the real object configuration $T_B(C)$, then this object is at virtual object configuration C. Having C, the planner can check which are the grasping poses required for pick-up or place operations looking at table *grasping_placing_poses*.

Computing Relative Object Configurations

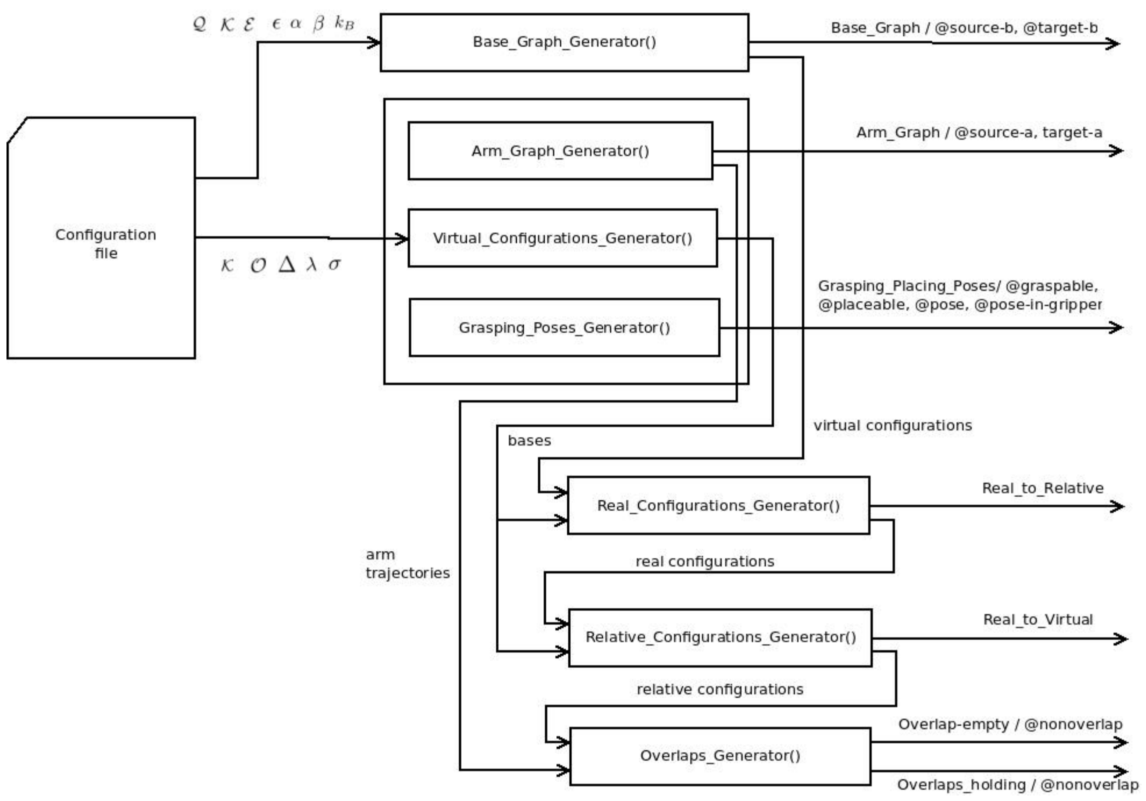
We precompute tables to prevent collisions for avoiding the use of collision checkers during planning time. Overlap tables are accessed by procedures each time that the robot performs an arm translation action. For this purpose we define state constraints of the form $non{-}overlap(B, Traj, C, Hold)$. Non overlaps are used to express that moving the arm along trajectory $Traj$ when the robot current base configuration is B, while the gripper is holding $Hold$, will cause a collision with some object being at configuration C.

Similar as in chapter 5, performing these collision tests during planning time requires to precompile two additional tables called ***HT*** and ***NT***. The table ***HT*** captures the fact that the robot is holding an object with its gripper, while the table ***NT*** assumes nothing is being held. The differences between these tables and the ones computed in chapter 5 is that now overlaps take into consideration that we can have different object types, while previously we assumed that we had one object type. Overlap tables are made of tuples $\langle Tr, s_h, s_o, C\rangle$ where Tr is a trajectory in the arm graph, s_h is the type of the object being held if any, s_o is the type of the object which produces a collision, and C now is what we will call a *relative object configuration*.

To review, *relative object configuration* are different than both virtual and real object configurations. Relative object configurations are defined as the set of configurations $T_B^{-1}(C)$ for all bases B and all real object configurations C, where T_B^{-1} is the inverse of the linear transformation T_B in (5.1). We compute a new structure called *real_to_relative* composed by tuples of the form $\langle B, C, T_B^{-1}(C)\rangle$ to get the relative configuration given the robot base B and the real object configuration C. The maximum number of relative configurations is

$$N_{rel} = N_B \times N_R. \tag{6.6}$$

Being W the tuple $\langle B, C\rangle$, where B is current base configuration and C is a real object configuration, the function $real_relative$ maps W into relative object configurations. The meaning of $real_relative(W)$ is that when the robot base is at



$\mathcal{Q}\ \mathcal{K}\ \mathcal{E}\ \epsilon\ \alpha\ \beta\ k_B$
Base_Graph_Generator()
Base_Graph / @source-b, @target-b
Arm_Graph_Generator()
Arm_Graph / @source-a, target-a
Configuration file
Virtual_Configurations_Generator()
$\mathcal{K}\ \mathcal{O}\ \Delta\ \lambda\ \sigma$
Grasping_Placing_Poses/ @graspable, @placeable, @pose, @pose-in-gripper
Grasping_Poses_Generator()
virtual configurations
bases
Real_Configurations_Generator()
Real_to_Relative
arm trajectories
real configurations
Relative_Configurations_Generator()
Real_to_Virtual
relative configurations
Overlap-empty / @nonoverlap
Overlaps_Generator()
Overlaps_holding / @nonoverlap

Figure 6.3: Inputs and outputs for preprocessing stage.

B and some object o_s is at the real object configuration C, then this object is at relative object configuration $T_B^{-1}(C)$. So during planning time we have constant access to check which is the relative configuration of an object at any state s. In other words, being C a real 3D object configuration obtained by mapping a point C' in the virtual table by applying transformation [formula T_B(C)], then C'' denotes a point in the “virtual” space relative to the base B_0. Again, relative object configurations C'' that do not fall within the virtual table, are pruned. Fig. 6.4 shows each relative object configuration C'' obtained from applying transformation $T_B^{-1}(C)$ to real object configurations C.

Constructing the overlap tables and checking for collisions

Constructing overlap tables requires a set of relative object configurations and arm trajectories. Overlap tables contain the information about collisions while the robot performs a trajectory. The *HT* contains the tuple $\langle Tr, s_h, s_o, C\rangle$ for a trajectory Tr and a relative object configuration C, iff the robot arm moving along trajectory Tr will collide with an object of type s_o in the relative configuration C

when the robot base is at B_0 and the gripper is carrying an object of type s_h. Similarly, the tuple $\langle Tr, s_h, s_o, C\rangle$ belongs to the *non-holding overlap table (NT)* iff the same condition arises when the gripper is empty, where s_h and s_o are the object types of the object being held and the object which causes a collision, respectively.

The number of calls to MoveIt! is given by the number of trajectories. For each trajectory Tr, the collision checker tests in one single scan which relative configurations C are on the way. Given the set of relative object configurations, and for all computed trajectories, MoveIt! checks which of those trajectories collides with an object of some type being at some relative object configuration. As relative object configurations denote a point in the "virtual" space relative to the base B_0, collisions are only checked while the robot is in B_0. As an example, the figure 6.4 show the relative object configurations and the robot being at base configuration $B = B_0$. All trajectories are checked from B_0 and not from any other base $B = B'$. Using overlap tables during planning time allows to check for collisions in constant time.

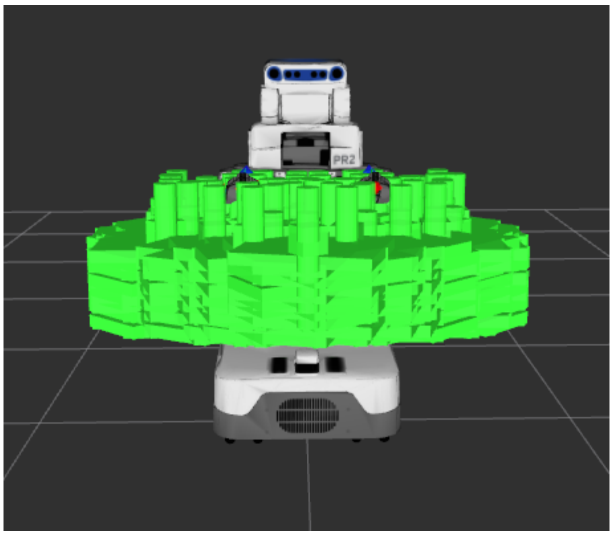


Figure 6.4: Planning scene with relative object configurations after preprocessing stage. Each relative object configuration denotes an object with some shape. Configurations are relative to the robot base. Each configuration is inside the robot workspace.

A state constraint is a formula of the form $non{-}overlap(B, Tr, C, Hold)$. It denotes whether trajectory Tr collides with and object in a real configuration C when the robot base is B. If $Hold$ is $None$, this is checked by testing whether the tuple $\langle Tr, s_h, s_o, T_B^{-1}(C)\rangle$ is in the *NT* table, and if $Hold$ is not $None$, by testing whether the pair is in the *HT* table, where $T_B^{-1}(C)$ is the inverse transformation of a real configuration and denotes a relative configuration.These are lookup operations in the two (hash) tables *NT* and *HT*. Being N_{Traj} the number of arm trajectories, N_{rel} the number of relative object configurations and $N_T = |O|$ the number of object types, then the maximum size of overlap tables is $N_{Traj} \times N_{rel} \times {N_T}^2 \times 2$. This last number is independent of the number of objects but higher than the number of virtual configurations. As before, in the worst case, it is bounded by the product of the number N_B of robot bases and the number N_r of real object configurations, which in turn is bounded by $N_B \times N_C$, where N_C is the number of virtual object configurations. Usually, however, the number of entries in the overlap tables *NT* and *HT* is much less, as for most real object configurations C and base B, the point $T_B^{-1}(C)$ does not fall into the "virtual table" that defines the local space of the robot when fixed at B_0.

To summarize, the preprocessing stage provides a number of compiled tables and two graphs: A *base_graph* and an *arm_graph* from base and arm robot transitions; a table *grasping_placing_poses* that maps grasping poses into virtual object configurations depending on object geometry and the way that has been grasped; a table *real_to_virtual* that captures which is the corresponding virtual configuration C given a base B and a real object configuration $T_B(C)$; a table *real_to_relative* to get the relative configuration given the robot base B and the real object configuration C; and two overlap tables (*HT* and *NT*) capturing whether a given trajectory produces some collision. The figure 6.3 shows the inputs and outputs of the preprocessing stage. Each box is a different function. Outputs are tables and graphs. Symbols denoted by @ are the procedures which make use of them.

6.3.4 Complexity

We analyze the space and time complexity of constructing each table and graph. The space complexity shows how graphs and tables grow in size. The time complexity of constructing the graphs (base and arm), it depends on the number of calls to MoveIt!, for computing motion plans. Similarly, the time complexity of computing overlap tables depends on the number of calls to MoveIt!, for checking for collisions. Table 6.3 shows a summary of the previously defined notation.

Considering the space complexity of graphs and tables, we refer to Table 6.1. More specifically, for graphs:

- The space complexity for the base graph depends on the number of discretized bases N_B, the number of connections k_B from each base times 2, as each considered trajectory is counted twice, for both directions.

- The space complexity for the arm graph, depends on the maximum number of arm trajectories N_{Traj} times 2, as trajectories are counted twice. One from the resting pose to another end effector pose, and the other way around.

Regarding constructed tables:

- The size of the table capturing grasping and placing poses is the maximum number of grasping poses N_{GP}.
- The maximum size of the $real\_to\_virtual$ table depends on the total number of discretized bases N_B and the total number of virtual object configurations N_C.
- The maximum size of the $real\_to\_relative$ table depends on the total number of discretized bases N_B and the total number of real object configurations N_R. Being $N_R = N_B \times N_C$.
- The maximum size of overlap tables is $N_{Traj} \times N_{rel} \times {N_T}^2 \times 2$.

Regarding time complexity, we are going to analyze arm and base graphs, and overlap tables as the number of calls to MoveIt!. The processes to construct the base and the arm graph call MoveIt! to compute motion plans (trajectories) between configurations. On the other hand, constructing overlap tables requires calls to MoveIt! for checking for collisions between previously computed arm trajectories and objects being at relative object configurations. Table 6.2 shows the time complexity.

- Constructing the base graph requires a maximum number of calls to MoveIt! given by the total number of discretized bases N_B and the total maximum number of connections k_B.
- Constructing the arm graph requires a maximum number of calls to MoveIt! given by the total number of virtual object configurations N_C, the total number of k grasping poses per virtual object configuration and the total number of k' trajectories per each grasping pose.
- Checking for overlaps requires a number of calls to MoveIt! which depend on the number of trajectories N_{Traj} and the number of object types N_T, as trajectories are checked while the robot is holding the different object types using a bounding box for the different possible orientations of the object inside the gripper.

