High level task planning with inference for the TIAGo robot

FINAL MASTER THESIS

Adrià Roig Moreno

Director: **Cecilio Angulo**

Master Degree in Automatic Control and Robotics

aroigmor8@gmail.com

October 6, 2016

PAL ROBOTICS

UPC
UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Abstract

The need to combine task planning and motion planning in robotics is well understood. The task planner generates a plan to solve the problem while the motion planner executes the actions of the problem. The previous framework is applied in many state machines that solve complex problems. But in this project we want to present an interface that communicates the task planner layer and the motion planner layer, and updates the geometric information of the environment to inform the task planner. This framework allows to solve complex tasks with basic information of the goal, and replan whenever the motion could not be executed. All the information of the problems is modelled as logical predicates.

The objective of this project is to generate a generic model of the environment, with a set of feasible motions of the robot, and use this interface to solve many different planning problems involving those actions, by just giving simple goals. The result is to make the robot more autonomous and allow that any user could use it by giving simple orders.

Moreover this project presents the different frameworks and algorithms used to simulate those actions in the robot such as: Sequential Quadratic Programming optimization, Rapidly Random Exploring Tree (RRT) or SBPL global planning. It also shows an introduction to PDDL language used to model the problem and the actions, and the Fast-Froward (FF) solver that is the responsible to translate the problem as a graph and solve it.

Finally we test it on different experiments in simulation, by using the TIAGo platform of PAL robotics. The results are promising and allow to dream in service robots solving complex tasks simply computing and modelling basic actions.

Acknowledgements

Finally I wish to thank all the members of the company, in particular to Mr. Francesco Ferro, the CEO of the company who let me develop the practises in PAL Robotics. This has made me learn a lot at a personal and professional level, in a very pleasant working environment.

But I want to specially thank to Mr Hilario Tome, who his a very patient person that helped me a lot during all the project, developing a lot of the tools and solving many of the problems that arouse.

To end I want to thank the other internships of the company with whom I have worked these months and I have developed a friendship relation.

Contents

Glossary

ADL Action Description Language.. 5, 8

FCL Flexible Collision Library.. 46

FF Fast-Forward solver.. III, 16–20, 44, 47, 61, 62, 69

IK Inverse Kinematics.. 23, 26, 30–33, 41, 44, 46, 49–52, 61, 63, 66

KKT Karush-Kuhn-Tucker.. 23–25

NLP Non Linear Optimization Problems.. 23, 24

OMPL Open Motion Planning Library.. 31

PDDL Planning Domain Definition Language.. III, 5–7, 9–11, 29, 38, 40–42, 44, 45, 47–50, 57, 63, 64

QP Quadratic Programming.. 23, 24, 28

ROS Robotic Operating System.. 4, 33, 35, 37, 47

RRT Rapidly-exploring Random Tree.. III, 22, 31–33, 46, 49–51

SBPL Search-based Planning Laboratory researches methodologies and algorithms that enable autonomous systems to act fast, intelligently and robustly. His research concentrates mostly on developing novel planning approaches, combining planning with machine learning. They investigate on real-time planners ranging from autonomous navigation and autonomous flight to multi-agent systems and to full-body mobile manipulation.. III, 35, 37, 52

SQP Sequential Quadratic Programming.. 23, 26, 30

STRIPS Stanford Research Institute Problem Solver.. 5

1 Introduction

In order to develop high-level tasks, service robots need to combine high-level task planning with low-level motion planning. Task planning determines the strategies or actions that the robot should execute at each step, while motion planning is required to carry out the desired movements. However, combine both is a hard task and a research topic since task planning descriptions usually ignore the strong geometric constraints that exhibit Robotics planning problems. For example, before deciding to pick up a cup, we should determine whether it is geometrically feasible to move the robot base to an appropriate position. Traditionally, such problems have been attacked top-down, separating high-level task planning (e.g., sequencing pick-and-place operations) from lower-level planning (e.g., finding feasible paths for the arm). Task planning is simplified by ignoring low-level details, but the resulting plans may be inefficient or even infeasible due to missed lower level synergies and conflicts.

The main contribution of this project is to develop a general framework for the TIAGo robot that allow us to solve different kind of planning problems by combining task planning and motion planning such as in the work of [20] [3]. Our interface should allow us to generate plans in a robust manner, reacting and re-planning when facing with unexpected changes. Relevant geometric information of the environment is transmitted to the task planner in terms of logical predicates. The task planning layer analyses if the preconditions necessary to execute the actions are satisfied, and extracts a plan, while the motion planner layer executes the plan.

But sometimes at the beginning we don't dispose of all the geometric information, or even the environment changes. To adapt to those changes, instead of use a perception layer to update the information, the motion planner translates all this information into symbolic form to the task planner. The inference step could be seen as a trial error in which the motion plan tries to execute the plan, and informs the planner when it fails. The interface translates this information to the task planner that tries to find a new plan.

In our simulations we work with environments full of obstacles. Once the motion planner finds that an object is impeding another, updates the information and calls the task planner. The task planner calls again the solver with the new states. Iteratively the same process is repeated until the motion planner executes a valid plan.

The main purpose of this project is to develop an intelligence based framework which allows the robot to work autonomously by executing a set of basic actions. The robot should be capable of develop high-level tasks defined with clear and simple goals, by just sequencing low level tasks.

The following sections describe the used platform, the language used for the Task Planner to model all the geometric information, the implementation of our interface, and the resolution of multiple planning problems. In the implementation of the interface, is detailed how it was developed a simulator to visualise the problem, and the tools necessary to create new problems or environments; how were implemented the basic motion actions for the robot; and how those actions are modelled as logical predicates in the language of the Task Planner.

2 The TIAGo robot

The platform used to develop this project is the TIAGo robot from PAL Robotics, a Spanish company situated in Barcelona around 2004. The area of activity of the company is mainly oriented to robotics with branches in electronics, software development and mechanics. His objective is to create and develop robots for service and research.

The TIAGo robot is one of the newest robots of the company. It is a one hand robot mounted on a mobile base that can manipulate objects. The robot has a 7-DOF arm that can be mounted with different interchangeable end effectors, with a maximum payload of 2Kg. The mobile base is equipped with differential drive wheels and a navigation laser that allows the robot to navigate and mapping avoiding obstacles. The robot also includes a lift torso providing it a long reach for manipulation. TIAGo uses Ubuntu Linux LTS as operating system, and could be "controlled" using ROS.

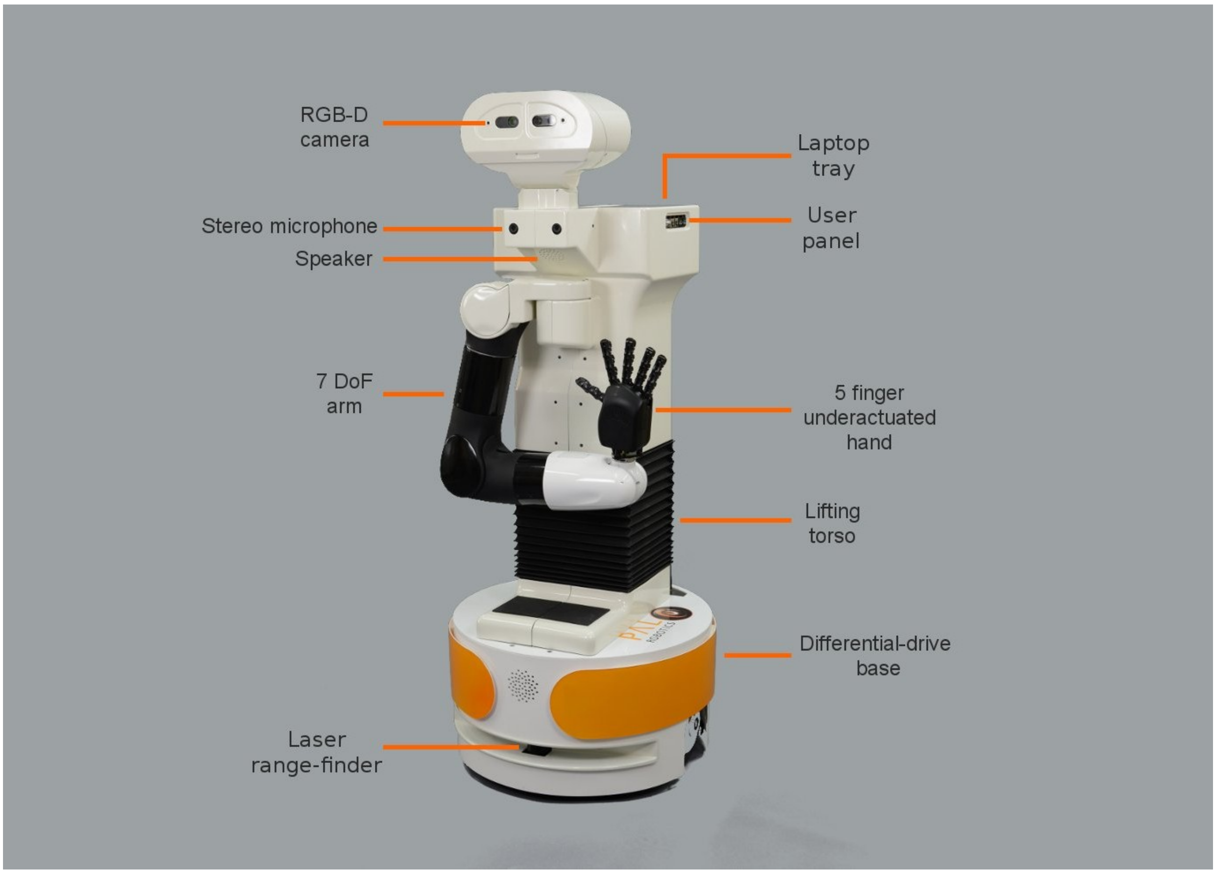


Figure 2: The TIAGo robot from PAL

3 The Planning Domain Definition Language

This section introduces to the Planning Domain Definition Language and all of his formalisms.

3.1 Introduction

The Planning Domain Definition Language (PDDL) [15] is a planning language first developed by Drew McDermott in 1998, as an attempt to standardise previous existing Artificial Intelligence planning languages. All those planning languages are used to model a given planning problem with certain conditions, and then automatically generate a chain of actions that with lead to a desired goal which can be expressed as a set of conditions. The described actions will also depend on conditions, which need to fulfilled in order to be executed, and will produce effects in the environment once executed.

The first developed planning language was STRIPS (Stanford Research Institute Problem Solver) developed in 1971 by Richard Fikes and Nils Nilsson, and nowadays is the starting point for most of the languages for expressing automated planning problems. Later on Pednault (an IBM researcher specialist in the field of Data abstraction) in 1987 proposed the Action description language (ADL) which is considered an evolution of STRIPS. Nowadays there exists a high variety of planning languages and improvements of the original PDDL language that allow to work with continuous actions, probabilistic planning problems or even with multi-agent problems. Although we will only focus on STRIPS and ADL, since they are the most common and used languages.

3.2 Formulation

In the Planning Domain Definition Language the planning task that we want to solve is expressed as a triple $\langle O, s_0, g \rangle$, where O is the set of parametrised propositional actions defined by preconditions and effects, s_0 is the initial state, and g is the goal. Each state s including the initial state and the goal are expressed as a set of logical atoms or conditions. At the same time the actions are defined as a triple $\mathbf{o} = \langle \mathbf{pre(o)},\ \mathbf{add(o)},\ \mathbf{del(o)} \rangle$; where $\mathbf{pre(o)}$ represent the preconditions necessary to apply the action, $\mathbf{add(o)}$ is the list of added effects once the action is executed, and $\mathbf{del(o)}$ is the list of deleted effects. We say that a single condition or atom f is achieved by the action if $f \in \mathbf{add(o)}$. Then we say that an action is applicable if our current state s fulfils $s \subseteq \mathbf{pre(o)}$. The result of applying action o_i in state s_i generates a new state s_{i+1}, such that $\boldsymbol{s_{i+1}} = \boldsymbol{o_i}(\boldsymbol{s_i})$. Our objective is to find a given a sequence of actions $\boldsymbol{o_0}, ..., \boldsymbol{o_n}$, such that $\boldsymbol{s_i}$ satisfies the preconditions for $\boldsymbol{o_i}$, and at the same time allows us to move from the initial state $\boldsymbol{s_0}$ to the goal state $\boldsymbol{g}$ (see Equation 1)

$$g = o_n ... o_0(s_0) \qquad \text{with } s_i \subseteq \text{pre}(o_i) \quad \forall i = 0, ..., n \tag{1}$$

Arrived at this point where all the basic concepts surrounding PDDL are defined, the next step is to describe the basic syntax of the language in order to understand further examples. All the planning tasks defined in PDDL are separated in two different files that contain:

- Objects: Represent "the things" involved in the problem. It could be for a example a robot, an object or a position.
- Predicates: Are the conditions or atoms that describe the problem, and represent the properties of the objects. Although PDDL supports quantified conditions, we be focus only on true or false conditions.
- Initial state: State from where we begin.
- Goal: State we want to reach.
- Actions: Ways of changing the state of the world.