Data Structure	Space Complexity
Base Graph	$O(N_B \times k_B \times 2)$
Arm Graph	$O(N_{Traj} \times 2)$
Grasping/Placing Poses	$O(N_{GP})$
Real to Virtual	$O(N_B \times N_C)$
Real to Relative	$O(N_B \times N_R)$
Overlaps HT	$O(N_{Traj} \times N_{rel} \times {N_T}^2 \times 2)$
Overlaps NT	$O(N_{Traj} \times N_{rel} \times {N_T}^2 \times 2)$

Table 6.1: Space complexity per each data structure. Space complexity shows the size of each graph and table in worst case.

Data structure	Time Complexity
Base Graph	$O(N_B \times k_B)$
Arm Graph	$O(N_C \times k \times k')$
Overlaps(HT and NT)	$O(N_{Traj} \times N_T)$

Table 6.2: Time complexity per each data structure which is constructed by making calls to MoveIt!.

Notation	Description
N_C	Maximum number of virtual object configurations
N_B	Maximum number of base configurations
N_{GP}	Maximum number of grasping poses
N_R	Maximum number of real object configurations
N_{rel}	Maximum number of relative object configurations
N_{Traj}	Maximum number of trajectories
N_T	Number of different object types
k_B	Number of closest bases
k	Number of grasping poses per each object at some virtual configuration
k'	Number of trajectories per each grasping pose.

Table 6.3: Description of previously defined notation.

6.4 Modeling CTMP Problems

In section 5.3 we saw how to model *Pick-and-Place* tasks in FSTRIPS. In this chapter we show how we can model different CTMP problems and how we can solve them using general and domain independent algorithms. The considered CTMP problems involve: 1) a robot with a seven degrees of freedom manipulator that can move around a 3D space and 2) a number of 3D objects of different types(shapes) located on top of support surfaces (tables). The robot is a PR2 using a single arm, but can be generalized easily to any robot or manipulator.

```
(:action MoveBase
 :parameters (?e - base-graph-traj-id)
 :prec (and (= Arm ca0)
            (= Base (@source-b ?e))
 :eff (and (:= Base (@target-b ?e)))
)

(:action MoveArm
 :parameters (?e - arm-graph-traj-id)
 :prec (and (= Arm (@source-a ?e))
 :eff (and (:= Arm (@target-a ?e))
           (:= Traj  ?t))
)

(:state-constraint
  :parameter (?o - object-id)
   (@non-overlap Base Traj Hold (Conf ?o)))
```

Figure 6.5: General CTMP Model Fragment in Functional STRIPS: Action and state constraint schemas. Abbreviations used. Symbols preceded by "@" denote procedures. State constraints prevent collisions during arm motions. This model fragment is general for all the exposed domains.

A fragment of the planning encoding featuring $MoveBase$ and $MoveArm$ actions and the $non{-}overlap$ state constraint, is shown in Figure 6.5. This model fragment is common to all of our modeled CTMP problems. These two actions are exactly the same as in chapter 5. It assumes the two finite and directed graphs obtained from the preprocessing stage: the *base graph* and the *arm graph*. Nodes represent base configurations or end effector poses respectively, while edges represent base or arm trajectories.

To remind the reader, state variables $Base$ and Arm represent the base configurations (x, y, θ) of the robot base and the end-effector poses $(x, y, z, \theta, \psi, \phi)$. The last performed arm trajectory is represented by the state variable $Traj$, which is required to check for collisions not only at the beginning and the end of the arm action, but during the whole motion. The state variable $Hold$ denotes if an object is being held, and which object is that. Finally, $Conf(o)$ denotes the current real object configuration of object o. A current real object configuration is represented by a tuple of the form $(x, y, z, \theta, \psi, \phi)$. To remind reader, procedures $(@source{-}b$ $?e)$ and $(@source{-}a\ ?e)$ check on the previously computed graphs that the current base configuration or end effector pose is the source node of edge e, while $(@target{-}a\ ?e)$ and $(@target{-}b\ ?e)$ return which is the target node of edge e and update the base and arm configurations. All base, arm and object configurations, which are represented by symbolic ids, have been computed previously in the preprocessing stage. Base and arm trajectories are motion plans which have been also computed during the graphs construction.

Finally, state constraints are used to prevent collisions during arm trajectories. Collisions are to be avoided not only at the beginning and at the end of the motion, but also during the trajectory execution. For simplicity we assume that collisions result exclusively from arm motions and not from base motions. It can be easily extended using the same procedure with state constraint formulas. Collisions may occur between arm motions and movable objects, not with static objects. This is because mobile objects are located in top of support surfaces which are static. Base and arm motions are precomputed taking static objects like support surfaces into account. In addition, we enforce the arm to be in a resting configuration (ca0) when the robot moves its base.

To review, there is one state constraint denoted by the procedure *@nonoverlap(Base,Traj,Conf(o),Hold)* for each object o. $Base$ is the current robot base, $Traj$ is the last executed arm motion, $Conf(o)$ is the current object o configuration and $Hold$ denotes the object being held, or it returns empty. So, the state constraint *non-overlap* checks if a collision occurs between object o being at configuration $Conf(o)$ and the robot moves the arm following trajectory $Traj$ while being at base configuration $Base$. The test depends also on whether the gripper is holding an object or not and the type of the object if being held. As we have shown in section 6.3, this procedure is also computed from two *overlap tables* that are precompiled by calling a motion planner using MoveIt! and a collision-checker. If $Hold = \emptyset$ the procedure checks in table *NT*. On the other hand, if $Hold \neq \emptyset$ the procedure checks in table *HT*. The overlap tables encode collisions between trajectories and relative object configurations, depending on the shape of the objects involved in the collision. The procedure *@nonoverlap(Base,Traj,Conf(o),Hold)* takes as an argument the real object configuration $Conf(o)$. In order to check in the overlap tables, first the real configuration must be converted to the corresponding relative configuration. Having the base $Base$ and the real object configuration $Conf(o)$, the function $real\_relative$ checks which is the corresponding relative configuration.

6.4.1 Modeling Pick-and-Place Problems

Pick-and-Place problems involve moving some objects of different types from some initial configuration to a final configuration or set of configurations, which may require moving obstructing objects as well.

```
(:action Pick-up
 :parameters (?o - object-id)
 :prec (and
  (=  Hold  None)
  (@graspable ?o))
 :eff (and
  (:= Hold ?o)
  (:= (Conf ?o) c-held))
  (:= (Conf-g ?o)
      (@pose-gripper Base Arm (Geom ?o) (Conf ?o))
    )

(:action Place
 :parameters (?o - object-id)
 :prec  (and
  (= Hold ?o)
    (@placeable ?o)
 :eff (and (:= Hold None)
   (:= (Conf ?o)
    (@pose Base Arm (Geom ?o) (Conf-g ?o)))
   (:= (Conf-g ?o) None)
   )
```

Figure 6.6: CTMP Model Fragment for *Pick-and-Place* in Functional STRIPS: Actions *Pick-up* and *Place* schemas which extend the general model fragment.

The new additions with respect the modeling presented in section 5.3 are the state variables *Conf-g(o)* and *Geom(o)*, which represent the pose of the object relative to the gripper and the object type, respectively. *Conf-g(o)* is computed by taking into consideration the object pose and the end effector pose when the object is grasped. The object type includes different shapes: cylinders, cubes or trays, and it can be extended to different shapes as described in section 6.2.

Modeling a *Pick-and-Place* problem is similar to the one we saw in section 5.3. However, there are some differences. Previously, we assumed to use only one type of object (cylinders). Now we can extend these, and the rest of the problems, by having different object types (shapes) which can be located in different orientations. For this reason, the semantics of procedures @*graspable*, @*placeable* and @*pose* have a significant difference. The procedure denoted by the symbol @*graspable* checks if the robot being at some base and arm configuration, can grasp an object o of some specific type, determined by its shape by just closing the gripper. The procedure first converts the current object configuration in a virtual configuration. Given the current base $Base$ and the current real object configuration $Conf(o)$ of object o, using function $real\_virtual$ we can check on table *real_to_virtual* which is the corresponding virtual configuration C. The procedure continues by checking if it exists a tuple $\langle Arm, Geom(o), Conf - g, (o), C\rangle$ in the table *grasping_placing_pose*, where Arm is the current end effector pose, $Geom(o)$ is the type of object o, $Conf{-}g(o)$ is the relative object-gripper configuration and C is the virtual configuration.

The *Pick-up* action effect updates the holding state variable $Hold$ to the object that has been grasped. The configuration of object o is set up to a specific id which represents that o is being held (c-held). Finally, $Conf{-}g(o)$ is updated through the procedure denoted by the symbol @*pose-gripper*. This procedure returns the relative object configuration respect to the robot gripper.

The procedure denoted by the symbol @*placeable* checks if the robot being at some base $Base$ and arm configuration Arm, can place and object o of some specific type and with a relative object-gripper configuration $Conf{-}g(o)$ by just opening the gripper. This procedure needs to know the type of the object. For example, a cylinder can be placed on the corner of the table with some configuration, but a tray would fall if it is placed in the same configuration. It works similar to the @*graspable* procedure. As effects of the *Place* action, the state variable $Hold$ is updated to $None$, as well as $Conf{-}g(o)$. This is because now the object has no relative object-gripper configuration. Finally, the object configuration is updated through the procedure @*place(Base Arm Geom(o) Conf-g (o))*. Given the robot base and arm configurations, the type of the object and the relative pose between the object and the gripper, it gives the resulting object configuration after open the gripper. These procedures update state variables by doing fast lookup operations on precomputed tables. The number of state variables remains the same, with the addition of only one extra state variable, $Conf{-}g(o)$.

The initial situation provides initial values for the state variables *Base*, *Arm* (resting), *Traj* (dummy), *Conf-g(o)*, *Geom(o)* and *Conf(o)* for each object. Goals describe target object configurations.

6.4.2 Modeling Blocks World Problems

Blocks World problem is the classical Blocks World planning problem but modeled as a CTMP problem. Initially all blocks are located on top of the table or on top of other blocks, composing one or different towers. The goal is to stack or unstack blocks to compose one or more towers.

```
(:action Unstack
 :parameters (?o - object-id)
 :prec (and
  (=  Hold  None)
  (@graspable ?o)
  (@clear ?o))
 :eff (and
  (:= (Hold) ?o)
  (:= (on ?o) no_object)
  (:= (Conf ?o) c-held))
  (:= (Conf-g ?o)
      (@pose-gripper Base Arm (Geom ?o) (Conf ?o))
    )

(:action Stack
 :parameters (?o ?o' - object-id )
 :prec  (and
  (= Hold ?o)
  (@placeable ?o)
  (@stable
    (@pose Base Arm (Geom ?o) (Conf-g ?o)) ?o') )
 :eff (and
  (:= (Hold) None)
  (:= (on ?o) ?o')
  (:= (Conf ?o)
  (@pose Base Arm (Geom ?o) (Conf-g ?o)))
  (:= (Conf-g ?o) None)
   )
```

Figure 6.7: CTMP Model Fragment for *Blocks World* in Functional STRIPS: Action schemas *Unstack* and *Stack*, extend the general model fragment.

The general model fragment seen in section 6.4 is extended with actions *Un-*

stack(o), for unstacking an object o, and *Stack(o, o')* for stack an object o on top of another object o'. The *Unstack* action requires that the gripper is empty and that @*graspable(o)* is true, which is the same procedure as in *Pick-and-Place* domain. The only difference is that object o must be clear. The procedure denoted by the symbol @*clear(o)*, checks if there is any other object on top of o that prevents to grasp o. Notice that procedures @*graspable(o)* and @*nonoverlap(Base,Traj,Conf(o),Hold)* have different functionality than @*clear*. While the first checks if an object can be grasped by just closing the gripper and the second ensures that there is no collision during the arm motion, @*clear(o)* ensures that the object can be raised without compromise the stability of the stack.

$Hold = o$ and @*placeable(o)* are preconditions of the action *Stack(o, o')* as in *Pick-and-Place* domain. There is also a new precondition @*stable(Conf(o), o')*. The procedure denoted by the symbol @*stable* checks if the whole structure is stable when placing object o. For this purpose, a stability test is done not only between o and o', but between all other objects on top of or under o. The stability test consist to check if all the stack is aligned with the base center of mass within a margin that depends on the object geometry.

Finally, initial situation provides initial values for the state variables *Base*, *Arm* (resting), *Traj* (dummy), *Conf-g(o)*, *Geom(o)*, and *Conf(o)* for each object. Goals describe the target location of each block, denoted by fluents *on(o, o')*.

6.4.3 Modeling Structure Building Problems

```
(:action Unstack
 :parameters (?o - object-id)
 :prec (and
  (=  Hold  None)
  (@graspable ?o)
  (@clear ?o))
 :eff (and
  (:= (Hold) ?o)
  (:= (Conf ?o) c-held))
  (:= (Conf-g ?o)
        (@pose-in-gripper Base Arm (Geom ?o) (Conf ?o))
    )

(:action Stack
 :parameters (?o)
 :prec  (and
  (= Hold ?o)
  (@placeable ?o)
  (@stable
    (@pose Base Arm (Geom ?o) (Conf-g ?o)) ?o′) )
 :eff (and
  (:= (Hold) None)
  (:= (Conf ?o)
    (@pose Base Arm (Geom ?o) (Conf-g ?o)))
  (:= (Conf-g ?o) None)
   )
```

Figure 6.8: CTMP Model Fragment for *Structure Building* in Functional STRIPS: Action schemas Unstack and Stack, extend the general model fragment.