In the domain file we write all the predicates and actions. In the problem file we specify the objects, initial state and goal conditions. The objective is to create a generic domain file, linked with specific actions for the robot that let us solve a large amount of different problems by just modifying the objects and/or the initial and goal states.

We start by explain the domain file containing the actions, that should be written according the following structure.

```
(define (domain domain_name)
    (:requirements :adl :typing ...)
    (:types a b ...)
    (:constants ct1 ...)
    (:predicates (pred1 ?x - a) (pred2 ?x - a ?y - b) ...)

    (:action action_name
        :parameters(?param1 - a)
        :precondition (pred1 ?param1)
        :effect (pred2 ?param1)
    )
    ...
)
```

This file must contain:

- The domain name. Is the name of our problem. Has to be the same as the one in the problem file.
- The list of requirements. Because PDDL is a very general language and most solvers support only a subset of rules, domains may declare requirements such as the language we are using (strips or adl) or if for example we are including types in our files.
- The types of objects. Is not mandatory, and can be substituted by a list of predicates such as *(is_type_a ?obj)*, *(is_type_b ?obj),...* Is used to clarify and to avoid writing a bigger list of preconditions for the actions.
- The constants, that are PDDL objects that exist even considering new environments, or different conditions. Must be declared when they are used as a parameter in the preconditions or effects of an action. As for example if after apply some action we end always at a fixed position. If we use types, the constants can be defined as *ct1 - a* i.e ct1 is of type a.
- The predicates of the problem define the properties and the geometric constraints of the PDDL objects as logical conditions. Having unnecessary predicates will consume more time for the solver, and having less predicates than the necessaries won't allow us to solve the problem, and can give us a false set of actions. The predicates are defined as *pred_name? param1 param2 ...* with a given name and a list of parameters that represent the objects associated. A predicate could be as simple as *(is_free? arm)* which tell us if the arm of the robot is free. Or for example *(block? obj1 obj2 pos1 pos2)* which says that the obj1 is blocking to the obj2 in the position pos1 for obj1 and pos2 for obj2. If we define types is possible to define each type of object in the description of the parameters.
- The actions. They must be identified by a given name. Each action has a list of parameters that are used in the preconditions or effects. Constants don't need to be passed. Each action contain the list of conditions (as predicates) necessary to be applied and the effects of the predicates once is applied.

The second file is called problem file and contains the objects that are not constants, the initial conditions and the final conditions of the problem.

```
(define (problem problem_name)
   (:domain domain_name)
   (:objects x - a ...)
   (:init
   (pred1 x)
   ...
   )
   (:goal
   (pred2 x)
   )
)
```

Our purpose is to solve planning problems with just modifying the start conditions and the goal. That's why the domain file remain fixed for every problem while we generate a different problem file for each situation. The initial conditions and the goal conditions are expressed as a set of predicates. In strips all those predicates that are not initialised are considered as false while in ADL are considered as unmentioned. The objective is to get a list of actions that set the predicates of our problem from the initial conditions to the desired ones.

The following list describes a list of tools of the language, necessary to understand further examples.

- **and** - Allows to link several conditions. For example if we have *(and (pred1 ?x) (pred2 ?y))* in the preconditions of an action it will be only applied when both predicates are true. It's equivalent to the logic symbol AND ($\bigwedge$).

- **not** - Allows to force a precondition or effect for a given predicate to be negative. For example if we want to set an object as not free then we write *(not (free ?x))*. It's equivalent to the logic symbol $\neg$.

- **or** - Allows to check if a condition or another is satisfied as a precondition of a given action. For example if our precondition is *(or (pred1 ?x) (pred2 ?y))* for a given action, then it's enough that it satisfies one of the two predicates. Is equivalent to the logic symbol or $\bigvee$.

- **forall** - Is used to evaluate or set an action for a set of objects. For example if we want to see if *(pred1 ?x -a ?y - b)* is satisfied by all the objects of type a.
- **when** - Is used in the effects of the action to set a predicate once a condition is satisfied. For example *(when (pred1 ?x) (pred2 ?y))* when pred1 is satisfied for x then pred2 y will be true.
- **imply** - Is equivalent to when but is only used in the preconditions.

3.3 Examples

Now that the basic tools for understand most of the PDDL problems have been explained, we will present a set of classical planning examples in order to familiarise the reader with the language.

3.3.1 The Hanoi Problem

The Hanoi problem is a logic game in which a set of discs of different size in a stick (ordered from the biggest one to the smallest one) should be translated to another stick maintaining the same order. As a condition we can not put a bigger disc on top of a smaller one, and we can only take discs from the top of the heap. Figure 3 shows the Hanoi problem with three discs and three sticks represented as a graph. Each node represent a feasible configuration of the problem, while the edges represent the possible movement at each configuration. The following Code 1, Code 2 represents his implementation in PDDL.

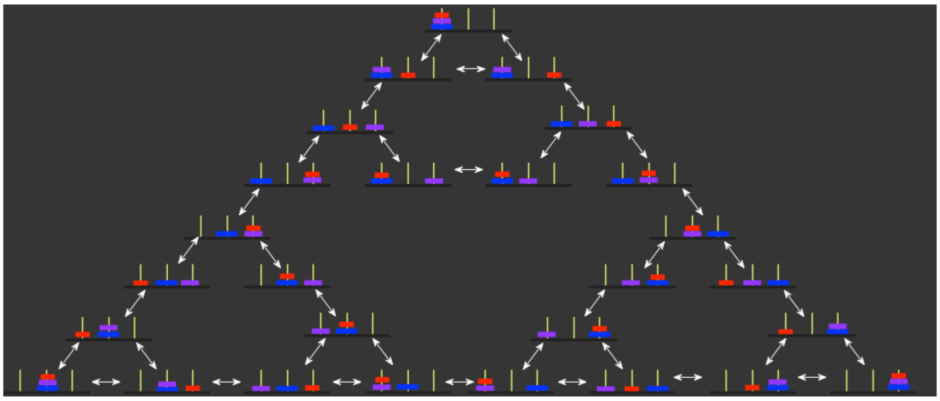


Figure 3: Hanoi Problem represented as a graph

Listing 1: Domain file

```
(define (domain hanoi)
  (:requirements :strips)
  (:predicates (clear ?x) (on ?x ?y) (smaller ?x ?y))

  (:action move
    :parameters (?disc ?from ?to)
    :precondition (and (smaller ?to ?disc) (on ?disc ?from)
                              (clear ?disc) (clear ?to))
    :effect  (and (clear ?from) (on ?disc ?to) (not (on ?disc ?from))
                    (not (clear ?to))))
  )
```

Listing 2: Problem file

```
(define (problem hanoi3)
  (:domain hanoi)
  (:objects peg1 peg2 peg3 d1 d2 d3)
  (:init
   (smaller peg1 d1) (smaller peg1 d2) (smaller peg1 d3)
   (smaller peg2 d1) (smaller peg2 d2) (smaller peg2 d3)
   (smaller peg3 d1) (smaller peg3 d2) (smaller peg3 d3)
   (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
   (clear peg2) (clear peg3) (clear d1)
   (on d3 peg1) (on d2 d3) (on d1 d2))
  (:goal (and (on d3 peg3) (on d2 d3) (on d1 d2)))
  )
```

In the example we have three sticks (peg1, peg2, peg3) and three discs (d1 d2 d3) which are objects from PDDL. To model the problem is enough with three predicates.

- smaller - Compares the disc size. If disc d1 is smaller than d2, then (smaller d2 d1) is true. Is not commutative with the parameters.

- clear - Says if a disc or stick is available to move it, or receive another disc on top.

- on - Says if a disc is on top of another disc or a stick. It's not recursive in the sense that if d1 is above d2 and d2 is above d3, the predicate (on d1 d3) will be false. Like smaller is not commutative with the parameters.

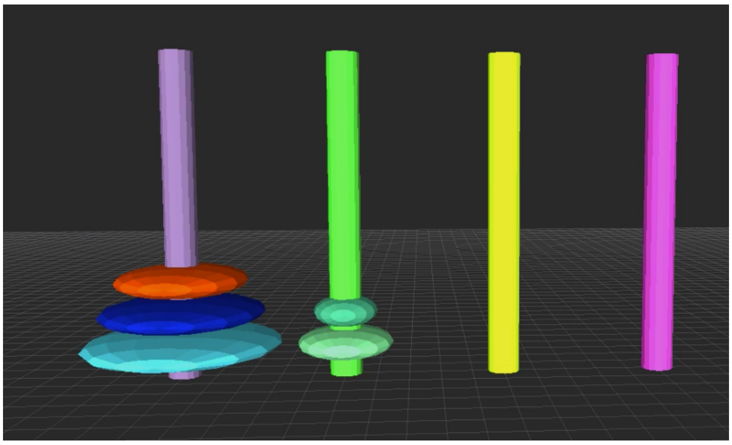


Figure 4: Hanoi problem visualisation with markers

In the domain file we describe the predicates and actions of the problem. There is only one action called move disc, that requires that the disc that want to move is clear (clear disc); the object where we want to move is clear (clear to); the object where we will place the disc on top must be necessary of a bigger size *(smaller ?to ?disc)* and the disc that we want to move must be on top of another object *(on ?disc ?from)*. This last precondition is used to check that the parameter from corresponds to the current position of the object, in order to set it as false in the effects. Once the action is applied the object where we come from has to be clear *(clear ?from)*, and the object where we place the disc is no more clear. *(not (clear ?to))*. Finally we also have to consider that the disc we are moving is not anymore where it was i.e *(on ?disc ?to) (not (on ?disc ?from))*. We start with the three discs in peg1 and we want to finish with all the discs in the same order but in peg3.

Notice that there doesn't exists a unique formulation for a problem, or a given technique to model it. It is only necessary to list all the properties or situations describing a problem and try to adapt the actions with the effects and predicates.

3.3.2 The Trucks problem

The following example is a classic problem that shows part of the potential of the PDDL language. Moreover in this example we will see a lot of tools explained in the previous section.

In the general problem a set of trucks has to pick and deliver several packages at different locations at different times. Our objective is to deliver some of the packages at time in specific locations. As feasible actions we can pick objects from our current position, unload objects from our current position, drive from a location to another, or deliver a package. The trucks has a size called truck-area that specifies the number of objects that can be load in a truck.

To model the problem the following predicates have been used:

- at - Used to locate a truck into a given location.
- in - Says if a package is inside the truck in a given position.
- connected - True when both locations are connected.
- free - Says if a position inside the truck is free.
- time-now - Specifies the current time that is updated after executing some actions.
- next - Used to specify the following time. For example we could specify that after t3 comes t4.
- le - Says if a time is "less or equal" than another one.
- delivered - Specifies that a package has been delivered at a given location at a specific time.
- at-destination - Specifies that a package is at a given location.
- closer - Used to "organize" the positions inside the truck. For example if we put some package close to the exit, we will disturb us from putting any other package inside the truck even when other positions are empty.

In the problem represented in Code 4, we consider only a single truck with two spaces on his trunk, and a set of three packages that has to be delivered. In the initial conditions we also specify the times as objects and we also set the connection between locations. Then our objective is two deliver two packages at a given time, and the other package has no time specified.

To solve the problem the four actions are modelled in Code 3.

- To pick/load an object we need to be at the current position of the package, and we will need at least one free position in the truck. The predicate *(forall (?a2 - truckarea) (imply (closer ?a2 ?a1) (free ?a2 ?t)))* specifies that no other package can block the position where we want to place our package. Once the action is applied, the package won't be at the given location since it will be inside the truck. The position inside the truck won't be free until we unload the package.

- The action unload is quite similar to the previous one, denying or inverting the precondition and effects. Again we have that we use the forall and imply tools to model that we can't extract an object from a given position when some other objects are blocking it.
- The drive action is the simplest one since it checks the current time, the current truck, and if there exists a connection between both locations. It increases one unit of time and locates the truck into another location.
- The action deliver checks that the deliver time must be less or equal than the current time, and checks the current position of the truck. Once applied set that the package has bee delivered at destination at time.

Listing 3: Domain file

```
(define (domain Trucks)
(:requirements :typing :adl)

(:types truckarea time location locatable - object
        truck package - locatable)

(:predicates (at ?x - locatable ?l - location)
             (in ?p - package ?t - truck ?a - truckarea)
             (connected ?x ?y - location)
             (free ?a - truckarea ?t - truck)
             (time-now ?t - time)
             (next ?t1 - time ?t2 - time)
             (le ?t1 - time ?t2 - time)
             (delivered ?p - package ?l - location ?t - time)
             (at-destination ?p - package ?l - location)
             (closer ?a1 - truckarea ?a2 - truckarea))

(:action load
 :parameters (?p - package ?t - truck ?a1 - truckarea ?l - location)
 :precondition (and (at ?t ?l) (at ?p ?l) (free ?a1 ?t)
                    (forall (?a2 - truckarea)
                            (imply (closer ?a2 ?a1) (free ?a2 ?t))))
 :effect (and (not (at ?p ?l)) (not (free ?a1 ?t)) (in ?p ?t ?a1)))

(:action unload
 :parameters (?p - package ?t - truck ?a1 - truckarea ?l - location)
 :precondition (and (at ?t ?l) (in ?p ?t ?a1)
                    (forall (?a2 - truckarea)
                            (imply (closer ?a2 ?a1) (free ?a2 ?t))))
 :effect (and (not (in ?p ?t ?a1)) (free ?a1 ?t) (at ?p ?l)))

(:action drive
 :parameters (?t - truck ?from ?to - location ?t1 ?t2 - time)
 :precondition (and (at ?t ?from) (connected ?from ?to)
                    (time-now ?t1) (next ?t1 ?t2))
 :effect (and (not (at ?t ?from)) (not (time-now ?t1))
              (time-now ?t2) (at ?t ?to)))

(:action deliver
 :parameters (?p - package ?l - location ?t1 ?t2 - time)
 :precondition (and (at ?p ?l) (time-now ?t1) (le ?t1 ?t2))
 :effect (and (not (at ?p ?l)) (delivered ?p ?l ?t2)
          (at-destination ?p ?l)))
)
```

Listing 4: Problem file

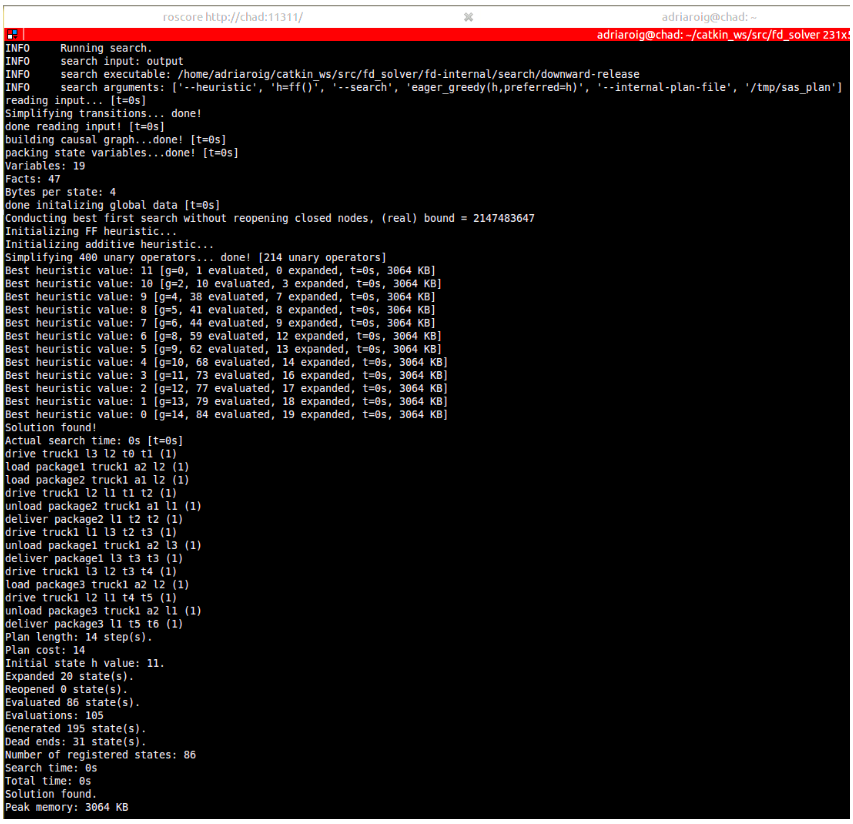
```
(define (problem truck-1)
(:domain Trucks)
(:objects
        truck1 - truck
        package1 - package      package2 - package
        package3 - package
        l1 - location l2 - location      l3 - location
        t0 - time t1 - time t2 - time t3 - time
        t4 - time t5 - time t6 - time
        a1 - truckarea a2 - truckarea)

(:init
        (at truck1 l3) (free a1 truck1) (free a2 truck1)
        (closer a1 a2) (at package1 l2) (at package2 l2)
        (at package3 l2) (connected l1 l2) (connected l1 l3)
        (connected l2 l1) (connected l2 l3)     (connected l3 l1)
        (connected l3 l2) (time-now t0) (le t1 t1)
        (le t1 t2) (le t1 t3) (le t1 t4) (le t1 t5) (le t1 t6)
        (le t2 t2) (le t2 t3) (le t2 t4) (le t2 t5) (le t2 t6)
        (le t3 t3) (le t3 t4) (le t3 t5) (le t3 t6) (le t4 t4)
        (le t4 t5) (le t4 t6) (le t5 t5) (le t5 t6) (le t6 t6)
        (next t0 t1) (next t1 t2) (next t2 t3) (next t3 t4)
        (next t4 t5) (next t5 t6))

(:goal (and
        (delivered package1 l3 t3)
        (at-destination package2 l1)
        (delivered package3 l1 t6)))
)
```

For this specific problem the solver give us the following list of actions.

1. drive truck1 l3 l2 t0 t1
2. load package1 truck1 a2 l2
3. load package2 truck1 a1 l2
4. drive truck1 l2 l1 t1 t2
5. unload package2 truck1 a1 l1
6. deliver package2 l1 t2 t2
7. drive truck1 l1 l3 t2 t3
8. unload package1 truck1 a2 l3
9. deliver package1 l3 t3 t3
10. drive truck1 l3 l2 t3 t4
11. load package3 truck1 a2 l2
12. drive truck1 l2 l1 t4 t5
13. unload package3 truck1 a2 l1
14. deliver package3 l1 t5 t6


roscore http://chad:11311/
adriaroig@chad: ~
adriaroig@chad: ~/catkin_ws/src/fd_solver 231x

```
INFO     Running search.
INFO     search input: output
INFO     search executable: /home/adriaroig/catkin_ws/src/fd_solver/fd-internal/search/downward-release
INFO     search arguments: ['--heuristic', 'h=ff()', '--search', 'eager_greedy(h,preferred=h)', '--internal-plan-file', '/tmp/sas_plan']
reading input... [t=0s]
Simplifying transitions... done!
done reading input! [t=0s]
building causal graph...done! [t=0s]
packing state variables...done! [t=0s]
Variables: 19
Facts: 47
Bytes per state: 4
done initalizing global data [t=0s]
Conducting best first search without reopening closed nodes, (real) bound = 2147483647
Initializing FF heuristic...
Initializing additive heuristic...
Simplifying 400 unary operators... done! [214 unary operators]
Best heuristic value: 11 [g=0, 1 evaluated, 0 expanded, t=0s, 3064 KB]
Best heuristic value: 10 [g=2, 10 evaluated, 3 expanded, t=0s, 3064 KB]
Best heuristic value: 9 [g=4, 38 evaluated, 7 expanded, t=0s, 3064 KB]
Best heuristic value: 8 [g=5, 41 evaluated, 8 expanded, t=0s, 3064 KB]
Best heuristic value: 7 [g=6, 44 evaluated, 9 expanded, t=0s, 3064 KB]
Best heuristic value: 6 [g=8, 59 evaluated, 12 expanded, t=0s, 3064 KB]
Best heuristic value: 5 [g=9, 62 evaluated, 13 expanded, t=0s, 3064 KB]
Best heuristic value: 4 [g=10, 68 evaluated, 14 expanded, t=0s, 3064 KB]
Best heuristic value: 3 [g=11, 73 evaluated, 16 expanded, t=0s, 3064 KB]
Best heuristic value: 2 [g=12, 77 evaluated, 17 expanded, t=0s, 3064 KB]
Best heuristic value: 1 [g=13, 79 evaluated, 18 expanded, t=0s, 3064 KB]
Best heuristic value: 0 [g=14, 84 evaluated, 19 expanded, t=0s, 3064 KB]
Solution found!
Actual search time: 0s [t=0s]
drive truck1 l3 l2 t0 t1 (1)
load package1 truck1 a2 l2 (1)
load package2 truck1 a1 l2 (1)
drive truck1 l2 l1 t1 t2 (1)
unload package2 truck1 a1 l1 (1)
deliver package2 l1 t2 t2 (1)
drive truck1 l1 l3 t2 t3 (1)
unload package1 truck1 a2 l3 (1)
deliver package1 l3 t3 t3 (1)
drive truck1 l3 l2 t3 t4 (1)
load package3 truck1 a2 l2 (1)
drive truck1 l2 l1 t4 t5 (1)
unload package3 truck1 a2 l1 (1)
deliver package3 l1 t5 t6 (1)
Plan length: 14 step(s).
Plan cost: 14
Initial state h value: 11.
Expanded 20 state(s).
Reopened 0 state(s).
Evaluated 86 state(s).
Evaluations: 105
Generated 195 state(s).
Dead ends: 31 state(s).
Number of registered states: 86
Search time: 0s
Total time: 0s
Solution found.
Peak memory: 3064 KB
```

Figure 5: Solution of the trucks problem using the FD solver

Just changing the initial conditions, for example including more trucks or more packages, we are solving different problems, all of them modelled in a single file. Adding this to the fact that the solver is really fast, it encourages us to apply it on the robot.

3.4 FF solver

In this section we describe and evaluate the algorithmic techniques used in the FF planning solver [8]. We have tested other solvers [18], but we realise that FF is the most efficient with a high difference respect the others. One of the reasons why was the most successful automatic planner at the AIPS-2000 planning competition.

The FF solver relies on a forward state search, using a heuristic that estimates goal distances by a relaxation of the planning task, and introduces a novel search strategy that combines hill-climbing with systematic search. His search strategy introduces new pruning techniques that have been motivated by observing classical examples. Figure 3.4 shows the basic system architecture.

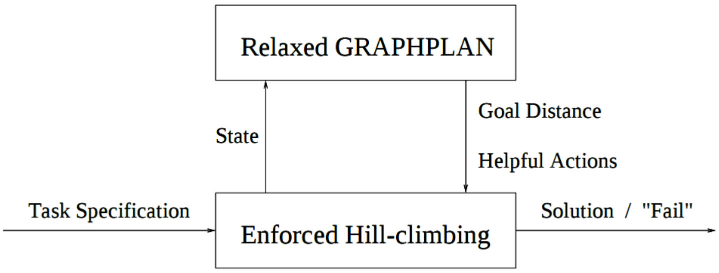


Figure 6: FF's base system architecture

The fundamental heuristic technique is the relaxed GRAPHPLAN that informs to the search engine with a goal distance estimation, and with a set of promising successors also called helpful actions. On every state the enforced hill-climbing (the FF search algorithm) either outputs a solution plan or reports that it has failed. But as enforced hill-climbing doesn't works when the graph contains dead ends, a complete best-first algorithm is invoked. Finally as enforced hill-climbing sometimes wastes a lot of time achieving goals that need to be cared for later, two techniques consisting on adding goal deletion and order the goals are integrated.