Structure Building problems involve moving objects of different types to build a structure with a given altitude. Initially, objects are placed in top of a table. The initial situation specifies the initial values for the state variables *Base*, *Arm* (resting), *Traj* (dummy), *Conf-g(o)*, *Geom(o)*, and *Conf(o)* for each object. Goals describe altitudes for any object, denoted by the procedure @*object_altitude(o, a)*. The procedure @*object_altitude(o, a)* checks if object o is at some altitude range a, where a is a symbolic type to identify a range of altitudes. Specifically, a denotes a range of the form $z_1 \leq a \leq z_2$. Where $z_1 < z_2$ are real values.

The model fragment does not differ too much from *Blocks World* model. *Un-*

stack(o) action schema is identical. The only difference is on action *Stack(o)*, which now has only one object as parameter. The reason is that an object o can be stacked on top of more than one object. For example, a tray can be placed on top of two or more blocks. For this purpose, the stability procedure differs from the one seen in Blocks World Model. The procedure denoted by the symbol @*stable* is a collection of physical tests among the objects involved in the structure. For this purpose we define a set of geometric constraints depending on the different object types that compose the structure. A cube or a cylinder can be placed on top of any other object if the stability test is satisfied. Also, these objects can be on top of a tray if they don't exceed the tray boundary. On the other hand, a tray can only be placed on top of another tray if it is aligned with the center of mass as seen before, or can be placed on top of two or more objects if their height is the same and the distribution of them compose a stable base. This last criteria means that at least two base objects must be at a sufficient distance and correct alignment to act as a support for the tray.

6.5 General Algorithm

The compilation of task and motion planning problems is efficient, does not depend on the number of objects and results in planning problems that are compact. Moreover, motion planners and collision-checkers are only used at compilation time and not during search.

We rely on the use of width-based search with a planning algorithm that is called Best First Width Search (BFWS). BFWS is a best-first search algorithm that combines width-based measures to compute novelty [72], with an implicit form of goal serialization and some benefits of the goal directed heuristic search. BFWS has been shown to have a very good performance on the classical planning benchmarks [74]. BFWS is a standard best-first search with a sequence of evaluation functions $f = \langle h, h_1, \ldots, h_n \rangle$ where the node that is selected for expansion from the OPEN list at each iteration is the node that minimizes h, using the other h_i functions for breaking ties. The novelty of a new generated state s, is the size of the smallest tuple of atoms that is true in s, but false in all previous generated states s' that have the same h_i values. It is shown in [33] a general BFWS algorithm adapted to planning with simulations called BFWS(R). Planning with simulations ignore the action structure and has only access to the structure of states and goals only. Indeed, the results of applying some action a to a state s can be obtained from external descriptions, as a black box procedure.

The algorithm BFWS(R) is a best first search algorithm where R is a set of atoms which is computed once from the initial state s_0 during preprocessing step on the simulation process, and then it is used to compute novelty during the search. A set R contains all the atoms that are made true from s_0 to each subgoal. In other words, R is subset of potential, intermediate subgoals, in the way to the top goals. Following the different sets R in [33], we consider the set R_G. This set is goal-oriented, meaning that it contains all atoms made true by computed plans, from s_0 to each subgoal. These plans are computed in a preprocessing stage using $IW(1)$ or $IW(2)$ from s_0. In other words, R_G is a goal oriented version of set $R[k]$. A set $R[k]$ contains the set of atoms that are true in states reached from s_0 by running an Iterated Width algorithm $IW(k)$, where $k = \{1, 2\}$. First, $IW(1)$ is run. If $IW(1)$ finds plans that satisfies all goals, then R is the set of atoms made true by such plans. Following the definition of $IW(k)$, if $IW(1)$ does not find a solution for all goals, then $IW(2)$ is run from s_0. If $IW(2)$ finds plans that satisfies all problem goals, then R is the set of atoms made true by such plans.

The algorithm BFWS(R) uses an evaluation function $f = \langle \#r, \#g \rangle$ for computing the novelty of a state s. Ties are broken by using $\#g$ for a state s, where $\#g(s)$ is the number of goals that have not been achieved yet in s, and the accumulated cost. The value of $\#r$ in a state s, denoted as $\#r(s)$ is a counter of the number of atoms in the set R that are made true at some point from s_0 to s. R is computed during the $IW(k)$, and it is used at search time to compute $\#r(s)$.

6.6 Extensions of BFWS(R) for Handling State Constraints

CTMP problems require solutions that avoids collisions between objects and between objects and the robot. For this reason we make use of state constraints in order to prune nodes whose actions violate a state constraint. We propose a new domain independent computational approach to deal with state constraints during search time as a form of heuristic. For this purpose, our extended BFWS(R) uses a new counter $\#g'(s)$ to take into consideration those implicit subgoals that result from state constraints.

We define a set C of atoms which is computed using the same preprocessing $IW(k)$ for $k \in 1, 2$, used to compute R. The set C contains what we call no good atoms.

Definition 6.6.1 *An atom $X = x$ is a no good atoms iff the action plan that makes this atom true violates a state constraint.*

During the $IW(k)$ preprocessing we get a plan for each problem goal. For each of such plans, we add to the set C all atoms that are no good for those goals, and, recursively, for those subgoals of each no good atom which is true in s_0.

Formally, we have a state constraint of the form $\neg\ (p_1\& \ldots \& p_n)$ for atoms $p_i = (X = x)$, and a plan for each literal p and $\neg p$ computed during the IW preprocessing while ignoring state constraints. A state constraint of the form $\neg\ (p_1\& \ldots \& p_n)$ in our case comes from $@non{-}overlap(p_1\& \ldots p_n)$. Then, we say that $\langle p_1, \ldots, p_n\rangle$ is a no good iff there is a state constraint of the form $\neg(p_1\& \ldots \& p_n)$. An atom $p_k : X = x$ is a basic no good atom for goal G if $\langle p_1, \ldots, p_k, \ldots, p_n\rangle$ is a no good made true by some preprocessing plan for G and $p_k \in s_0$. We say that a no good $\langle p_1, \ldots, p_n\rangle$ is a support of atom p_k, where p_k may have more than one support.

We define two different closures added to these basic no good atoms for G. Let G' be a minimal set that includes G and goals $\neg p$ for each basic no good atom for G. The extended no good atoms for G are defined as basic no good atoms for such G', which is uniquely defined. i.e. We initially collect basic no good atoms p for G, then we add $\neg p$ to G and compute the basic no good atoms for the new goals G', until covered. Again as before, each no good atom p has at least one support. A support is a no good where p appears. There is no need to add $\neg p$ as a features or atom in the language, since $\neg p$ is achieved in first state where p is not true.

If $p_k : X = x$ is an extended no good atom for G with no good $\langle p_1, \ldots, p_k, \ldots, p_n\rangle$, then $p'_k : X = x'$ is also an extended no good atom for G if $\langle p_1, \ldots, p'_k, \ldots, p_n\rangle$ is also a no good. The final set C of no good atoms is the resulting set of extended no good atoms for G.

From this set C we can obtain another counter $\#c(s)$. The counter $\#c(s)$ represents the number of atoms in C that are false in s. Meaning, $\#c(s)$ is the number of variables X appearing in C, such that all atoms $X = x$ in C are false in s. Notice that if C contains k atoms $X = x$ in C for the same state variable X, then either k or $k - 1$ of these atoms will always be false. This is because if $X = x$ is true in s, then $X = x'$ for $x'! = x$ will be false. The extended BFWS(R) uses an evaluation function given by novelty measures $w_{\langle \#r(s),\, \#g'(s)\rangle}$, where $\#g'(s)$ = $\#g(s)$ + $\#c(s)$ and breaking ties using the $\#g(s)$ counter and accumulated costs,.

A key element in our BFWS algorithm is the extension of problem states with the extra Boolean features defined explicitly when modeling a problem. Extra features are procedures with a specific symbol, exactly as the rest of the procedures. The difference is that they play an active role for computing the novelty. In addition, the set R is computed based on problem state variables, so it does not take extra boolean features into account. For this reason, we modify the set of atoms R to include those extra features. The set R now contains those atoms achieved from s_0 to each subgoal plus those achieved boolean features during the $IW(k)$ preprocessing. Similarly, now the counter $\#r(s)$ is a counter of the number of atoms and the number of extra features in the set R that are made true at some point from s_0 to s.

6.7 Extensions and Optimizations

As detailed in section 5.4, extra boolean features play an active role when computing the novelty. The fact of computing the novelty of a state s due $\#r(s)$ and $\#g'$ makes a partition of search space in different subproblems. The value of these extra features can make a significant difference in width-based search. In this section we define which extra boolean features has been added.

6.7.1 Additional Features and Heuristics

For *Pick-and-Place* and other domains, we define a set of extra boolean features. They are the ones denoted by symbols @*graspable**, @*placeable**, @*resting_object* and @*holding_at_base*. As seen in section 5.4, @*graspable** and @*placeable** are needed as there are no state variables related to preconditions @*graspable(o)* and @*placeable(o)*, as these symbols denote procedures. The features @*resting_object* and @*holding_at_base* partition the search space into different subproblems.

We will now define these extra boolean features. We have seen that set C contains the no good atoms. An atom $X = x$ belongs to C iff the action plan that makes this atom true violates a state constraint. If we denote $Conf(o) = c$ as atom p, then the procedure @*graspable*(o)* returns true iff $graspable(o) \wedge (p \notin C)$. In other words, *graspable*(o)* returns true iff *graspable(o)* is true and $Conf(o) = c$ is not a no good atom. *placeable*(o)* is true if object o is placeable, o is a goal object and the procedure @*place(Base Arm (Geom ?o) (Conf-g ?o))* returns a goal object configuration. Also it returns true if o is placeable, o is not an obstructing object and $Conf(o)$ = @*place(Base Arm (Geom ?o) (Conf-g ?o))* $\notin C$. @*resting_object(o)* returns true iff $Hold = o \wedge Arm = resting$. @*holding_at_base(o, b)* returns true iff $Hold = o \wedge Base = b$.

For Blocks World domain, there is one new defined extra boolean feature, which is used as action precondition and has been described in section 6.4.2: @*clear(o)*.

For Building Structures Problem, two new extra features have been defined: @*clear(o)*, as in Blocks World problem and @*object-altitude(o, a)* as described in section 6.4.3. Examples of how extra features are defined and computed can be found in `https://goo.gl/67Zibk`.

In addition, we have added another improvement which is a slight modification of the domain independent counter $\#c$. Defining the $\#c$ counter as $\#c(s)$ = *2 * # of atoms is* C *that are true in s + 1 if object being held*, produces better results on planning time. The planning algorithm is still general and domain-independent. This modification of the $\#c$ counter is added as an additional extension.

6.7.2 Extending the Input Language and the Compilation Process

The preprocessing stage seen in section 6.3 that compiles a CTMP problem to a classical planning problem is efficient and compact. However, the initial and goal configurations depend on a discretized ”virtual grid”. The experiments that we report in section 6.8 are generated randomly by setting the initial and goal configurations for both robot and objects, from the set of precomputed real object configurations. In order to define a CTMP problem with the initial and goal configurations of robot and objects, we propose to extend the input language as follows:

We extend the input language I_p defined in section 6.2 by adding the initial and goal (real) configurations for both robot and objects.

Definition 6.7.1 *(Extended Input Language) The extended input of a preprocessing stage is* $I_p = \langle \mathcal{W}, \mathcal{O}, \mathcal{R}, \mathcal{J}, \mathcal{D} \rangle$ *where:*

- A world $\mathcal{W}$ is defined as $\mathcal{W} = \langle \mathcal{Q}, \mathcal{E} \rangle$ as before.
- $\mathcal{O}$ is a set of manipulable objects, such that each $o_s = \langle s, \mu_s, A_s, G_s, m, Q_I, Q_G \rangle$, where the new components stand for:
  - m is the number of objects of type s.
  - Q_I is the set on initial configurations of objects of type s, such that each $q_i = (x, y, z, \theta, \psi, \phi)$, where $i = 1, \ldots, m$.
  - Q_G is the set of goal configurations of objects of type s, such that each $q_i = (x, y, z, \theta, \psi, \phi)$, where $1 \leq i \leq m$.

- A robot $\mathcal{R}$ is a tuple of the form $\mathcal{R} = \langle r_I, r_G, \mathcal{K} \rangle$, where:
  - r_I is the robot base initial configuration $r_I = (x, y, \theta)$. If r_I is empty, the initial base configuration is not taken into consideration.
  - r_G is the robot base goal configuration $r_G = (x, y, \theta)$. If r_G is empty, the goal base configuration is not taken into consideration.
  - $\mathcal{K}$ is the definition of the kinematic model of the robot, as before.
- $\mathcal{J}$ is the set of constraints expressing join limits, as before.
- $\mathcal{D}$ is the set of discretization parameters. The same parameters as described in chapter 6.

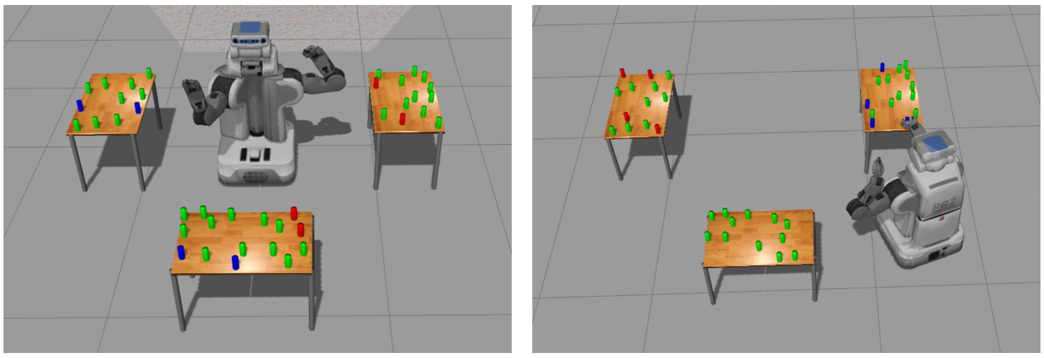


Figure 6.9: *Pick-and-Place* problem in a 3-table environment, initial (left) and goal (right) situations. The objective is to put the blue objects on the rightmost table and the red objects on the leftmost table.

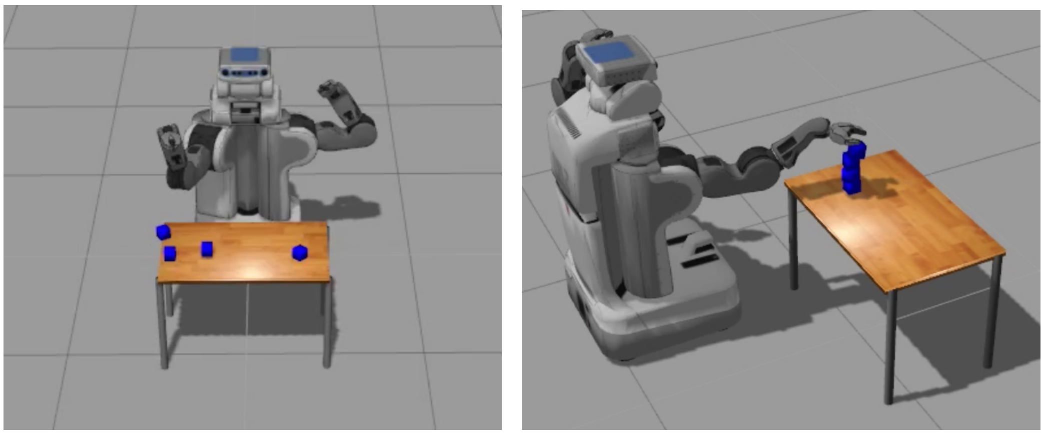


Figure 6.10: An instance of *Blocks World* problem. The goal is to stack all blocks. Left image shows the initial state. Right image shows the goal state.

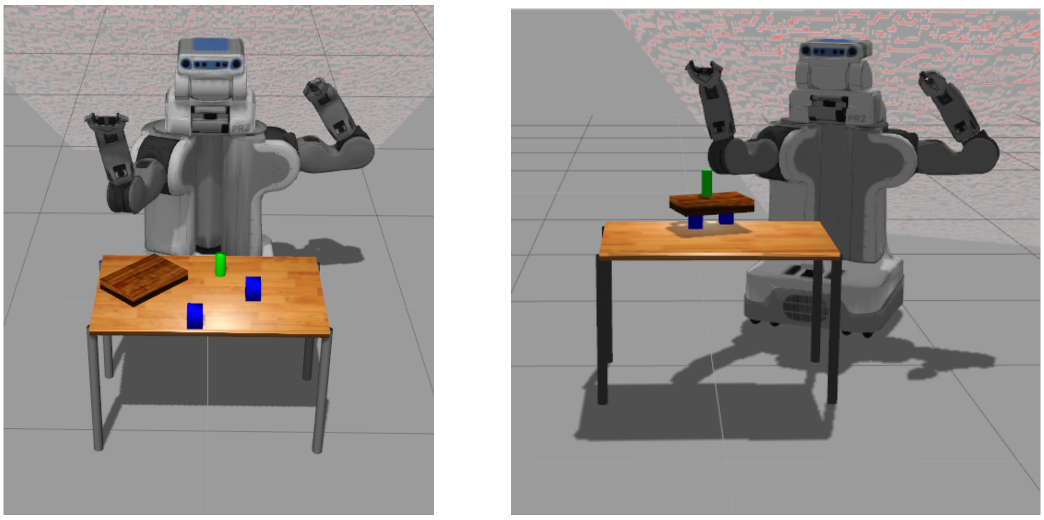


Figure 6.11: An instance of *Structures Building* problem. The left image shows the initial state, with two blocks, one cylinder an one tray. The goal is to reach a specific altitude for the cylinder. Right image shows the goal state. The two blocks must be placed closely to become a stable base for the tray.

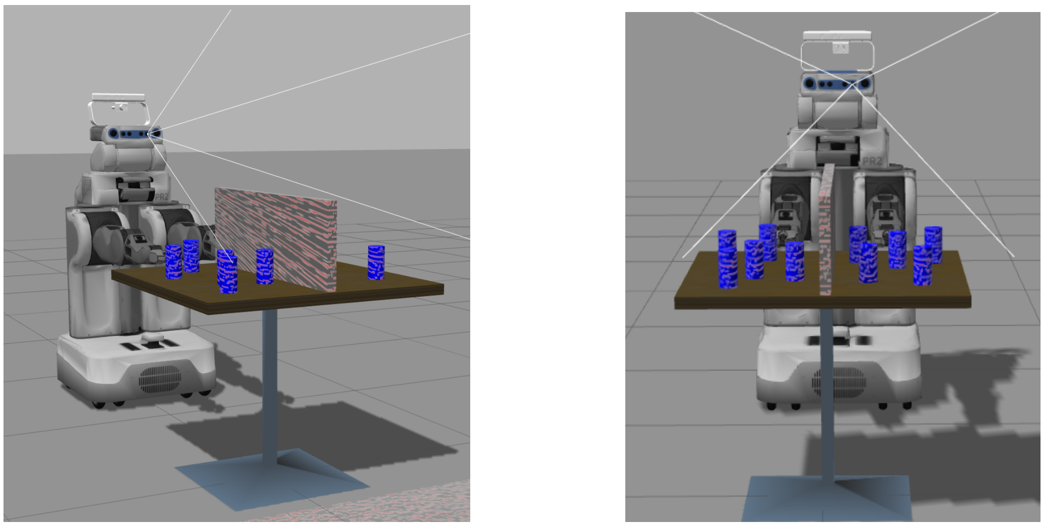


Figure 6.12: An instance of *Pick-and-Place* problem with an obstacle in the middle of the table

6.8 Experimental Evaluation

We evaluate our planning algorithm in a set of different domains: *Pick-and-Place*, *Blocks World* and *Structures Building*. All domains involve either one or three tables. Table 6.4 shows all relevant data about compilation process. Columns show the name of the compiled domains: *Pick-and-Place* with 1 table and 3 tables, and with an static obstacle (obs) in the middle of the table, *Blocks world* with 3, 4 and 5 levels, and *Structures Building* with 3 levels. The other columns show the total number of arm trajectories, arms configurations, base configurations, total number of robot configurations, virtual object configurations, number of virtual grasping poses, relative object configurations, total number of real object configurations and overall compilation time. *Pick-and-Place* with different variations, *Blocks World* for different maximum levels and *Structures Building* have independent compilations. However, it is feasible to do just one compilation and use it for each different domain. For example, the compilation done for *Structures Building* domain can be used to generate *Pick-and-Place* problems, as well as *Blocks World* problems. For illustration purposes, we also show the results of each compilation process individually. As the instances have been generated randomly, we show the planning stats on preprocessed problems using an independent compilation for each type of problem. The CTMP problems are compiled in a timeslot between half a minute and three and a half minutes. The only exception is *Structures Building* problem, which takes almost 24 minutes. The most part of the time is mostly due to overlap tables computation. Checking collisions for more than a thousand arm trajectories in a planning scene with a thousand relative object configurations of different shapes requires a lot of time for MoveIt!. Methods for reduction compilation time of these types of problems are an interesting avenue for future work.

compilation	#traj.	#arms	#bases	#confs	#virt.	#GP	#rel.	#real	T(s)
pick-place-1t	129	40	118	4720	13	39	599	99	72
pick-place-3t	373	40	309	12360	13	39	784	300	182
pick-place-obs	143	45	118	5310	13	44	939	112	102
blocksworld-3l	101	54	31	1674	34	101	232	106	33
blocksworld-4l	318	164	118	19352	102	318	4646	768	138
blocksworld-5l	236	137	119	16303	94	236	6512	802	210
structures-3l	1199	213	31	6603	143	938	960	288	1429

Table 6.4: Compilation data for different domains. *Pick-and-Place* problems with one and three tables. *Pick-and-Place* with an obstacle. *Blocks World* for different levels and *Structures Building*.

The number of trajectories refer to the total number of computed arm trajectories. These trajectories have been computed by MoveIt! [102]. The arm configurations are end-effector poses. The base configurations are sampled and each one is connected to k_B nearest configurations. We select those base configurations which are oriented towards the tables. Thus, the total number of robot configurations are *#arm confs* $\times$ *#base confs*. To review, as defined in section 6.3, the maximum number of virtual object configurations is N_C. There are a maximum number of k grasping poses per each virtual configuration and k' trajectories per each grasping pose. Thus, the maximum number of virtual grasping poses are N_{GP} and up to $N_{GP} \times k'$ arm trajectories. A virtual grasping pose is a grasping pose with the robot located at base $B_0 = \langle 0, 0, 0 \rangle$ that can grasp or place an object located at the related virtual configuration. Thus, the real object configurations are computed using the transformation $T_B(C)$ defined previously. The maximum number of real object configurations are given by the total number of bases multiplied by the total number of virtual object configurations. The real object configurations which do not yield inside a table are pruned. The set of relative object configurations are computed following the transformation $T_B^{-1}(C)$ for all bases B and all real object configurations C. The maximum number of relative object configurations are given by the number of robot bases multiplied by the number of real object configurations. Times from the results are reported in seconds. For each compilation we can generate a number of problem instances. All the reported problem instances have been generated randomly. Unlike the approach presented in chapter 5, we can use objects of different shapes, we can define the allowed rotations and the grasping poses per each different object type. For simplification, in these experiments we assume that objects can be rotated in yaw axis, and pitch and roll are always 0. However, it is easy to define rotation poses for pitch and roll. Experiments without this simplification are reserved for future work.

Table 6.5 defines the discretization parameters $\mathcal{D}$ used per each compilation. From left to right, ϵ is the base (x, y) discretization, α the base orientation discretization, β denotes the maximum absolute orientation of the robot towards a table, k_B is the k nearest neighbors for each discretized base B, σ is the discretization on top of the virtual grid (virtual table), λ denotes the maximum possible number of stacked objects and $|\Delta|$ is the number of direction approaches (trajectories) to a given grasping pose. For all *Pick-and-Place* compilations, all parameters are the same. Notice that $\lambda = 1$, as *Pick-and-Place* compilation only discretizes configuration right above the support surface. For *Blocks World* and *Structures* domain, λ changes to the number of stacking levels. If $\lambda = 3, 4, 5$ then the maximum number of stacked objects are 3,4 and 5. The number of direction approaches $|\Delta|$ changes from the *Pick-and-Place*, being $|\Delta| = 4$ to *Blocks World* and *Structures* being $|\Delta| = 1, 2$. The reason is that having more direction approaches is more suitable for cluttered environments like *Pick-and-Place*.

compilation	ϵ	α	β	k_B	σ	λ	$\|\Delta\|$
pick-place-1t	0.5	0.2618	0.6109	12	0.15	1	4
pick-place-3t	0.5	0.2618	0.6109	12	0.15	1	4
pick-place-1t-obs	0.5	0.2618	0.6109	12	0.15	1	4
blocksworld-3lvls	0.5	0.2618	0.6109	12	0.15	3	1
blocksworld-4lvls	0.5	0.2618	0.6109	12	0.15	4	2
blocksworld-5lvls	0.5	0.2618	0.6109	12	0.2	5	1
structures-3lvls	0.5	0.2618	0.6109	12	0.2	3	1

Table 6.5: Compilation parameters for different domains. It shows the discretization parameters provided as input to the system.

We evaluate our model in a set of different domains with one and three tables. For each domain we generate a number of random instances with increasing number of objects and goals. The first domain is *Pick-and-Place*. We show per instance result in tables 6.6, 6.7, 6.8 and 6.9. Each column shows statistics regarding the use of BFWS(R). The leftmost column shows instance characteristics: Number of objects - number of goals - instance id, $|R|$ is the size of set R, $|C|$ is the size of set C. Remaining columns are the number of expanded nodes, the plan length, simulation time (Prep), search time (Search) and total time (Total). The initial and goal states for a sample problem instance are shown in Fig. 6.9 where the robot needs to place all blue objects on one table and all red objects on another table. Table 6.6 and table 6.7 show the results of our BFWS planner for the *Pick-and-Place* domain with one table, using sets R_G[1] and R_G[2] respectively. The first one, computes set R using a goal oriented $IW(1)$ preprocessing that prunes nodes with novelty greater than 1. Adding additional features: @*graspable**, @*placeable**, *resting_object* and @*holding_at_base* allows to solve each subgoal with width 1 during preprocessing time. Preprocessing time is shown to be fast as IW runs in time exponential in the problem width. We have set a search time limit of 100 seconds. There are four instances that time out, however BFWS with set R_G[2] performs very well during search. Preprocessing time is increased as $IW(2)$ now prunes nodes with novelty greater than 2. This means that nodes with novelty 2 are also expanded. However, not computing boolean features *resting_object* and @*holding_at_base* makes the node expansion faster during search. In both cases, results show to be competitive and scale well with the number of objects. Table 6.8 shows the results of manipulating objects in an environment with 3 tables. Table 6.9 shows the results in *Pick-and-Place* domain with a different variation: there is a small blocking obstacle in the middle of the table as shown in Fig. 6.12. Most of the instances are solved in a few seconds and require only the expansion of a few thousands of nodes in the search tree.