3.4.1 Heuristic estimator

In this subsection we introduce the base heuristic method used in FF. But before is necessary to describe a set of basic concepts.

In previous sections we described a planning task P as triple $\langle \boldsymbol{O}, \boldsymbol{s_0}, \boldsymbol{g} \rangle$ where O is the set of actions, S_0 is the initial state and g is the goal formed by a set of atoms, facts or conditions.

Definition 3.1. Given a planning task $P = \langle \boldsymbol{O}, \boldsymbol{s_0}, \boldsymbol{g} \rangle$. The relaxation P' of P is defined as $P' = \langle \boldsymbol{O'}, \boldsymbol{s_0}, \boldsymbol{g} \rangle$, with

$$O' = \{\langle \text{pre(o), add(o)}, \emptyset \rangle \quad | \quad \langle \text{pre(o), add(o), del(0)} \rangle \ \in O\}$$

In other words the relaxed planning task is obtained by ignoring the delete lists of all the actions.

Definition 3.2. An action sequence $\langle \boldsymbol{o_1}, .., \boldsymbol{o_n} \rangle$ is called a relaxed plan of P if and only if solves the relaxation P' of P.

The relaxed plan simplifies the planning task and it could be solved in polynomial time. The base heuristic method in FF is derived by applying GRAPHPLAN to relaxed planning tasks.

In the GRAPHPLAN algorithm, the planning graph is a directed, layered graph that contains two kind of nodes: fact nodes and action nodes. Each time step i contains the layer of facts or conditions that can possibly made true in i steps, and the layer of actions that are possibly applicable given those conditions. The GRAPHPLAN extends the planning layer until a fact layer that contains all goals. After, starting from the goal layer a recursive backward search algorithm is invoked until the initial layer. In the backward search, at each time step $i > 0$ the algorithm initialises the set of selected actions at layer $i - 1$ as empty, and for each condition consider all the achieving actions at layer $i - 1$ selecting the first one that not exclude any action previously selected. If it's not possible then goes back to the $i + 1$ time step and chooses a different action.

Definition 3.3. Given a pair of actions (o, o') we define them as mutually exclusive if one action deletes a precondition or an add effect of the other.

In the same way we can define two facts (f, f') as mutually exclusive at time step $i > 0$ if each action at level $i - 1$ that achieves f is mutually exclusive from each action at level $i - 1$ that achieves f'.

The classic GRAPHPLAN algorithm backtracks each time that all the possible action that achieve a fact are mutually exclusive with some of the previous selected actions. But in a relaxed plan, as we don't have delete effects, the algorithm will never backtrack. In this case if the task is solvable, the planning graph is extended until a fact containing all the goals; and starting from the top layer going down to the initial layer a relaxed plan is collected. Moreover all this process only takes polynomial time.

The heuristic of FF is defined as:

$$h(S) = \sum_{i=0,\dots,m-1} |O_i| \tag{2}$$

Where each O_i is the set of actions selected in parallel at time step i such tat $\langle O_0, ..., O_{m-1} \rangle$ from a relaxed solution. m is the first fact layer containing all goals.

This heuristic don't assume facts to be achieved independently as other algorithms, and takes in account possible interactions that can occur.

name		⟨pre	add,	del⟩
opG_1	=	$(\{P\},$	$\{G_1\}$	$\emptyset)$
opG_2	=	$(\{P\},$	$\{G_2\}$	$\emptyset)$
opP	=	$(\emptyset,$	$\{P\}$	$\emptyset)$

Considering the short example 3.4.1 and starting by the initial state $\{\emptyset\}$, the goal $\langle\{G_1\},\{G_2\}\rangle$ is achieved in fact layer two by selecting the actions opG_1 and opG_2 from action layer one. At layer one is achieved the fact $\{P\}$ after selecting the action opP at layer 0. Summarising the resulting plan is $\boldsymbol{\langle\{opP\},\{opG_1,opG_2\}\rangle}$ which has distance estimation of three. Other planners consider independent facts, and this implies that the distance estimation is four since we consider two unities for achieving fact G_1 and two more unities for achieving G_2.

As the returned solution is not always optimal, cause the optimal solutions can not be synthesised efficiently, the FF applies some techniques to make GRAPHPLAN return solutions as short as possible. As we are interested in obtain optimal solutions, but in some cases the FF is not optimal, we combine this planner with the FD solver that returns optimal solutions (see section 5.4). We got in touch with Jörg Hoffmann (the main developer of FF) that told us that FF is much faster than other solvers because it doesn't have to look for the optimal solution. Although in most of the experiments always return the best solution.

The first technique applied to reduce the length of the solution is consider whenever is possible dummy functions called NOOPs that propagates facts f from one layer to another. This functions has no other effect than f and no other precondition than f. The GRAPHPLAN search tries to select the NOOPs first, before other real actions that achieve f. If no NOOP is available we set the "easiest" real action that satisfies the fact. The difficulty of an action is defined by:

$$\text{difficulty(o)} := \sum_{p\in\text{pre(o)}} \min\{i \quad | \quad p \text{ is member of the fact layer at time step } i\} \tag{3}$$

The second technique lies on linearize whenever is possible the actions. If some action just add a precondition from another action we can consider both of them as a single one. This reduces the number of layers and possible actions for the relaxed plan. There are few situations in which linearization is applied since linearize a problem is NP-complete.

3.4.2 Enforced Hill-climbing

In this section we introduce FF base search algorithm. Is a variation of the hill-climbing method because combine local and systematic search.

In heuristic forward search the search space is the space of all reachable states with their heuristic evaluation. From the current state S the algorithm uses breadth first search to obtain the nearest state S' with better heuristics i.e $h(S') < h(S)$.

If in one iteration breadth first search for a better sate fails, the algorithm restart the search and tries to solve the task by a complete heuristic search algorithm. This could happen whenever there exists some dead end in the graph, even being solvable. As is PSPACE-hard decide whether a given planning task has a dead end or not, the algorithm just starts to apply enforced hill-climbing until the goal is reached or the algorithm fails.

Finally two heuristic techniques are introduced to prune the search space:

1. Helpful actions selects a set of promising successors to a search state. This imply to restrict the action choice in any state by just select those actions that are relevant. For example consider a problem of thousand of objects when the objective is to move two of them. This technique will only consider the actions related to those objects, throwing away the irrelevant actions. Or it will consider only those actions that are applicable and add at least one goal to the next time step.

2. Added goal deletion cuts out branches where some goal has apparently been achieved too early. In such a case that from the current state S the algorithm founds a relaxed solution plan that contains an action o that deletes one of the goals, we remove this state S from the search space, and we don't generate any successor. Also in a preprocessing phase the planner decides heuristically an ordering for all the goals. These are then fed to enforced hill-climbing in an incremental manner. And if during the search a state reaches a "future goal" the state is from the search space.

4 RRT

In this section we briefly introduce the concept of Rapidly-exploring Random Tree [12] as an algorithm designed to efficiently search in non-convex, high-dimensional spaces by randomly building a space-filling tree. This method is specially designed to handle non-holonomic constraints or even kinodynamics constrains, and a high degrees of freedom in an efficient manner.

The aim of this algorithm is to generate a continuous path in a given space, from an initial state to a goal state, avoiding all the obstacles. Usually this space is the configuration space of the joints of the robot, but for example on a kinodynamic planning problem the space is the tangent of the configuration space.

Listing 5: RRT Algorithm

```
GENERATE_RRT($x_{init}$, $K$, $\Delta t$)
    $T.init(x_{init})$;
    for k=1 to $K$ do
        $x_{rand} \longleftarrow$ RANDOM_STATE();
        $x_{near} \longleftarrow$ NEAREST_NEIGHBOR($x_{rand}$, $T$);
        $u \longleftarrow$ SELECT_INPUT($x_{rand}$, $x_{near}$);
        $x_{new} \longleftarrow$ NEW_STATE($x_{near}$, $u$, $\Delta t$);
        $T.add\_vertex(x_{new})$;
        $T.add\_edge(x_{near}, x_{new}, u)$;
Return $T$
```

The algorithm generates random samples from the search space, and tries to connect them with the nearest state in the tree. If the connection is feasible a new state is added to tree. The tree grows until there exists a connection between some of the sates the start and the goal.

Some of the advantages of this method is that:

- Tends to explore unexplored portions of the state space.
- Is probabilistically complete under very general conditions.
- Is simple which facilitates performance analysis.

Also there are a lot of variations of this algorithm that include nice properties to the search. For example RRT growth can be biased to increase the probability of sampling an specific area. The growth factor (i.e max distance at the random states are generated) also affects the areas explored during the search.

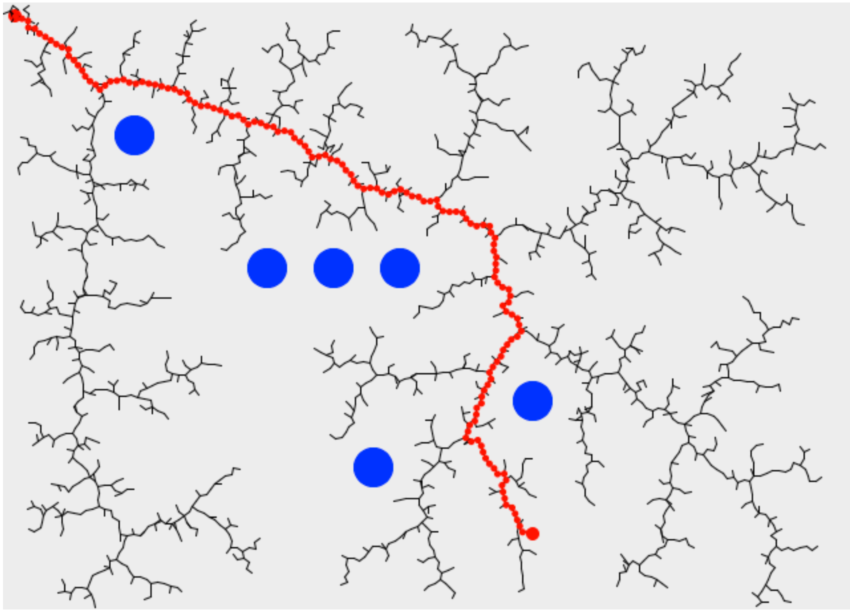


Figure 7: RRT tree expansion

5 Sequential Quadratic Programming

This section synthesises the basic concepts of sequential quadratic programming, that is an optimisation method used when computing the IK of the robot. Sequential Quadratic Programming (SQP)[16][7][9][6] is one of the most successful methods for the numerical solution of constrained nonlinear optimization problems. Is very powerful since it can handle any degree of non-linearity including non-linearity in the constraints. But main disadvantage of this method is that it incorporates several derivatives, which likely need to be worked analytically.

5.1 Basic definitions

The SQP is used to solve NLP problems such as:

$$\begin{aligned} &\min f(x) \\ &\text{over } x \in \mathbb{R}^n \\ \text{s. t. } h(x) &= \quad 0 \\ g(x) &\leq \quad 0 \end{aligned} \tag{4}$$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the objective function, and the functions $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $g : \mathbb{R}^n \rightarrow \mathbb{R}^p$ describe the equality and inequality constraints.

In a general case this problem is not a Quadratic Programming problem (QP), since the objective function needs to be quadratic and the constrains linear. But this optimization method iteratively models the NLP into a QP sub-problem, and solves it to construct a new iterate QP. Finally the solution converges to a local minimum x^*.

To construct it uses the Langrangian multipliers and the KKT conditions:

Definition 5.1. The Lagrangian functional $L : \mathbb{R}^{n \times m \times p} \rightarrow \mathbb{R}$ associated with the NLP is defined as:

$$L(x, \lambda, \mu) := f(x) + \lambda h(x) + \mu g(x) \tag{5}$$

where $\lambda \in \mathbb{R}^m$, $\mu \in \mathbb{R}^p$ are defined as the Lagrangian multipliers.

The $\nabla f(x)$ denote the gradient of the function f in x, and the second derivative of the function (or Hessian matrix) is denoted as $Hf(x)$.