For *Block World* and *Structures Building* tables, each column shows statistics regarding the use of BFWS(R) with set R_G[2]. Preprocessing prunes nodes with novelty greater than 2. The leftmost column reports instance characteristics: Number of objects - number of levels - instance id. Remaining columns are the number of expanded nodes, the plan length, simulation time (Prep), search time (Search) and total time (Total).

Table 6.10 and table 6.11 shows the results in *Blocks World* domain. There are $k = \{3, 4, 5\}$ blocks and a maximum stacking level of $\lambda = k$. This means that there can be up to 3, 4 or 5 blocks forming towers of 3 to 5 blocks. Table 6.10 shows results of *Blocks World* with 3 and 4 levels. Moreover, some instances have 3 towers of 3 blocks each one in the initial configuration. The goal is to set a new tower of 3 blocks. Table 6.11 shows the problems with 5 blocks and 5 levels. Our BFWS performs well with these problems. There are 3 different types of subproblems involving 5 blocks and up to 5 levels:

1. In the first subproblem the goal is to stack all blocks in a tower.

2. The second subproblem is exactly the opposite: There is a stack of blocks in the initial state and the goal is to unstack all of them and place them on top of the table.

3. In the third subproblem all blocks are initially stacked as a tower. The goal is to stack all blocks composing another tower. This type of problem is the most challenging one for our approach.

Table 6.12 shows the results of *Structure Building* domain. There are 3, 4 and 5 objects of different shapes (cylinders, blocks and trays). The goal is to reach a given altitude with some of these objects. The robot must arrange the different objects in order to create a stable structure. Fig. 6.11 shows the initial and goal states of one of the instances. There are two blocks, one cylinder and one tray. The cylinder must be located at some altitude. For this purpose, the robot has moved the blocks in order to place them forming a stable base to stack the tray on top of them. At the same time, the tray composes a stable base to stack the cylinder at the desired altitude.

The Functional STRIPS encodings, tables and videos illustrating the results are available in `https://goo.gl/67Zibk`.

o-g-id	$\|R\|$	$\|C\|$	Expanded	Length	Prep. (s.)	Search(s.)	Total(s.)
10_1_1	8	28	3159	32	1,43	3,01	4,44
10_1_2	8	37	1776	21	0,98	2,77	3,75
10_1_3	10	36	567	36	0,68	0,28	0,96
10_1_4	8	21	593	32	0,64	0,27	0,91
10_1_5	10	72	640	41	1,42	0,32	1,74
10_2_1	13	66	8729	51	1,15	5,77	6,92
10_2_2	11	39	2934	58	1,7	1,74	3,44
10_2_3	13	64	10832	60	1,44	7,51	8,95
10_2_4	9	23	2589	25	0,76	1,25	2,01
10_2_5	0	0	0	0	0	>100	0,00
10_3_1	16	45	1458	71	1,48	0,67	2,15
10_3_2	16	87	14347	63	1,26	10,16	11,42
10_3_3	14	32	5454	83	1,36	7,25	8,61
10_3_4	14	50	12940	73	0,87	11,31	12,18
10_3_5	14	53	8311	64	1,48	8,07	9,55
15_1_1	8	43	4684	47	3,02	10,01	13,03
15_1_2	8	21	1590	26	1,56	2,84	4,40
15_1_3	8	26	1199	22	1,62	0,66	2,28
15_1_4	14	64	672	52	3,09	0,39	3,48
15_1_5	4	0	1157	17	2,89	0,59	3,48
15_2_1	0	0	0	0	0	>100	0,00
15_2_2	0	0	0	0	0	>100	0,00
15_2_3	0	0	0	0	0	>100	0,00
15_2_4	21	107	32174	99	2,95	49,61	52,56
15_2_5	15	39	3010	57	2,45	1,62	4,07
15_3_1	18	82	21947	75	3,01	19,75	22,76
15_3_2	0	0	0	0	0	>100	0,00
15_3_3	14	20	14749	76	2,92	24,47	27,39
15_3_4	0	0	0	0	0	>100	0,00
15_3_5	0	0	0	0	0	>100	0,00
20_1_1	0	0	0	0	0	>100	0,00
20_1_2	0	0	0	0	0	0,02	0,02
20_1_3	14	75	476	28	5,69	0,65	6,34
20_1_4	10	19	1255	15	5,13	1,58	6,71
20_1_5	12	83	1147	47	4,82	1,51	6,33
20_2_1	0	0	0	0	0	>100	0,00
20_2_2	23	145	19137	125	4,92	16,21	21,13
20_2_3	0	0	0	0	0	0,00	0,00
20_2_4	0	0	0	0	0	>100	0,00
20_2_5	19	108	22576	108	4,22	22,80	27,02
20_3_1	0	0	0	0	0	>100	0,00
20_3_2	0	0	0	0	0	>100	0,00
20_3_3	0	0	0	0	0	>100	0,00
20_3_4	0	0	0	0	0	>100	0,00
20_3_5	0	0	0	0	0	>100	0,00

Table 6.6: *Pick-and-Place* problem using set R_G[1].

o-g-id	$\lvert R\rvert$	$\lvert C\rvert$	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
10_1_1	8	28	542	25	3,22	0,14	3,36
10_1_2	8	37	473	21	0,88	0,12	1,00
10_1_3	10	36	1054	36	0,46	0,39	0,85
10_1_4	8	21	775	32	0,38	0,19	0,57
10_1_5	10	72	1179	41	4,38	0,28	4,66
10_2_1	13	66	84749	54	1,8	22,42	24,22
10_2_2	11	39	3987	47	3,97	0,91	4,88
10_2_3	13	64	95070	73	1,77	24,41	26,18
10_2_4	9	23	1064	25	0,48	0,25	0,73
10_2_5	11	56	33641	71	4,41	8,71	13,12
10_3_1	16	45	1670	76	4,38	0,38	4,76
10_3_2	16	87	71074	82	2,11	18,47	20,58
10_3_3	14	32	72868	86	2,9	20,82	23,72
10_3_4	14	50	103887	73	0,44	27,60	28,04
10_3_5	14	53	3233	77	4,37	0,78	5,15
15_1_1	8	43	3447	47	6,14	1,01	7,15
15_1_2	8	21	2806	37	0,77	0,82	1,59
15_1_3	8	26	767	22	1,04	0,21	1,25
15_1_4	14	64	1445	50	7,26	0,38	7,64
15_1_5	4	0	287	17	5,79	0,10	5,89
15_2_1	-	-	-	-	-	>100	>100
15_2_2	9	62	76881	53	0,33	23,86	24,19
15_2_3	17	132	54630	75	2,02	16,84	18,86
15_2_4	21	107	53196	111	6,38	15,55	21,93
15_2_5	15	39	1619	57	2,96	0,41	3,37
15_3_1	18	82	8214	75	6,58	2,12	8,70
15_3_2	14	21	206391	105	5,65	64,35	70,00
15_3_3	14	20	5849	76	5,89	1,60	7,49
15_3_4	-	-	-	-	-	>100	>100
15_3_5	10	72	3611	49	5,4	1,02	6,42
20_1_1	12	46	44726	74	0,21	15,79	16,00
20_1_2	0	0	0	0	0	0,01	0,01
20_1_3	14	75	814	28	4,56	0,25	4,81
20_1_4	10	19	519	33	3,22	0,16	3,38
20_1_5	12	83	511	47	2,51	0,17	2,68
20_2_1	13	49	133505	68	1,76	44,95	46,71
20_2_2	23	145	35013	107	7,71	10,98	18,69
20_2_3	27	160	247957	239	9,85	86,97	96,82
20_2_4	17	90	16634	90	0,83	5,57	6,40
20_2_5	19	108	92173	96	3,82	32,66	36,48
20_3_1	24	85	228286	128	7,1	78,00	85,10
20_3_2	-	-	-	-	-	>100	>100
20_3_3	-	-	-	-	-	>100	>100
20_3_4	16	61	265053	112	3,27	92,98	96,25
20_3_5	20	49	41443	111	8,47	14,31	22,78

Table 6.7: *Pick-and-Place* problem using set R_G[2].

o-g-id	—C—	—R—	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
10_1_1	4	21	1027	17	17,78	0,99	18,77
10_2_2	4	0	694	10	0,33	0,66	0,99
10_3_3	10	0	1609	58	116,31	1,57	117,88
10_4_4	15	40	10853	106	96,59	9,63	106,22
15_1_5	4	0	430	13	1,28	0,90	2,18
15_2_6	7	0	1135	43	61,70	1,26	62,96
15_3_7	12	8	3841	78	150,09	4,05	154,14
15_4_8	15	19	4415	86	118,54	4,19	122,73
20_1_9	6	22	1101	24	15,28	1,21	16,49
20_2_10	7	15	14362	35	9,90	17,26	27,16
25_3_11	18	52	5352	115	96,03	6,27	102,30
25_4_12	-	-	-	-	-	>100	-
30_1_13	6	15	10921	38	65,34	27,43	92,77
30_2_14	11	25	11784	52	17,16	31,06	48,22
30_3_15	24	107	23860	126	206,36	60,94	267,30
30_4_16	-	-	-	-	-	>100	-

Table 6.8: *Pick-and-Place* problem with three tables using the extend version of BFWS(R) with set R_G[2].

o-g-id	—R—	—C—	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
10_1_1	14	68	162	19	0,19	0,05	0,24
10_1_2	4	11	458	14	0,78	0,11	0,89
10_1_3	4	0	57825	28	0,41	15,44	15,85
10_1_4	4	22	470	12	0,16	0,13	0,29
10_1_5	6	44	594	12	0,18	0,14	0,32
10_1_6	6	21	34041	22	0,05	9,05	9,10
10_1_7	8	27	4958	40	0,94	1,23	2,17
10_1_8	6	18	516	22	0,48	0,12	0,60
10_1_9	4	0	154	12	0,2	0,04	0,24
10_1_10	6	52	303	18	0,14	0,08	0,22
15_1_1	10	53	548	26	0,4	0,15	0,55
15_1_2	6	21	517	21	0,69	0,15	0,84
15_1_3	8	62	12686	36	0,39	4,11	4,50
15_1_4	6	6	555	23	1,87	0,15	2,02
15_1_5	20	154	49582	159	0,79	19,47	20,26
15_1_6	10	26	1209	45	5,72	0,31	6,03
15_1_7	8	57	851	33	5,33	0,23	5,56
15_1_8	16	141	1457	42	1,35	0,38	1,73
15_1_9	10	107	1059	37	0,37	0,28	0,65
15_1_10	12	74	2940	65	1,53	0,81	2,34
20_1_1	12	74	43725	67	4,39	16,17	20,56
20_1_2	12	34	1148	39	10,12	0,4	10,52
20_1_3	10	13	1195	52	8,65	0,35	9,00
20_1_4	8	24	850	30	4,04	0,26	4,30
20_1_5	12	102	1661	46	2,3	0,49	2,79
20_1_6	14	92	441907	72	2,72	174,21	176,93
20_1_7	20	125	275291	112	6,23	105,72	111,95
20_1_8	8	69	38932	46	2,42	13,3	15,72
20_1_9	10	61	784	26	0,14	0,24	0,38
20_1_10	12	101	42749	64	0,31	15,04	15,35

Table 6.9: *Pick-and-Place* problem with an obstacle in the middle of the table, using the extend version of BFWS(R) with set R_G[2].

b-l-id	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
Init: All blocks in top of the table Goal: All blocks in a stack					
3_3_1	653	38	0,05	0,05	0,1
3_3_2	14248	101	0,05	1,18	1,23
3_3_3	1156	35	0,05	0,11	0,16
3_3_4	368	19	0,01	0,03	0,04
3_3_5	515	20	0,06	0,04	0,1
3_3_6	2603	27	0,06	0,21	0,27
3_3_7	652	45	0,06	0,05	0,11
3_3_8	7351	39	0,05	0,59	0,64
3_3_9	633	38	0,05	0,05	0,1
3_3_10	40761	23	0,06	3,26	3,32
4_4_1	5221	42	16,78	6,51	23,29
4_4_2	5795	37	17,93	7,74	25,67
4_4_3	15452	54	5,26	23,33	28,59
4_4_4	50628	49	28,42	79,93	108,35
4_4_5	2302	50	8,43	2,14	10,57
4_4_6	7734	37	3,61	11,05	14,66
4_4_7	6240	53	13,60	8,76	22,36
4_4_8	121497	64	15,23	234,72	249,95
4_4_9	6148	28	0,94	8,77	9,71
4_4_10	73596	68	10,82	121,38	132,20
Init: 3 tower of 3 blocks each one Goal: A new tower of 3 blocks					
3-3-1	18626	56	114,10	133,73	247,83
3-3-2	86130	44	476,49	585,98	1062,47
3-3-3	-	-	-	-	>2000
3-3-4	11097	32	65,85	17,26	83,11
3-3-5	17086	65	438,62	36,10	474,72
3-3-6	218396	50	319,42	1422,71	1742,13
3-3-7	129321	52	682,72	783,02	1465,74
3-3-8	-	-	-	-	>2000
3-3-9	-	-	-	-	>2000
3-3-10	235368	56	473,75	1448,50	1922,25