Definition 5.2. If $x*\in \mathbb{R}^n$ is a local minimum of the NLP problem then there exists Lagrange multipliers $\lambda^* \in \mathbb{R}^n$, $\mu^* \in \mathbb{R}^p_+$ such that:

$$\nabla L(x^*, \lambda^*, \mu^*) = \nabla f(x^*) + \nabla h(x^*)\lambda^* + \nabla g(x^*)\mu^* = 0 \tag{6}$$

With $\mu^* g_i(x^*) = 0 \; \forall i = 1, ..., p$.

Then we say that (x^*, λ^*, μ^*) satisfy the first order necessary optimality or KKT conditions.

Definition 5.3. If $x^* \in \mathbb{R}^n$ is a local minimum of the NLP and:

$$\begin{aligned} g_i(x^*)\mu_i = &\quad 0 \\ \mu_i^* > &\quad 0 \qquad \text{in the active constraints i.e,} \quad \{i \mid g_i(x^*) = 0\} \end{aligned}$$

This condition is called strict complementary slackness at $x*$.

And the matrix $G(x)$ is defined as:

$$G(x) := (\nabla h_1(x), ..., \nabla h_m(x), \nabla g_{i_1}(x), ..., g_{i_q}(x) \tag{7}$$

where $i_1, ..., i_q$ are the coefficients of the active constraints.

Definition 5.4. Then we say that it satisfies the second order sufficient optimality conditions if:

1. The columns of $G(x^*)$ are linearly independent.
2. Strict complementary slackness holds at $x*$.
3. The Hessian of the Lagrangian with respect to x is positive definite on the null space of $G(x^*)^T$.

If (x^*, λ^*, μ^*) satisfy the first and second order sufficient optimality conditions, then x^* is a local minimum, and the Lagrangian multipliers are uniquely defined.

5.2 Construction of the problem

Since we have to transform the problem from Equation 5.1 into a QP problem, the more intuitive idea is to apply Taylor expansion to linearize the objective function and the constraints. If we apply Taylor until second order to the objective function and until first order for the constraints we obtain:

$$\begin{aligned} &\min \; f(x^k) + \nabla f(x^k)(x - x^k) + \frac{1}{2}(x - x^k)^T Hf(x^k)(x - x^k) \\ &\text{over } x \in \mathbb{R}^n \\ &\text{s.t. } h(x^k) + \nabla h(x^k)^T \cdot ((x - x^k) = 0 \\ &\qquad g(x^k) + \nabla g(x^k)^T \cdot ((x - x^k) \leq 0 \end{aligned} \tag{8}$$

But this kind of approximation doesn't works, as there are plenty of examples showing that if we choose a value close to the local minimum, the problem doesn't converge.

However the first order and second order conditions imply that x^* is a minimum of the problem.

$$\begin{aligned} &\min\ L(x, \lambda^*, \mu^*) \\ &\text{over } x \in \mathbb{R}^n \\ &\text{s.t. } h(x) = 0 \\ &\quad g(x) \leq 0 \end{aligned} \tag{9}$$

Where λ^*, μ^* are the Lagrangian multipliers associated to the local minimum x^*.

The Lagrangian functor can be expanded using Taylor and we obtain the problem:

$$\begin{aligned} \min \quad & \nabla L(x^k, \lambda^k, \mu^k)^T d(x) + \frac{1}{2} d(x)^T HL(x^k, \lambda^k, \mu^k) d(x) \\ \text{over} \quad & d(x) = (x - x^k) \in \mathbb{R}^n \\ \text{s.t.} \quad & h(x^k) + \nabla h(x^k)^T d(x) = 0 \\ & g(x^k) + \nabla g(x^k)^T d(x) \leq 0 \end{aligned} \tag{10}$$

There are two lemmas that demonstrate that if the problem has not inequality constraints or $\mu^k = 0$ for all the inactive constraints, the Equation 5.2, Equation 5.2 are equivalent. The second condition is satisfied by the KKT conditions explained in the previous section. This problem is quadratic and thus must be solved with non-linear methods, but this sub-problem with one variable is much easier to tackle than the parent problem.

Finally if our initial value (x^0, λ^0) is close enough to the minimum, the problem from Equation 5.2 can be solved iteratively by the Newton method since it converges quadratically to the solution (x^*, λ^*). In this case the inequality constraints disappear, cause if the initial value is close enough ot the minimum, the active constraints are the same.

The solution is obtained iterating over k:

$$x^{k+1} = x^k + d_x \tag{11}$$

$$\lambda^{k+1} = \lambda^k + d_\lambda \tag{12}$$

where (d_x, d_λ) represents the step or direction that push us to the minimum. This values are the solution and the the associated multiplier of the problem from Equation 5.2.

Although there is a drawback applying this method, and is that we have to calculate the Hessian of the Lagrangian which could be hard to compute. In order to solve that we approximate the Hessian by the follow equation.

$$\nabla L((x^{k+1}, \lambda^{k+1}) - \nabla L((x^{k}, \lambda^{k+1}) \approx HL((x^{k+1}, \lambda^{k+1})(x^{k+1} - x^{k}) \tag{13}$$

5.3 IK as an optimization problem

Once that we have already exposed how the the method works and how is constructed, is the moment to explain how the IK is transformed as an optimization problem.

In our case what we want to minimise is distance and orientation from the pose generated by a given configuration to the desired pose. So the objective function is to apply forward kinematics over each configuration and calculate the error in distance and orientation from the desired pose.

But also we want to avoid that the final configuration is in collision or that this solution is outside the joint limits. This new conditions add new constraints to the problem and at the same time generates new local minimum.

Since we are just interested in the global minimum, we need a hill climbing method that jumps to the local minimum to find the global one. We apply a relaxation method [19] of the solution in which the constraints corresponding to the collisions or the joint limits are added in the objective function, with variables weights. This strategy turn the infeasible constraints into penalties driving the constraints violations to zero.

Each inequality constraint $g_i(x) \leq 0$ is added in the objective function as $|gi_i(x)|^+$, where $|x|^+ = \max(x, 0)$. While the equality constraints $h_j(x) = 0$ are added as $|h_j(x)|$. In all the cases the penalties are multiplied by some coefficient α_j that is increased at each step (for example by a factor of 10), to drive constraint violations to zero. Since all these functions are not regular, we can not apply the SQP method.

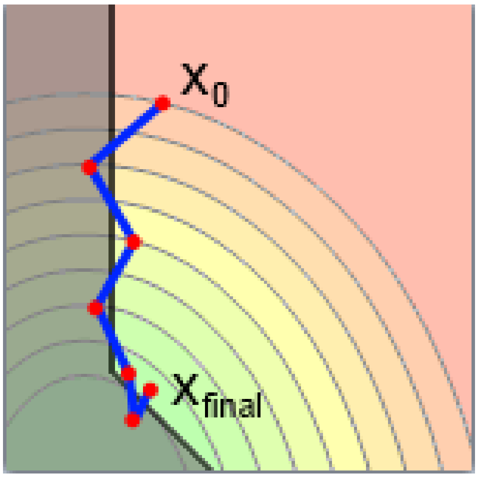


Figure 8: Relaxation of the constraints

To solve this issue we add new slack variables t to the objective function, and add new constraints:

$$0 \leq t$$
$$g_i(x) \leq t$$

5.3.1 No-collisions constraint

In this subsection we detail how the collision constraints are defined [19].

Definition 5.5. We consider the distance between two sets A, B as:

$$\text{dist}(A, B) = \inf\{\|T\| | (T + A) \cap B \neq \emptyset\} \tag{14}$$

where T express the smallest translation that puts the shapes in contact.

Definition 5.6. And the penetration distance between two sets A, B is defined as:

$$\text{penetration}(A, B) = \inf\{\|T\| | (T + A) \cap B = \emptyset\} \tag{15}$$

Observe that the first one is zero if the shapes are in contact, and the second one is zero when the shapes are not in contact.

From those distances we define the signed distance:

$$\text{sd}(A, B) = \text{dist}(A, B) - \text{penetration}(A, B) \tag{16}$$

That is positive when the objects are not in contact, and negative otherwise.

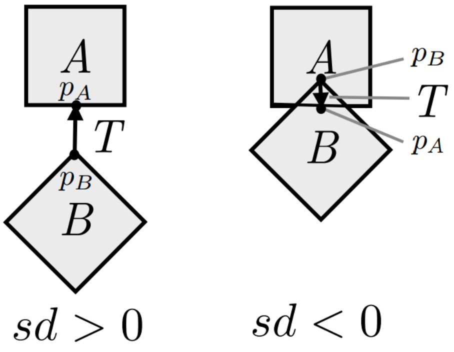


Figure 9: Signed distance

So in order to avoid collision we have to ensure that:

$$\mathrm{sd}(A_i, O_j) \geq d_{safe} \qquad \forall i \in \{1, .., N_{links}\}, \, \forall i \in \{1, .., N_{obstacles}\} \tag{17}$$

Since the previous constraints are not linear if we consider the l_1 distance we have to linearize to add in to the QP problem. This linearization is done with respect the closest points of both objects (p_A, p_B), the direction of the minimum translation $\hat{n}$ and the poses of the objects in the world ($F_A^w(x)$, ($F_B^w(x)$).

$$\mathrm{sd}_{AB}(x) \approx \hat{n} \cdot (F_A^w(x) p_A - F_B^w(x) p_w) \tag{18}$$

6 Interface Implementation

This section describes the implementation of the whole project step by step. First, we will explain the implementation of the basic motions for the robot in C++. The second subsection explains how this actions are modelled in PDDL, according to the geometric information required by the actions in C++. The third section defines the creation of a simulator to visualise the experiments. The fourth section explains the development of the interface that stores the current geometric information of the problem, translates it to PDDL, and executes the actions of the solution. The last section explains the inference layer, used to update the information of the task planning layer once the motion planner can not execute the action. Furthermore in each section, we describe all the successive improvements developed to improve robustness and efficiency.

6.1 Creating a library of local actions in C++

In this subsection we will explain the computation of the basic motions for the robot. This section is a prerequisite of the project, because we need a set of basic motions for the robot to test our experiments. Since there are infinite planning problems as simple as bring me an object and as complex as build a mock-up, the list of actions is infinite. We can define the actions as basic motions such as move to a position, or as a combination of motions such as iron. The idea behind the project is to have a huge list of actions for the robot and try to solve real-life tasks autonomously. For our experiments we have developed the actions pick, move arm, place and move the base.

6.1.1 Pick

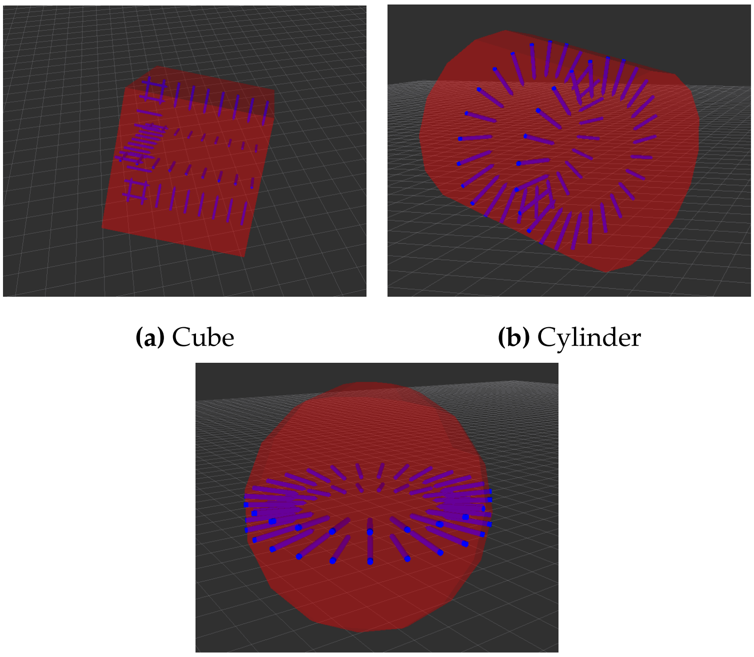
The action pick allows the robot to take an object from the environment with his hand or gripper. His implementation is divided in three steps:

1. Generate grasping candidates
2. Compute the Inverse Kinematics of the robot using the grasping candidates
3. Plan a free collision trajectory from the current configuration to the desired one.

The grasping candidates are the poses poses in which the hand/gripper is almost in contact with the object, pointing forward to the object. The resultant poses will depend on the shape and the size of the object. We have calculated them for a cylinder, a rectangle and a sphere. In each case we translate the frame of the object (in the centre of it) to a new frame in the border by parametrise the shape, and rotating the new frame with the x-axis pointing inside. In the rectangle the frame is translated to each one of the faces (excepting the down face) by half of the width, length or height. For each face a different rotation is implemented. We use cylindrical coordinates (see Equation 19) for the cylinder case and spherical coordinates (see Equation 20) for the sphere case. For the cylinder case is just necessary one rotation in z of $\alpha + \pi$ radians, where α is the angle from Equation 19. While for the sphere case we have to rotate in x and z. To avoid calculations, we use the symmetry of the sphere to generate the hemisphere negative candidates.

$$\phi(\alpha, t) = (r \cdot \cos(\alpha), r \cdot \sin(\alpha), h \cdot t) \qquad \text{where, r (radius/2) and h (height/2) are constants.} \tag{19}$$

$$\psi(\theta, \phi) = (r \sin(\theta) \cos(\phi), r \sin(\theta) \sin(\phi), r \cos(\theta)) \qquad \text{where, r (radius/2) is constant.} \tag{20}$$



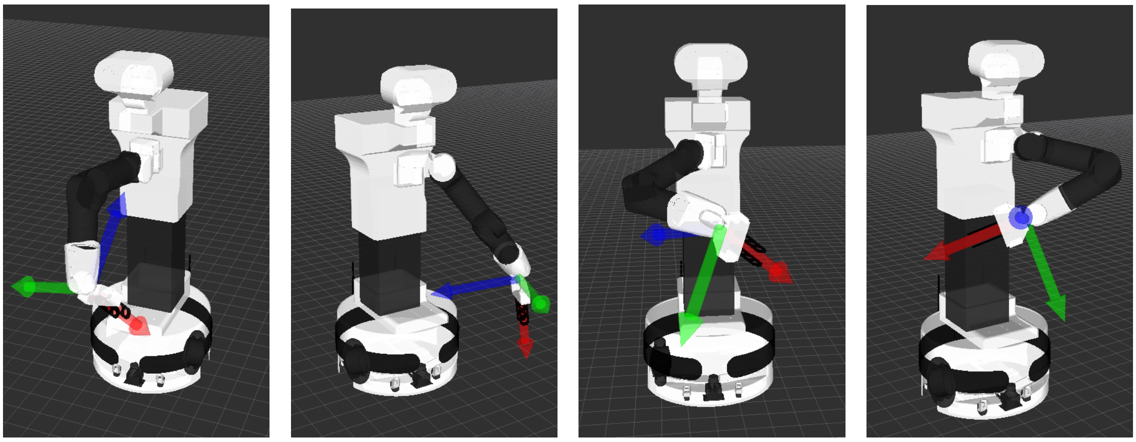
(a) Cube

(b) Cylinder

(c) Sphere

Figure 10: Grasp candidates generation

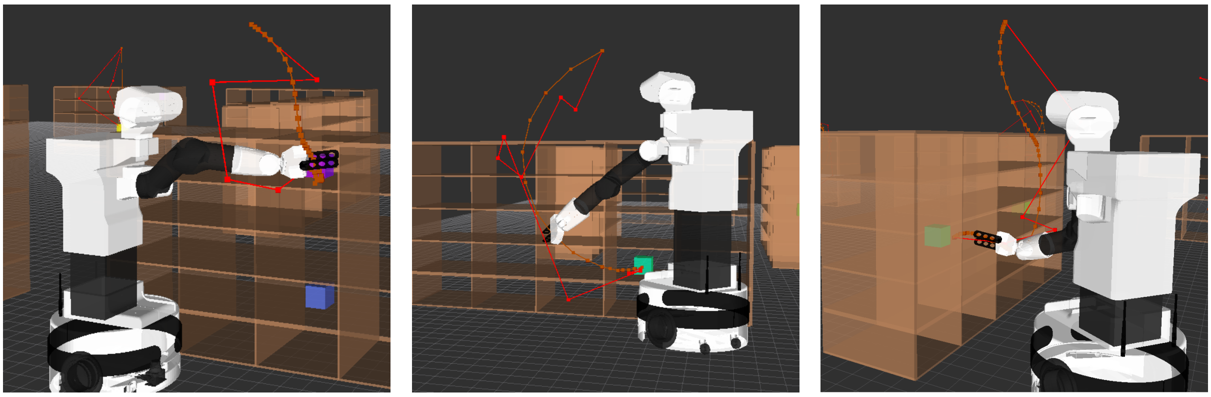
The second step is to calculate the inverse Kinematics of the robot, to obtain a valid joint configuration for the robot. The problem is solved as an optimisation problem where all the joint limits are defined as constraints. At the beginning the IK was only calculated for the upper body, meaning that the robot had to be placed in a convenient position to pick the object. After some improvements we included the collision detection in the IK, to detect when an object blocks another. In the last version the IK also returns a solution for the base. This allows us to grasp objects from hard situations such as a shelf. In both improvements the IK still an optimisation problem in which the obstacles are added as inequality constraints. Adding obstacles generate local minimum that are not valid solutions, but we solve it by applying the method described in the SQP section. The floating planar base is added to the model of the robot as two prismatic joints and a rotational joint, without joint limits.



(a) Pose 1 **(b)** Pose 2 **(c)** Pose 3 **(d)** Pose 4

Figure 11: Inverse Kinematics for TIAGo

The last step is to find a valid joint trajectory in the C-space avoiding collisions from the current configuration to the solution of the IK. We use the RRT implemented in the OMPL that generates a set of random nodes free from collision in the C-space, and link them creating a path from the start to the goal. We use this method because is one of the fastest ones, and is probabilistic complete. The global planners methods used to plan a path for the base offer a poor result with so many degrees of freedom. The only drawback is that it gives long trajectories waisting a lot of energy. To solve this problem we optimise the given trajectory in the joint space minimising the energy.



(a) Case 1 **(b)** Case 2 **(c)** Case 3

Figure 12: RRT Trajectory Generation: Red trajectory obtained from the random tree node generation. Orange trajectory is optimised minimising the energy from the red trajectory

In simulation the trajectory from the RRT is interpolated using a Catmul-Rom interpolator [21] to execute a smooth and continuous trajectory, cause in the solution the points are not equidistant.

To avoid check collisions in the IK and the RRT between the gripper tips and the grasping object we add the grasping object to a blacklist.

We have to extract the best solution for all the grasping candidates. If the IK and the RRT found a solution for some candidate the solution is executed, otherwise the best solution is computed. In case that the IK returns collision for all the candidates, we consider the best solution as the one with less "new objects in contact" that doesn't collide with the object we want to grasp or with the furniture. The "new objects in contact" are those objects that are not blocking another objects at the current moment. The choose of the best solution determines the direction in which the robot pushes away objects when grasping an object hampered by others. Applying the last condition we enforce the robot to keep pushing away objects from the same side.

Finally once it executes the solution the arm always return to a fixed configuration with the object on his hand. This configuration tries to expose the minimum possible the arm to avoid collisions when moving the base. Although this configuration is not always the same, cause sometimes the final configuration is in collision and the RRT could not reach it. In this case it takes another configuration from a set of configurations.

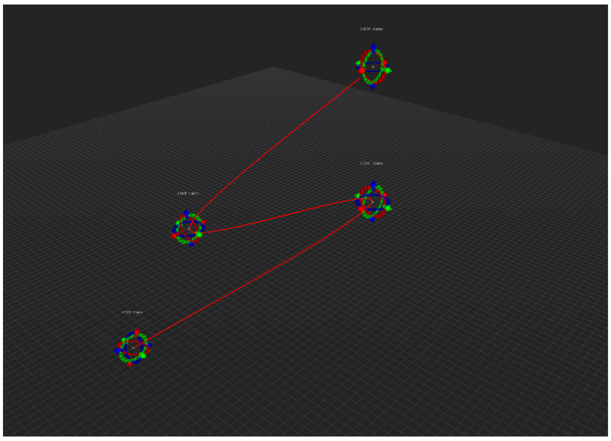


Figure 13: CatmulRom trajectory used to interpolate the points of the RRT trajectory

If the IK returns a solution for the base, we have to calculate a plan for the base and move it to the desired position. The action pick becomes a combination of the actions pick and move base. The algorithm first tries to calculate if there exists a solution with the IK without base, and if no solution calls the IK with base. Once the base moves to the desired position, it calls again the IK without base due to small errors in the real base pose produced by the discretization of the grid map. It could seem that the improvement of the IK with base is inefficient cause we to calculate several times the inverse kinematics of the robot. But before this implementation, we calculated random positions for the base until reach a valid one, which was much more slower. Moreover allows to grasp objects from inside of a shelf which is a really hard situation.

6.1.2 Place

The action of placing an object is quite similar to the one of picking an object. The main difference is that we don't have to calculate the grasping candidates, or search for the best solution. We first compute the IK to the placing pose, and next we calculate the trajectory without collisions. Finally we end at a fixed up position. At the end we reset the blacklist to detect possible future collisions with the object we have placed.

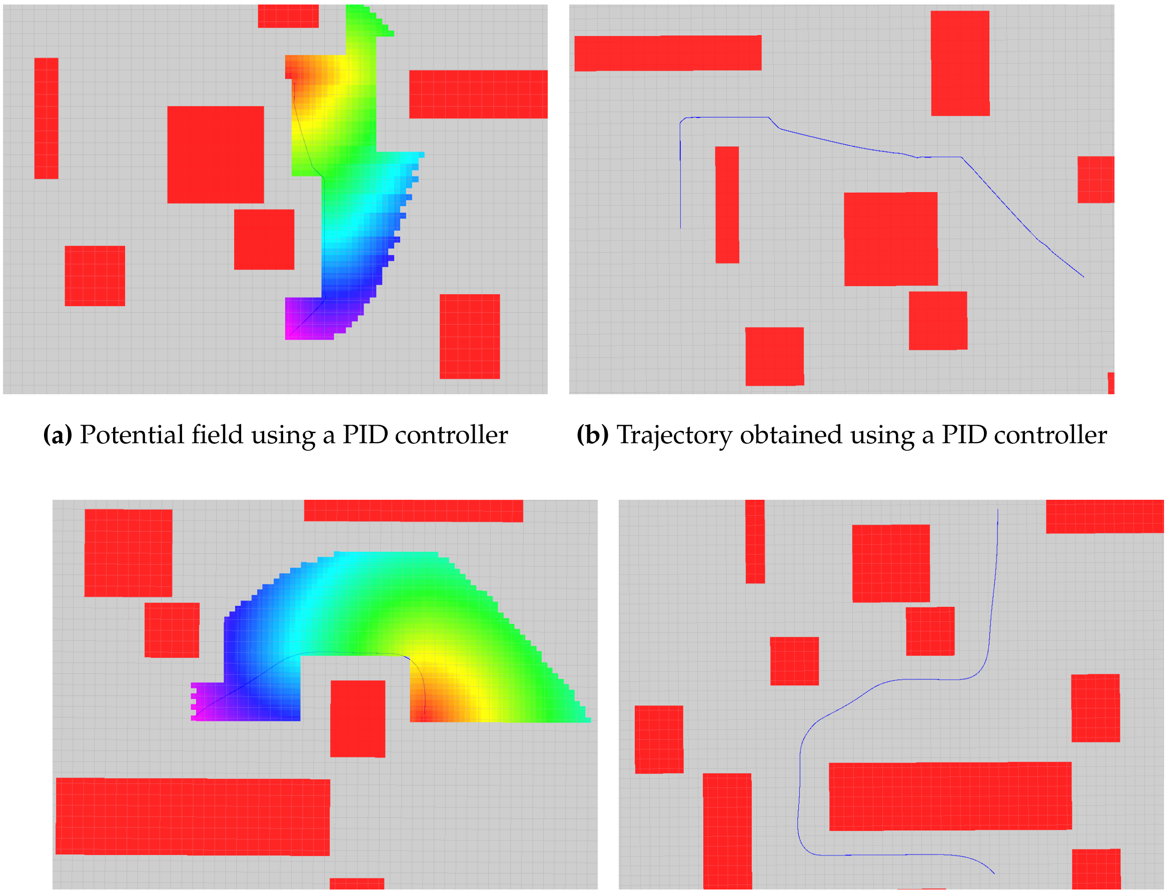
6.1.3 Move Arm

The action to move the arm to a new position is quite similar to the one of pick or place. In this case the desired pose is respect the base link, so we have transform it to the global frame by composing it with the transformation of the base. The resultant pose is passed to the IK without base, and the RRT calculates a valid trajectory without collisions.