Table 6.10: *Blocks World* problem with towers of 3 and 4 levels and 3 towers of 3 blocks in the initial situation. The used algorithm is the extended version of BFWS(R) with R_G[2].

b-l-id	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
Init:All blocks on top of the table Goal: All blocks in a stack					
5_5_1	51856	64	51,54	64,21	115,75
5_5_2	69189	77	80,81	83,97	164,78
5_5_3	49147	74	23,36	61,24	84,60
5_5_4	19816	69	91,62	20,24	111,86
5_5_5	39308	61	37,71	49,25	86,96
5_5_6	16658	54	38,34	17,31	55,65
5_5_7	73449	70	29,16	98,28	127,44
5_5_8	32443	54	35,99	33,26	69,25
5_5_9	86572	70	13,13	132,18	145,31
5_5_10	78363	92	13,96	92,63	106,59
Init: All blocks in a stack Goal: All blocks in top of the table					
5_5_11	608	35	119,40	0,35	119,75
5_5_12	450	26	97,08	0,32	97,40
5_5_13	461	29	127,67	0,35	128,02
5_5_14	480	30	113,08	0,32	113,40
5_5_15	459	29	45,46	0,28	45,74
5_5_16	542	32	110,51	0,36	110,87
5_5_17	571	30	119,33	0,4	119,73
5_5_18	587	33	128,64	0,4	129,04
5_5_19	553	34	129,44	0,37	129,81
5_5_20	414	28	91,18	0,32	91,50
Init: All blocks in a stack Goal: All blocks in a different stack					
5_5_21	23198	48	249,22	29,73	278,95
5_5_22	527953	70	3,20	948,66	951,86
5_5_23	253562	39	2,77	454,33	457,10
5_5_24	787154	69	324,85	1559,04	1883,89
5_5_25	1042781	82	248,24	1899,31	2147,55
5_5_26	593516	80	253,20	1069,94	1323,14
5_5_27	767364	87	59,35	1247,66	1307,01
5_5_28	190432	39	2,86	278,10	280,96
5_5_29	193339	51	133,86	378,36	512,22
5_5_30	465769	89	316,24	852,55	1168,79

Table 6.11: *Blocks World* problem with towers of 5 blocks, using the extended version of BFWS(R) with R_G[2].

o-g-id	Expanded	Length	Prep. (s.)	Search (s.)	Total (s.)
3_1_1	2192,44	12	8,16	5,39	13,55
3_1_2	1169,70	15	19,68	5,9	25,58
3_1_3	2366,90	12	6,97	5,63	12,6
3_1_4	2366,90	12	6,97	5,63	12,6
3_1_5	2366,90	12	6,97	5,63	12,6
4_1_6	1524	18	27,27	7,16	34,43
4_1_7	49061	25	4,62	272,44	277,06
4_2_8	3249	21	22,24	19,70	41,94
4_1_9	1524	18	27,27	7,16	34,43
4_1_10	49061	25	4,62	272,44	277,06
5_2_11	3249	21	22,24	19,70	41,94
5_2_12	20609	24	21,16	138,67	159,83
5_2_13	20583	24	21,25	142,01	163,26
5_2_14	13174	21	16.35	89.17	105.52
5_2_15	13174	21	16.35	89.17	105.52

Table 6.12: *Structures Building* problem with 2 cubes, cylinders and trays.

6.9 Related Work

Combined task and motion planning for grasping and manipulation is an open research problem at the intersection of planning and robotics. It is not easy to compare empirical results, as there is a lack of standarization regarding a common language to define CTMP problems and common benchmarks to compare different approaches. However, some efforts have been done in order to propose a common set of benchmarks [63].

There are several approaches which addresses the combination of task and motion planners [10, 11, 41, 80, 78, 36], where the decomposition of task and motion planning is not independent, being the task planner influenced by the motion planner. However, approaches based on task and motion decomposition tends to cause lots of backtracking. Problems caused by backtracking and the resulting high number of calls to the motion planner are adressed in [65, 64, 6]. External procedures have also been used to avert the difficulty of performing geometric reasoning within a logically oriented planning language [22, 24, 84], as well as to predict the effects of actions involving complex physics [61]. The integration however is still hierarchical as the interaction between the planner and the external procedures is limited.

The use of classical planners off-the-shelf appears in [100] within a plan-simulate-revise-replan cycle where failed simulated executions are used to revise the symbolic model and where off-the-shelf classical and motion planners are integrated through the use of a planner-independent interface layer. This idea is very appealing but it is far from clear how to make such revisions in general. The errors in the motion planning goals have to be identified and fed back to the symbolic planner in the form of logic predicates, the main challenge being the proper identification of the offending atoms that prevent the enactment of specific high-level actions. FFRob [35] does not use the FF planner off-the-shelf [49] but adapts the computation of the heuristic to the presence of objects using the geometrical information, together with the symbolic information, exploiting a conditional reachability graph as a form of a probabilistic roadmap conditioned to the object configurations.

Hierarchical Task Network [27] approaches are explored in [53], where a hierarchical regression-based schema is developed that combines task and motion planning and in [105], where Hierarchical Task Networks are used to tackle robotic manipulation problems by modeling the bottom actions of the hierarchy with motion planning. Other Hierarchical approaches that control backtracking are addressed in [16, 17, 15].

The use of LTL specification to define hight levels tasks have also been addressed in [89] and [44], where the LTL languages specifies complex task to be performed by motion planners. The idea of using temporal logic specifications it is also addressed for motion planners in [90, 5].

In the aSyMov planner [11, 41], the integration between the symbolic and geometric components is performed after each step of the planning process. Other models appeal to SMT solvers for addressing both task planning and the geometrical constraints [85, 14].

A key characteristic of our approach is the use of a classical planner that do not need to decouple the problem into symbolic and geometric components.

6.10 Discussion

We have proposed a novel approach to fully an automatically compile CTMP problems into classical AI problems. We have proposed a formal language to easily specify task and motion problems. The preprocessing stage understands this language as input and compiles a set of graphs and tables which are directly used during planning time. The fact of compiling such tables, avoids the calls to motion planners and collision checkers during search. We show how to model three different task and motion planning problems: *Pick-and-Place*, *Blocks World* and *Structures Building*, starting from a general encoding which assumes the previous computation of a base and arm graph. The work developed in this chapter extends in several ways the proposed approach of chapter 5: First, the compilation is based on an input which specifies the task and motion problem. We can easily specify the world, the static and the 3D-shape movable objects, the allowed rotation poses and the possible grasping poses per each object type and the discretization parameters to compile the CTMP problem. Second, we show how can based on this compilation, we can model different task and motion problems. Third, we have developed a planning algorithm which is general and domain independent and it makes use of state constraints to compute a set C of no good atoms, which is used to compute the novelty of a state. In addition, we extended the set R of relevant no good atoms seen in [33] to take extra boolean features into consideration. The results have demonstrated the performance of our general algorithm.

CHAPTER 7

The Planner

7.1 General Architecture

Our planner architecture relies on our own developed software and on some off-the-shelf modules which will be described in the following sections. The system architecture shown in Fig. 7.1 describes each component of the architecture and the process work flow. With our proposed input language defined in section 6.2 we start specifying a CTMP problem, which is compiled into a classical planning problem with FSTRIPS and state constraints. We define step by step how our method works:

Input

First, we define an **Input (1)**. This input file is defined as a *JSON* extension file and specifies the object, robot and world geometries, as well as the discretization parameters, following the input language defined in chapter 6.

The main component of our architecture is defined as a ROS package. The package called **CTMP_pkg (2)** contains different ROS nodes with different functionality each one.

Preprocessing Node

Our main node is called **Preprocessing (3)**, and given an input it compiles a CTMP problem as a classical planning problem. The preprocessing node starts the compilation process calling **MoveIt! (4)** to compute motion plans for both

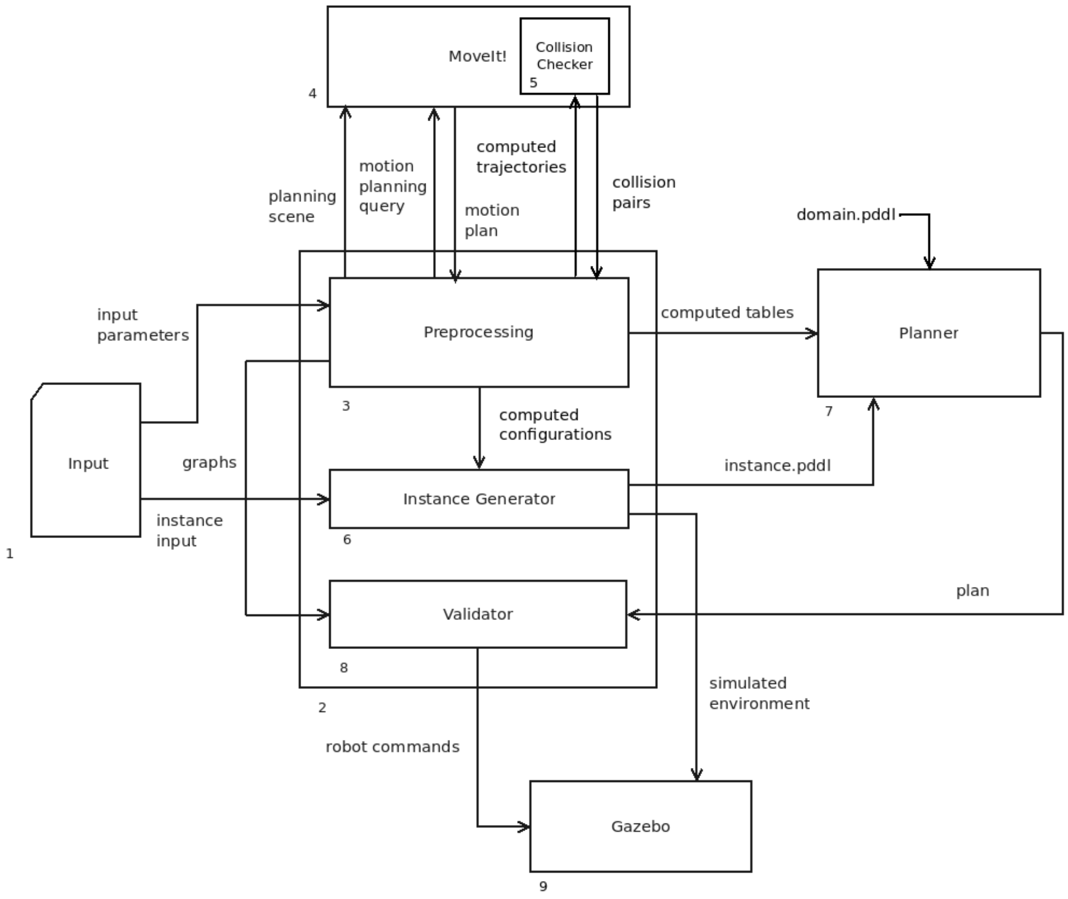


Figure 7.1: General architecture of the system

arm and base movements, and to check for collisions. Once the compilation begins we create what is called a *planning scene*. The planning scene represents the environment with the kinematic model of the robot and the rest of the elements, i.e. movable and static objects. A virtual table representing the robot arm workspace is added to the planning scene. The virtual table is discretized according some parameters specified in the input file. MoveIt! computes both arm trajectories while avoiding collisions with the virtual table and the robot itself and base trajectories while avoiding collisions with the real tables. This step computes two graphs: A base graph and an arm graph. For both graphs, nodes represent configurations, and edges represent trajectories between them. A graph is denoted by a data structure composed by tuples $\langle\langle c_i, t\rangle : c_j\rangle$, where c_i is and c_j are configurations and t is a valid trajectory between them. The preprocessing stage also computes a set of tables which will be used during search. The Preprocessing node takes this input and generates a set of serialized tables per each compilation process. These tables are used for the planner and accessed through external procedures in order to update robot and object configurations, as well to deal with overlaps defined by the state constraints. To compute state constraints, Moveit! uses its **Collision Checker (5)** to check for collisions between previous computes trajectories and relative object configurations, and returns a set of collision pairs.

MoveIt!

Basically, MoveIt! acts as a black box. It creates a planning scene with all the relevant information: Robot, virtual table and support surfaces when computing the base and arm graphs, and objects on relative object configurations when computing overlaps. During the computation of base and arm trajectories, MoveIt! receives queries with the initial and goal configurations for both, base and end-effector. If a motion plan is successfully found, MoveIt! returns a collision-free trajectory. When computing overlaps, we exclusively check for collisions between the computed arm trajectories and objects in relative configurations. For this purpose, the planning scene with the virtual table and the robot is updated to include all objects in relative configurations. Figure 6.3 illustrates a planning scene, with a PR" robot and all the possible objects in relative configurations. All previously computed arm trajectories are checked for collision with the planning scene. MoveIt! returns whether a trajectory collides with some element of the planning scene, and if so, which are the elements in collision. The overlap tables are filled with these collisions.

Instance Generator Node

The **Instance Generator (6)** node takes from the configuration file the type of instance to be generated. This means the number of objects and their shapes and/or the maximum levels for *Blocks World* and *Structures Building* problems and it generates a number of random instances. In order to specify the init and goal configurations, the Instance Generator node takes the computed object configurations generated by the Preprocessing node.