6.1.4 Move Base

The action move base changes the current position and orientation of the robot in the world, calculating a path free from obstacles from the current position to the desired one. The action has been implemented for the TIAGo robot and for the REEM-C robot (humanoid), so both algorithms will be described. Moreover we will explain and compare the two different algorithms used to create a plan for the TIAGo robot.

The first approach implemented on TIAGo, uses the navigation function of the navfn package provided by ROS [11]. This method receives a grid as a map, where the user determines the size, the discretization, and the cells representing obstacles. The planner creates a potential field from the start point (high potential value) to the goal (lowest potential value), associating a value to each cell. The graph search algorithm Dijkstraa or A*, depending on the choice of the user, calculates the optimal path. Two different kind of controllers has been implemented to follow the resultant trajectory and output the velocity of the wheels [10]. The first one is a PID controller that interpolates from point to point of the trajectory [5], and the second one is a Look Ahead controller [4] that controls the curvature. In both cases the linear velocity is maintained, and what we do is control the direction in which we are driving. Comparing both controllers, the Look ahead offers smoother trajectories and is easy to tune, but it goes far from the trajectory when there are sharp curves. The PID trajectory is closer from the trajectory of the plan but sometimes appears sharp edges. As both controllers only control position; a PID has been implemented to orientate first with trajectory and at the goal with the desired orientation. This orientation PID minimise the angle error, by extracting the minimum angle distance.



(a) Potential field using a PID controller

(b) Trajectory obtained using a PID controller

(c) Potential field using a Look ahead controller

(d) Trajectory obtained using a Look ahead controller

Figure 14: Navfn implementation

The second method implemented on TIAGo uses the package SBPL [13] [14] [22] from ROS. This method uses a set of primitives which represent short, kinematically feasible motions which form the basis of movements that can be performed by the robot. The planner combines this motion primitives to generate a path from the start position to the goal position. The problem could be seen as a graph where the three is expanded by linking the primitives. The final result is a trajectory in $\mathbb{R}^2 \times SO(2)$ (i.e (x, y, θ)) free from collisions.

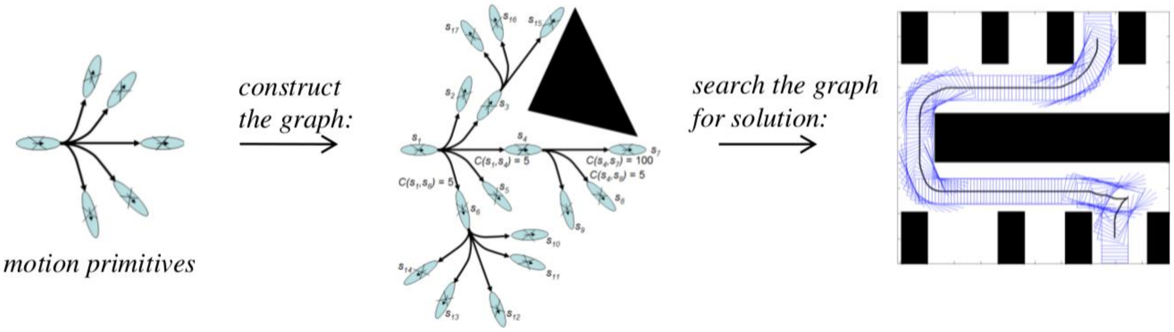


Figure 15: SBPL

To generate the motion primitives as feasible trajectories of the robot, we calculate them by optimisation, giving the start and goal position and adding the model of the robot and his constraints as optimisation constraints. For each orientation we generate a new set of primitives, cause we won't finish at the same position if we are moving forward at 0° or at 180°. The package allows the user to choose a circle discretization, and each primitive should start and end at some of the discretized angles. More primitives generate better and smooth trajectories but at the same time spend more time searching for a plan. Furthermore we give a specific weight to each primitive, giving higher priority to move forward or backward than turning. Different weight will generate different kind of paths.

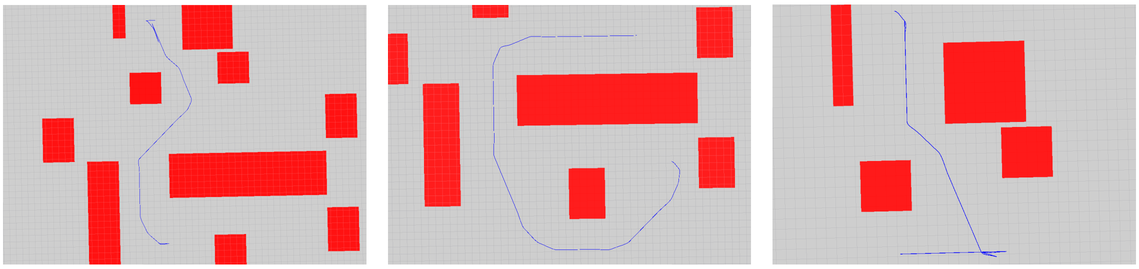
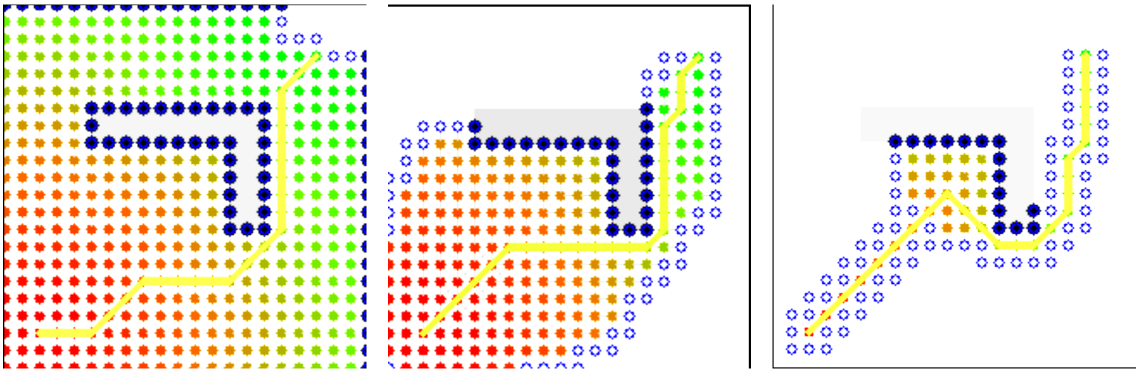


Figure 16: SBPL resultant trajectories

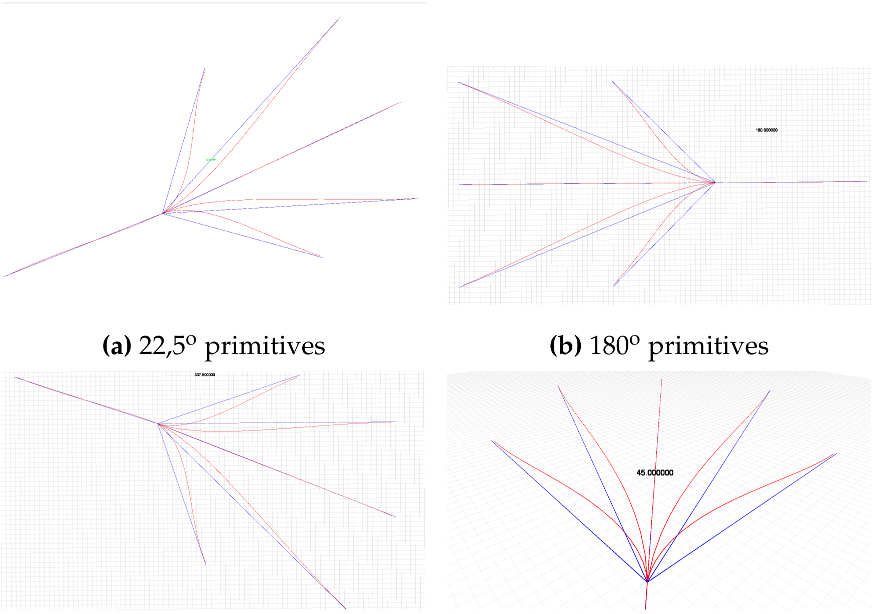
The planner uses ARA*, anytime D* or incremental A* as graph search algorithm to find the optimal path. All those methods are derived from A*, and the main advantage they offer is that they keep the information from previous searches, and in every iterative search they only need to explore the new states. Moreover in all those planners there is a parameter that relaxes the condition on the graph search finding good-enough solution in less time and using less memory. This parameter of "good enough" is often denoted as epsilon of the best solution, and return a solution that is no worse than epsilon times the cost of the optimal solution. Figure shows the state expansion of Dijkstraa, A* and weighted A* (i.e A* with epsilon > 0).



(a) Dijkstraa graph search **(b)** A* graph search **(c)** Weighted A*

Figure 17: Planner comparison

As the output trajectory includes the orientation, is not necessary to develop any kind of controller. We add the previous orientation PID controller once the whole trajectory is executed, due to the error of the desired orientation and the discretized one. That planner was the one used by the Stanford Team, when they won in the 2007 the DARPA Urban Challenge.



(a) 22,5^{o} primitives **(b)** 180^{o} primitives

(c) 337,5^{o} primitives **(d)** 45^{o} primitives

Figure 18: Generation of the primitives

Comparing both methods, SBPL offers better results in terms of trajectory and control. But the Navfn method is more efficient in terms of time at the first plans. This is a generalisation because both methods have parameters that can be tuned, and the resultant trajectories or the efficiency of both depend on those parameters.

To implement it on the REM-C robot, we have used the footstep_planner package from ROS. The package is an extension of SBPL, since it uses it to calculate valid trajectories for the humanoid. In this case the motion primitives are real steps for the robot, and the planner combine them to extract a list of steps. Each step has a position, orientation and a leg associated. A zero moment point trajectory is calculated to follow the trajectory from step to step, and from the inverse dynamics of the robot we extract the trajectory of the joints of the legs. As the torso has to follow the orientation of the legs, a quaternion interpolation is used to interpolate between the initial and final orientations. The resultant trajectory allows to robot to walk over some obstacles as for example stones.