Planner

Once our system has generated graphs, tables and the instances, and the problem has been modeled as a *domain.pddl* file, our **Planner (7)** computes a solution. This solution is composed by a plan file containing symbolic actions. Fig. 7.2 contains a fragment of a plan for the Blockworld problem.

```
transition_base(e218)
transition_base(e10650)
transition_arm(t232)
unstack(o5)
transition_arm(t10232)
transition_arm(t45)
stack(o5)
transition_arm(t10045)
transition_arm(t228)
unstack(o4)
transition_arm(t10228)
transition_arm(t121)
stack(o4)
```

Figure 7.2: A fragment of a plan for a *Blocks World* problem.

Validation Node

The **Validator (8)** node takes the plan and the trajectories computed by MoveIt! for both the base and the arm graph. The plan actions are translated into robot commands to be understood by the controllers of a simulated robot in **Gazebo (9)**. In other words, the Validator translates planning actions to robot actions and display them in Gazebo through ROS messages. An action like *transition_arm(t232)* means that the robot having the arm at some configuration has to perform an arm trajectory which has as an identifier *t232*. It is analogous for base trajectories.

Gazebo Simulator

Gazebo (9) simulates an environment with a PR2 robot. Robot actions given by the planner are translated to robot actions by the Validator node. These robot actions are performed on the simulated PR2. The instance generator spawns the initial set of objects configuration in the simulated environment, as well as the initial robot base and arm configuration.

7.2 Execution of plans

Plans are real robot plans. This means that given a plan π, this plan is executed on a simulated robot. Fig. 7.1 shows the pipeline of the whole process. When the Planner(6) finds a solution, the Validator(7) node translates the high-level planning actions into low level actions or robot commands. These commands are send as ROS messages to a simulated PR2 robot. As detailed before, any ROS-compatible robot with a manipulator can be used. The translation of actions is simple and straight-forward. When the graphs are computed, we store two data structures, one for the arm trajectories and the other one for the base trajectories. Both of them map the trajectory ids to a vector of joint values. The structure for arm trajectories is a function of the form $Traj(id) := \{\langle j_1^1, j_2^1, \ldots, j_n^1\rangle, \ldots, \langle j_1^m, j_2^m, \ldots, j_n^m\rangle\}$, where n is the number of joints and m is the joint vector index. On the other hand, the structure for base trajectories is a function of the form $Traj(id) := \{\langle x^1, y^1, \theta^1\rangle, \ldots, \langle x^m, y^m, \theta^m\rangle\}$. An example of a base trajectory is $\tau(10) = \{\langle 35.2, -169.42, 0.0\rangle, \ldots, \langle 604.2, -312.42, 1.571\rangle\}$, where each tuple represents a configuration base of the form $\langle x, y, \theta\rangle$. For pick-up, place, stack and unstack actions, the corresponding robot command is to open or close the gripper, using a feedback-loop-control to ensure the feasibility of the action.

7.3 Modules Used Off-the-Shelf

In this work, we used a number of tools, libraries and external software. In this section we explain how each of these modules have been used, and why:

- **Robot Operating System (ROS):** ROS [91] is a set of tools and libraries that are widely adopted by the robotics community to develop software for a range of robotic platforms. ROS is an attempt to standarize software development for robotics. It provides a communication system through messages and services, as well as a hardware abstraction and device drivers. It also allows for an easy and direct integration with Gazebo, as well with other simualtors like Morse [25] or V-Rep [95]. The software developed for this dissertation is composed by a ROS package with different nodes. Each ROS node has a different functionality. We used ROS Indigo version on Ubuntu 14.10. The compilation process as well as the visualization have been performed on a simulated PR2 robot in a 3D world and can be easily extended to other robots.

- **MoveIt!:** It is a motion planning framework. It is a software for arm manipulation, kinematics, navigation and control. There are a number of robots which are MoveIt! [102] compatible. MoveIt! is ROS compatible and it is composed by a set of packages and nodes. The main node is *Move Group* and it provides an intuitive interface to use motion planning algorithms in a constructed planning scene with different robot models. It also provides the tools to monitor the planning scene, check trajectories and collisions in a representation of a simulated or real environment. MoveIt! offers a set of motion planning algorithms using the *OMPL* [104] as external library for motion plans, as well as other planners such as *Search-Based Planning Library (SBPL)* [70] and other IK solvers.

- **Open Motion Planning Library (OMPL):** The OMPL [104] is a library that contains implementations for the state-of-the-art sampling-based motion planning algorithms. It is integrated with MoveIt!.

- **Gazebo Simulator:** Gazebo [59] is a ROS compatible robotics simulator. It is widely used by robotics community as it can simulate precise environments with complex robot systems like the PR2. Through and intuitive tag language, SDF, which allows for creation of new elements. It is a powerful physics-engine, recommended for grasping and manipulation tasks with a number of features like sensor data, 3D graphs and the facility to the develop custom plugins. We used Gazebo 2.2 version, which is compatible with ROS Indigo.

7.4 Implementation and Low Level Details

The planner is a set of developed tools that compile a CTMP problem into a classical planning problem. The system has been implemented as a ROS package, where the main ROS node performs the compilation. The planner (solver) is a modified version of the planner presented in [33]. All the implementation of the planner have been done in C++.

The input is provided in JSON format. The Preprocessing computes graphs and tables and store them in a compact and serialized form. The Planner deserialize and loads in memory these graphs and tables, to be accessed by external procedures during search. External procedures are those ones denoted by the symbol @. Each procedure is implemented as a C++ function. Graphs and tables are computed as C++ maps or sets of integers, where each integer represents an id.

PART IV

Conclusions

CHAPTER 8

Conclusions

In this dissertation we presented a novel computational approach that is based in a compilation process of CTMP problems to classical planning problems. We extended the FSTRIPS language to be able to handle state constraints and we proposed a formal input language to describe CTMP problems. We introduced a general and domain independent planning algorithm that can deal with state state constraints, independently of the CTMP problem, by computing weak heuristics from the FSTRIPS and state constraints problem.

8.1 Contributions

In this section we outline the main contributions of this thesis:

1. We proposed a novel integration of task and motion planning where the symbolic and geometrical components are addressed in combination, with neither part taking the back seat. We extended Functional STRIPS with the ability to incorporate state constraints, which are formulas for encoding implicit preconditions. We then presented how an expressive language like FSTRIPS can encode problems which involve symbolic and geometric reasoning, using state constraints to avoid collisions. We proposed a translation of FSTRIPS to STRIPS, by compiling away functions and state constraints. As state constraints encode implicit preconditions, we showed how these state constraints are converted in explicit preconditions for STRIPS encoding.

2. We have presented a framework to fully and automatically compile CTMP problems as classical AI planning problems, given a formal input language to describe problems that require task and motion planning. The compilation process is sound and probabilistic complete, it is independent in the number of objects, and generates a set of graphs and tables, which are being used during the planning stage for avoiding calls to motion planners during search. We showed how to model a set of CTMP problems starting from a general model fragment which assumes a base and arm graph computed during the compilation process.

3. Finally, we presented a general and flexible computational approach based on a domain independent planning algorithm to solve the proposed CTMP problems. The algorithm is a BFWS(R) extended to handle state constraints. The algorithm relies in a $IW(k)$ preprocessing on top of a relaxed version of the original problem where state constraints have been removed. We compute a set C of what we call no good atoms, which are atoms that violate a state constraint. During the $IW(k)$ preprocessing we get a plan for each problem goal and we add to C all no good atoms. We showed how we compute domain-independent heuristics from a problem encoded with FSTRIPS and state constraints. Moreover, we extend the set of relevant atoms taking into consideration the addition of extra boolean features defined on the model of the problem.

8.2 Ongoing and Future Work

Integration of task and motion planning is a wide area in planning and robotics. This dissertation suggests a number of improvements and optimizations, which yield to new lines of research. We are currently working on some of them, while other ones are reserved for future work.

Preprocessing and Compilation Process

The preprocessing process defined in section 6.3 is what we define as an *independent compilation*, since it does not depend on the initial and goal configurations of both objects and robot. When creating instances, the init and goal is taken from the set of precomputed configurations. Given the extended input language of section 6.7, now we can compile a CTMP problem taking into consideration the initial and goal configurations. The process is really simple, and we propose two new compilation types:

- **Init&Goal Preprocessing:** The preprocessing takes the initial and goal configurations for both, objects and robot. Meaning that the compilation process includes these configurations as real configurations. The process works as follows:

  - The initial and goal base configurations, if any, are considered as discretized base configurations and added to the base graph by following the same process as described in section 6.3.
  - Given the set of discretized bases B and all the (real) initial and goal object configurations C, we apply transformation $T_B^{-1}(C)$ to obtain the corresponding relative object configurations C'. These relative object configurations are added to the virtual grid as virtual object configurations. Then, we continue by generating grasping poses and arm trajectories per each discretized and new virtual object configuration C', as usual.

  The preprocessing works normally. The only differences are two: (i) connect the initial and base configurations to the base graph and (ii) transform the init and goal real object configurations to be represented as virtual object configurations. Then the preprocessing continues as described in chapter 5.

- **Incremental Preprocessing:** Extends a previous compilation with a new pair of init/goal object configurations and init/goal base configurations. The base and arm graphs are extended with the new configurations, and previously computed tables are also extended. The incremental preprocessing works as follows:

  - Similarly, the init and goal base configurations are connected to the base graph.
  - Once all bases are generated, the init and goal (real) object configurations are transformed using $T_B^{-1}(C)$ to obtain the virtual object configurations, as defined in previous extension. Then, we generate new grasping poses and arm trajectories per each new virtual object configuration, as usual. Notice that these new set of virtual object configurations is added to the previous compiled set. The table *grasping_poses* is extended with the new generated grasping poses for those new virtual object configurations. The arm graph is extended with the new computed trajectories.
  - The rest of the compilation process works as described in section 6.3. Overlap tables are extended in two ways: 1) checking all trajectories

with the new relative object configurations and 2) checking new trajectories with previously computed relative object configurations.

Additionally, with our approach the robot can grasp an object through different grasping poses and using a number of trajectories, from at least one base configuration. We propose a new method to extend the number of base configurations from where an object can be grasped or placed:

Base Extension: Compiles a new CTMP problem or extends a previous compilation process with the ability to compute new bases from where to grasp and object o being at some configuration $conf(o)$. The idea behind this extension is to be able to grasp and object from more than one base configuration. For this, given the set of real object configurations and the set of grasping poses, we compute new bases from where and arm trajectory yields to a valid grasping pose. The process is repeated k times per each real object configuration. Meaning that there are up to k bases from where the robot can grasp an object. New bases are connected to the base graph. As new bases have been generated, more overlaps are computed.

Finally, preprocessing has demonstrated to be efficient and fast. However, problems which involve a large number of relative object configurations and arm trajectories, like *Structures Building* problems, seems to slow down during overlap checking. We want to address this problem by exploring new methods of computing overlaps. One immediately approach would be to use a general shape for all object types and bounding boxes, to reduce the number of collision checks.

Planning Algorithm

The proposed planning algorithm in chapter 6 is general and fully domain-independent. The reported experiments are far from trivial and show a good performance of the extended BFWS(R) algorithm, being able to handle state constraints when computing novelty. The most of the reported instances are solved in a few seconds at most, even in less than a second for the *Pick-and-Place* problems and some of the *Blocks World* instances. However, there are some challenging problems which require further exploration. Problems involving one or more towers of blocks in the initial state, which goal is to compose another tower, are the most hard ones. The addition of extra boolean features makes a partition of the search space in subproblems and have an impact on the computation of the novelty. Exploring new ways for modeling these type of problems and exploiting the use of new extra boolean futures could improve the results.

Planning with simulations using BFWS(R) algorithm has only access to the structure of states and goals only, while ignoring the action structure. Action effects are returned by an external procedure, as a black box. We want to explore the use of physical simulators and motion planners as a black box. Given a simulated environment and the description of actions with signatures *move-base*, *move-arm*, *grasp* and *place*, a physical simulator can return the feasibility of these actions. This can be fulfilled by applying a width-based search algorithm, without PDDL encoding. The procedures for performing the actions on the simulated environment can simulate robot actions using motion planners, collisions checkers or physic libraries. We have preliminary results in 2D worlds which have not been reported on this thesis because are related to a different approach. There is a wide way for exploring this approach. However, we have to deal with some limitations, as avoiding expensive calls to motions planners and collision checkers per each node expansion.

Validation and Plan Execution

It is known that task and motion planning requires solutions for long term goals, where actions are robot actions. The validation of a plan in a non-deterministic simulator or in board of a real robot may fail as result of accumulated error or external noise. This makes that a previous computed trajectory may fail. Grasping an object of some type is complicate and delicate in the sense that grasping must be really accurate. This dissertation does not focus and does not contribute to grasping motion, but we want to extend our approach to incorporate *plan-execute-refine-replan cycle*, which will allow to refine arm and base trajectories for better graspable poses. The fact of performing an arm translation action ensures that the trajectory is collision-free, and the arm ends up in a pose where the robot can grasp the object by just closing the gripper. In a real robot, if the end effector is not in a proper grasping pose due to some deviation, the grasping (placing) action may fail. Performing a refinement by adjusting the end effector pose with an extra call to a motion planner when the feasibility of the action is not sure would solve the problem.

Experimentation

We reported a number of experiments on different types of problems, for both 2D worlds and 3D worlds. The *Pick-and-Place*, *Blocks World* and *Structures Building* domains are interesting from the integration of task and motion planning point of view. We want to extend the experiments in three different ways:

- Extending the object types with different shapes like other prisms, cups (where the robot has to grasp the object from the handle) or cutlery, to represent more daily environments. Additionally, we want to incorporate more allowed poses for the different object types and use dual arm manipulation for moving the orientation poses of objects.

- We want to extend our problem set with new problem domains. Some challenges, like the ones performed on the Robocup@Home competition[] would be interesting from the experimentation point of view. Setting-up a table or cleaning-up a room are real problems which could be performed with our approach.
- Finally, the experiments have been performed with a simulated robot. We want to apply our approach on a physical robot in a real environment. This requires to adapt our approach with a *plan-execute-refine-replan cycle*.

Part V

Appendix

Bibliography

Bibliography

[1] Albore, A., Palacios, H., and Geffner, H. (2009). A translation-based approach to contingent planning. In *IJCAI*, pages 1623–1628.

[2] Amarel, S. (1968). On representations of problems of reasoning about actions. *Machine intelligence*, 3(3):131–171.

[3] Bacchus, F. and Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27.

[4] Bellemare, M., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: an evaluation platform for general agents. *JAIR*, 47(1):253–279.

[5] Bhatia, A., Kavraki, L. E., and Vardi, M. Y. (2010). Sampling-based motion planning with temporal goals. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2689–2696. IEEE.

[6] Bidot, J., Karlsson, L., Lagriffoul, F., and Saffiotti, A. (2017). Geometric backtracking for combined task and motion planning in robotic systems. *Artificial Intelligence*, 247:229–265.

[7] Bohlin, R. and Kavraki, L. E. (2000). Path planning using lazy prm. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 521–528. IEEE.

[8] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33.

[9] Bylander, T. (1994). The computational complexity of STRIPS planning. *Artificial Intelligence*, 69:165–204.

[10] Cambon, S., Alami, R., and Gravot, F. (2009). A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126.

[11] Cambon, S., Gravot, F., and Alami, R. (2004). aSyMov: Towards more realistic robot plans. In *Proc. ICAPS*.

[12] Cohen, B. J., Chitta, S., and Likhachev, M. (2010). Search-based planning for manipulation with motion primitives. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2902–2908. IEEE.

[13] Cohen, B. J., Subramania, G., Chitta, S., and Likhachev, M. (2011). Planning for manipulation with adaptive motion primitives. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5478–5485. IEEE.

[14] Dantam, N., Kingston, Z., Chaudhuri, S., and Kavraki, L. (2016). Incremental task and motion planning: a constraint-based approach. In *Proc. of Robotics: Science and Systems*.

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

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

[17] de Silva, L., Pandey, A. K., Gharbi, M., and Alami, R. (2013b). Towards combining htn planning and geometric task planning. *arXiv preprint arXiv:1307.1482*.

[18] Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

[19] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

[20] Do, M. B. and Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. In *Proc. ECP 2001*, pages 82–91.

[21] Domshlak, C., Helmert, M., Karpas, E., Keyder, E., Richter, S., Röger, G., Seipp, J., and Westphal, M. (2011). Bjolp: The big joint optimal landmarks planner. *IPC 2011 planner abstracts*, pages 91–95.

[22] Dornhege, C., Eyerich, P., Keller, T., Brenner, M., and Nebel, B. (2010). Integrating task and motion planning using semantic attachments. In *Proceedings of the 1st AAAI Conference on Bridging the Gap Between Task and Motion Planning*, pages 10–17. AAAI Press.

[23] Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. (2009). Semantic attachments for domain-independent planning systems. In *Proc. ICAPS*, pages 114–121.

[24] Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. (2012). Semantic attachments for domain-independent planning systems. In *Towards service robots for everyday environments*, pages 99–115. Springer.

[25] Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., Lesire, C., and Stinckwich, S. (2012). Simulating complex robotic scenarios with morse. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 197–208.

[26] Edelkamp, S. (2006). On the compilation of plan constraints and preferences. In *ICAPS*, pages 374–377.

[27] Erol, K., Hendler, J., and Nau, D. S. (1994). Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128.

[28] Ferrer-Mestres, J., Frances, G., and Geffner, H. (2015). Planning with state constraints and its application to combined task and motion planning. In *Proc. of Workshop on Planning and Robotics (PLANROB)*, pages 13–22.

[29] Ferrer-Mestres, J., Francès, G., and Geffner, H. (2017). Combined task and motion planning as classical ai planning. *arXiv preprint arXiv:1706.06927*.

[30] Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120.

[31] Fox, M. and Long, D. (2003). PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[32] Francès, G. and Geffner, H. (2015). Modeling and computation in planning: Better heuristics from more expressive languages. In *Proc. ICAPS*.

[33] Frances, G., Ramırez, M., Lipovetzky, N., and Geffner, H. (2017). Purely declarative action representations are overrated: Classical planning with simulators. IJCAI.

[34] Garrett, C., Lozano-Pérez, T., and Kaelbling, L. (2014). FFRob: An efficient heuristic for task and motion planning. In *Proc. Int. WAFR*.

[35] Garrett, C., Lozano-Pérez, T., and Kaelbling, L. (2015a). FFRob: An efficient heuristic for task and motion planning. In *Algorithmic Foundations of Robotics XI*, pages 179–195. Springer.

[36] Garrett, C. R., Lozano-Pérez, T., and Kaelbling, L. P. (2015b). Backward-forward search for manipulation planning. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 6366–6373. IEEE.

[37] Geffner, H. (2000). Functional STRIPS: A more flexible language for planning and problem solving. In Minker, J., editor, *Logic-Based Artificial Intelligence*, pages 187–205. Kluwer.

[38] Geffner, H. and Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

[39] Geffner, T. and Geffner, H. (2015). Width-based planning for general video-game playing. In *Proc. AIIDE-2015*.

[40] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: theory and practice*. Morgan Kaufmann.

[41] Gravot, F., Cambon, S., and Alami, R. (2005). aSyMov: a planner that deals with intricate symbolic and geometric problems. In *Robotics Research. The Eleventh International Symposium*, pages 100–110. Springer.

[42] Gregory, P., Long, D., Fox, M., and Beck, C. (2012). Planning modulo theories: Extending the planning paradigm. In *Proc. of ICAPS*.

[43] Hauser, K. (2013). The minimum constraint removal problem with three robotics applications. *The International Journal of Robotics Research*.

[44] He, K., Lahijanian, M., Kavraki, L. E., and Vardi, M. Y. (2015). Towards manipulation planning with temporal logic specifications. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 346–352. IEEE.

[45] Helmert, M. (2006). The Fast Downward planning system. *JAIR*, 26:191–246.

[46] Helmert, M., Röger, G., and Karpas, E. (2011). Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, pages 28–35.

[47] Hoffmann, J. (2003). The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *JAIR*, 20:291–341.

[48] Hoffmann, J. and Nebel, B. (2001a). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[49] Hoffmann, J. and Nebel, B. (2001b). The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302.

[50] Hsu, D., Latombe, J.-C., and Motwani, R. (1997). Path planning in expansive configuration spaces. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 3, pages 2719–2726. IEEE.

[51] Ivankovic, F., Haslum, P., Thiébaux, S., Shivashankar, V., and Nau, D. (2014). Optimal planning with global numerical state constraints. In *Proc. of ICAPS*.

[52] Jiménez, S., Jonsson, A., and Palacios, H. (2015). Temporal planning with required concurrency using classical planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*.

[53] Kaelbling, L. and Lozano-Pérez, T. (2011). Hierarchical task and motion planning in the now. In *Proc. ICRA*, pages 1470–1477. IEEE.

[54] Karaman, S. and Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894.

[55] Karpas, E., Wang, D., Williams, B. C., Haslum, P., et al. (2015). Temporal landmarks: What must happen, and when. In *ICAPS*, pages 138–146.

[56] Kautz, H. A., Selman, B., et al. (1992). Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer.

[57] Kavraki, L. E. and Latombe, J.-C. (1998). Probabilistic roadmaps for robot path planning.

[58] Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580.

[59] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE.

[60] Kuffner, J. J. and LaValle, S. M. (2000). Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE.

[61] Kunze, L. and Beetz, M. (2017). Envisioning the qualitative effects of robot manipulation actions using simulation-based projections. *Artificial Intelligence*, 247:352–380.

[62] Ladd, A. M. and Kavraki, L. E. (2005). Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, pages 233–240.

[63] Lagriffoul, F. (2016). On benchmarks for combined task and motion planning. In *Proc. RSS Workshop on Task and Motion Planning*.

[64] Lagriffoul, F. and Andres, B. (2016). Combining task and motion planning: A culprit detection problem. *The International Journal of Robotics Research*, 35(8):890–927.

[65] Lagriffoul, F., Dimitrov, D., Saffiotti, A., and Karlsson, L. (2012). Constraint propagation on interval bounds for dealing with geometric backtracking. In *Proc. IROS*, pages 957–964. IEEE.

[66] Latombe, J.-C. (2012). *Robot motion planning*, volume 124. Springer Science & Business Media.

[67] LaValle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning.

[68] LaValle, S. M. (2006). *Planning algorithms*. Cambridge.

[69] LaValle, S. M. (2011). Rapidly exploring random tree. `http://msl.cs.illinois.edu/~lavalle/sub/rrt.py`.

[70] Likhachev, M. (2013). Search based planning library. `http://sbpl.net`.

[71] Lin, F. and Reiter, R. (1994). State constraints revisited. *Journal of logic and computation*, 4(5):655–677.

[72] Lipovetzky, N. and Geffner, H. (2012). Width and serialization of classical planning problems. In *Proc. ECAI*, pages 540–545.

[73] Lipovetzky, N. and Geffner, H. (2014). Width-based algorithms for classical planning: New results. In *Proc. ECAI*, pages 1059–1060.

[74] Lipovetzky, N. and Geffner, H. (2017a). Best-first width search: Exploration and exploitation in classical planning. In *Proc. AAAI-2017*.

[75] Lipovetzky, N. and Geffner, H. (2017b). A polynomial planning algorithm that beats LAMA and FF. In *Proc. ICAPS (to appear)*.

[76] Lipovetzky, N., Ramirez, M., and Geffner, H. (2015). Classical planning with simulators: Results on the atari video games. In *Proc. IJCAI-2015*.

[77] Lozano-Perez, T. (1983). Spatial planning: A configuration space approach. *IEEE transactions on computers*, (2):108–120.

[78] Lozano-Pérez, T. and Kaelbling, L. (2014). A constraint-based method for solving sequential manipulation planning problems. In *Proc. IROS*, pages 3684–3691. IEEE.

[79] Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118.

[80] Marthi, B., Russell, S., and Wolfe, J. (2010). Combined task and motion planning for mobile manipulation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

[81] McDermott, D. (1998). PDDL – the planning domain definition language. At `http://ftp.cs.yale.edu/pub/mcdermott`.

[82] McDermott, D. (1999). Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159.

[83] McDermott, D. (2000). The 1998 AI Planning Systems Competition. *Artificial Intelligence Magazine*, 21(2):35–56.

[84] Nebel, B., Dornhege, C., and Hertle, A. (2013). How much does a household robot need to know in order to tidy up? In *Proc. AAAI Workshop on Intelligent Robotic Systems, Bellevue, WA*.

[85] Nedunuri, S., Prabhu, S., Moll, M., Chaudhuri, S., and Kavraki, L. (2014). Smt-based synthesis of integrated task and motion plans from plan outlines. In *IEEE Int. Conf on Robotics and Automation (ICRA)*, pages 655–662.

[86] Palacios, H. and Geffner, H. (2006). Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). In *AAAI*, pages 900–905.

[87] Palacios, H. and Geffner, H. (2007). From conformant into classical planning: Efficient translations that may be complete too. In *ICAPS*, pages 264–271.

[88] Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couëtoux, A., Lee, J., Lim, C., and Thompson, T. (2015). The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*.

[89] Plaku, E. (2012). Planning in discrete and continuous spaces: From ltl tasks to robot motions. In *Conference Towards Autonomous Robotic Systems*, pages 331–342. Springer.

[90] Plaku, E. and Karaman, S. (2016). Motion planning with temporal-logic specifications: Progress and challenges. *AI Communications*, 29(1):151–162.

[91] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe.

[92] Ramirez, M., Lipovetzky, N., and Muise, C. (2015). Lightweight Automated Planning ToolKiT. `http://lapkt.org/`. Accessed November 2015.

[93] Reif, J. H. (1979). Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 421–427. IEEE.

[94] Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 39(1):127–177.

[95] Rohmer, E., Singh, S. P., and Freese, M. (2013). V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE.

[96] Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall. 2nd Edition.

[97] Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning.

[98] Son, T. C., Tu, P. H., Gelfond, M., and Morales, A. (2005). Conformant planning for domains with constraints: A new approach. In *Proc. AAAI*, pages 1211–1216.

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

[100] Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. (2014b). Combined task and motion planning through an extensible planner-independent interface layer. In *Proc. ICRA*, pages 639–646.

[101] Sucan, I., Moll, M., Kavraki, L. E., et al. (2012). The open motion planning library. *Robotics & Automation Magazine, IEEE*, 19(4):72–82.

[102] Sucan, I. A. and Chitta, S. (2013). Moveit! *Online at http://moveit. ros. org*.

[103] Şucan, I. A. and Kavraki, L. E. (2009). Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer.

[104] Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82. `http://ompl.kavrakilab.org`.

[105] Wolfe, J., Marthi, B., and Russell, S. (2010). Combined task and motion planning for mobile manipulation. In *Proc. ICAPS*, pages 254–258.