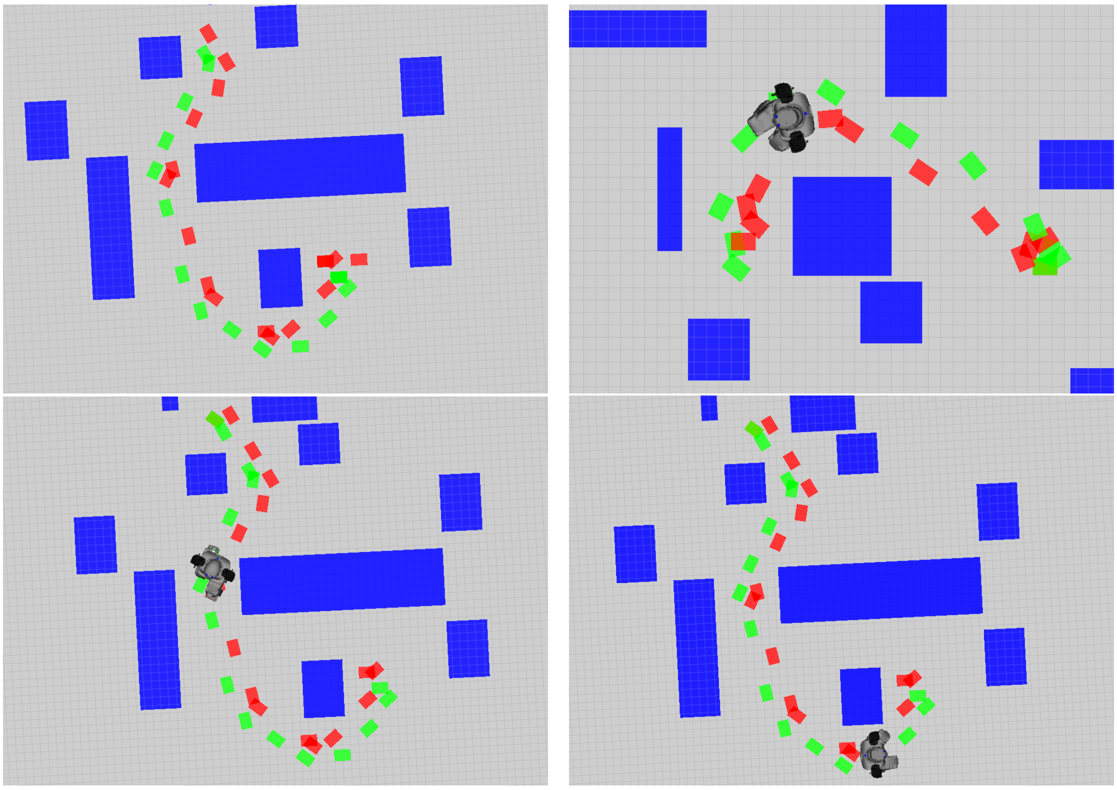


Figure 19: Footstep planner implementation for the humanoid

In all the implementations the grid map is represented as a matrix or a vector. To project all the objects from the environment onto the grid map, we project the objects onto the XY plane and we check which cells contain an obstacle. To avoid collisions with the base or the torso we apply a dilation to the grid map by convolute a mask of a desired size.

6.2 Modelling the actions in the PDDL language

In the last section we have created a "library" of basic motions for the robot. To combine task planning and motion planning we need to translate the geometric information to the task planner that checks the feasibility of the motions and creates a plan. This step is done by the PDDL that models the motions as geometric prerequisites and effects, in order that the task planner check the feasibility. Is necessary to model the actions according to the problem we are gonna solve. As our purpose is to create an interface capable to solve the maximum variety of planning problems we have model it including all the possible conditions. This section explains in detail how the actions have been modelled in PDDL.

First we start describing the predicates and the type of objects, which define the kind of problems we are gonna solve. Since we have computed the actions move base, move arm, pick and place we are capable to solve planning problems that consider moving objects from place. Solve other kind problems as for example drive a car, do the laundry, etc. requires extra actions as press the accelerator and/or requires a special modelling as for example *is_dirt? cloth*. Our problems has to be as real and general as possible so we have to include blocking situations. If we want to solve problems in which objects are hide within a drawer we require to create the motion open/close a drawer and model it on PDDL.

Types of objects:

- *object* - Are the objects from the environment.
- *arm* - Is the arm of the robot. Considering two arms allows us to grasp and place objects with both arms, although TIAGo has a single arm.
- *base_pos* - Are the base poses necessary to move the base
- *arm_pos* - Are arm poses necessary to move the arm, grasp or place an object. The positions of the grasping objects are represented as arm poses.

List of predicates:

- *(base_path ?x ?y - base_pos)* - True when exists a path between two base positions.
- *(arm_at ?x - arm_pos)* - Stores the current arm position of the arm. True when the arm is at the given position. As we have a single arm this predicate has only one parameter, otherwise we have to consider the arm and the arm_pos.
- *(base_at ?x - base_pos)* - Stores the current base position of the robot. True when the base is at the given position.

- *(free ?x - arm)* - True when the gripper do not have any object.

- *(attached_to ?x - object ?y)* - True when one object lies above another object. If cube_1 is on table_1 *attached_to cube_1 table_1* is true.

- *(base_close_to ?x - object ?y - base_pos)* - True when the given base position is close to the given object. Is used to move the robot close to the table when we want to grasp an object from the table.

- *(obj_at ?x - object ?y - arm_pos)* - Stores the position of the object. True when the given object is at the given position. Two objects could be at the same arm position but lying on different objects, i.e *(attached_to obj_1 ref)* must be different from *(attached_to obj_2 ref)*.

- *(free_pos ?x - object ?y - arm_pos)* - True when the given position is free and doesn't contain an object. A given position could be free for example for table_1 but not free for table_2, i.e *(free_pos table_1 pos)* must be different from *(free_pos table_2 pos)*.

- *(graspable ?x - object)* - True when the object is graspable. Not all the objects are graspable, as for example the tables.

- *(block ?x ?y - object ?z ?w - arm_pos)* - True when object one blocks object two in the given positions for object one and two. This predicate will be never set to false in the effects of some action. If we want to escape from a blocking situation, we have to move object two from his current position, cause the predicate is associated to the positions of the objects.

Now let's proceed to model the actions from the last section using the previous object types and predicates.

6.2.1 Pick

We start explaining the action pick, that is coded in PDDL as follows:

Listing 6: Pick action

```
  (:action pick
    :parameters (?grasp_obj ?ref_obj - object ?tool - arm
    ?grasp_pos ?curr_arm_pos - arm_pos ?curr_base_pos - base_pos)

    :precondition (and (graspable ?grasp_obj) (arm_at ?curr_arm_pos)
    (base_at ?curr_base_pos) (free ?tool)
    (attached_to ?grasp_obj ?ref_obj)
    (base_close_to ?ref_obj ?curr_base_pos)
    (obj_at ?grasp_obj ?grasp_pos)
    (forall (?obj - object ?pos - arm_pos)
    (imply (and (attached_to ?obj ?ref_obj) (obj_at ?obj ?pos))
    (not (block ?obj ?grasp_obj ?pos ?grasp_pos)))))

    :effect  (and (attached_to ?grasp_obj ?tool)
    (not (arm_at ?curr_arm_pos)) (arm_at up_pos)
    (not (obj_at ?grasp_obj ?grasp_pos)) (not (free ?tool))
    (not (attached_to ?grasp_obj ?ref_obj)))
)
```

The action receives as parameters:

- 2 objects
- 1 arm
- 2 arm poses
- 1 base pose

The solver generates all the possible combinations with the objects of the problem, checks which combinations are feasible, and selects the best solution. Using the preconditions we check that one of the objects correspond to the object where is located the grasping object *(attached_to ?grasp_obj ?ref_obj)*. The arm poses correspond to the position of the object *(obj_at ?grasp_obj ?grasp_pos)* and the current arm position *(arm_at ?curr_arm_pos)*. The base pose correspond to the current base position *(base_at ?curr_base_pos)*. Furthermore, to pick an object is mandatory that it satisfies the geometric conditions of the motion:

- The object must be graspable. We can not pick a table or a shelf $\Longrightarrow$ *(graspable ?grasp_obj)*
- The gripper could not carry any object. $\Longrightarrow$ *(free ?tool)*
- The robot must be close to the reference object where is located. $\Longrightarrow$ *(base_close_to ?ref_obj ?curr_base_pos)* + *(base_at ?curr_base_pos)*
- No other object above the table could block the object we want to grasp. $\Longrightarrow$ (forall (?obj - object ?pos - arm_pos) (imply (and (attached_to ?obj ?ref_obj) (obj_at ?obj ?pos)) (not (block ?obj ?grasp_obj ?pos ?grasp_pos))))

We use the tools imply and for of the PDDL language to check it for every object above the table in his current positions.

The solver selects the action pick, but also the grasping objects, if it satisfies the conditions and is involved in the solution. The rest of parameters for this action are fixed once we choose the grasping object.

In a first implementation we reset the blocking predicate in the effects and we didn't consider the current positions of the objects. But the solver consumed a lot of time. The solution is to maintain the blocking situations, but just check it in the current positions of the objects. We considered to add the current base position to the blocking predicate, because once we move to another position the locks are not the same. But when adding this parameter the solver enters in a loop cause the optimal solution is to move the base and pick the object.

The effects after applying the action pick are:

- The object is attached to the gripper. $\Longrightarrow$ *(not (attached_to ?grasp_obj ?ref_obj))* + *(attached_to ?grasp_obj ?tool)*
- The gripper has an object. $\Longrightarrow$ *(not (free ?tool))*
- The arm ends up at a fixed up position. $\Longrightarrow$ *(not (arm_at ?curr_arm_pos))* + *(arm_at up_pos)*

Using the IK with base, the predicate base_close_to is not more necessary. But we have to include the base_path predicate between the current pose and the grasping pose as a precondition; and update the base position in the effects. The final base pose is considered as a PDDL constant since is created in the motion planner.

6.2.2 Place

We proceed explaining the action place modelled in PDDL as follows:

Listing 7: Place action

```
  (:action place
    :parameters (?grasp_obj ?ref_obj - object ?tool - arm
    ?place_pos ?curr_arm_pos - arm_pos ?curr_base_pos - base_pos)

    :precondition (and (arm_at ?curr_arm_pos) (base_at ?curr_base_pos)
    (not (free ?tool)) (attached_to ?grasp_obj ?tool)
    (base_close_to ?ref_obj ?curr_base_pos)
    (free_pos ?ref_obj ?place_pos)
    (forall (?obj - object ?pos - arm_pos)
    (imply (and (attached_to ?obj ?ref_obj) (obj_at ?obj ?pos))
    (and (not (block ?grasp_obj ?obj ?place_pos ?pos))
    (not (block ?obj ?grasp_obj ?pos ?place_pos))))))

    :effect  (and (free ?tool) (obj_at ?grasp_obj ?place_pos)
    (not (arm_at ?curr_arm_pos)) (arm_at up_pos)
    (attached_to ?grasp_obj ?ref_obj)
    (not (free_pos ?ref_obj ?place_pos))
    (not (attached_to ?grasp_obj ?tool)))
)
```

This action is modelled quite similar to the pick action. Some of his preconditions and effects are think to work with harmony with the previous action.

The action receives the following parameters:

- 2 objects
- 1 arm
- 2 arm poses
- 1 base pose

In this case the preconditions check that one of the objects is the object we are grasping *(attached_to ?grasp_obj ?tool)*. One of the arm poses is the current arm pose *(arm_at ?curr_arm_pos)*. The base pose is the current base pose *(base_at ?curr_base_pos)*. The rest of parameters are chosen by the solver and correspond to the reference object where we place our object, the arm position corresponding to the place position and the arm.

The geometric conditions necessary to apply the motion are:

- The gripper has an object. ⟹ *(not (free ?tool))*
- The robot must be close to the reference object where we want to place the object. ⟹ *(base_close_to ?ref_obj ?curr_base_pos)* + *(base_at ?curr_base_pos)*
- The placing position doesn't contain any object ⟹ *(free_pos ?ref_obj ?place_pos)*
- We can not place an object if there is another one blocking it, or if the placing object blocks another one in the given placing position. ⟹ (forall (?obj - object ?pos - arm_pos) (imply (and (attached_to ?obj ?ref_obj) (obj_at ?obj ?pos)) (and (not (block ?grasp_obj ?obj ?place_pos ?pos)) (not (block ?obj ?grasp_obj ?pos ?place_pos)))))

As in the pick action we didn't set as free the position previously occupied by the object, once an object occupies a position is not free anymore. So the place position must be always a new position. Otherwise we realised that when several objects are blocking another, the solver moves one of them to a new position, but uses the position of the first to place the second one creating a new blocking situation. Although is necessary to generate a lot of poses to solve the problem, and is necessary to create two times the same pose with different names if we want to interchange objects from position.

Check that in the given place pose the object won't block any other is redundant cause the place position has to be new. The inverse condition checks that there is enough space to place the object. Moving the objects blocking the place position let the place pose "free" to place the object.

The effects after applying the action are:

- The object is attached to the reference object. ⟹ *(not (attached_to ?grasp_obj ?tool))* + *(attached_to ?grasp_obj ?ref_obj)*
- The arm end up at a fixed up position. ⟹ *(not (arm_at ?curr_arm_pos))* + *(arm_at up_pos)*
- The object is located at the place position. ⟹ *(obj_at ?grasp_obj ?place_pos)*

- The place position is not free anymore ⟹ *(not (free_pos ?ref_obj ?place_pos))*

- The arm dosn't carry any object ⟹ *(free ?tool)*

Using the IK with base we don't need the base_close precondition, but at the same time we have to update the position for the base. Although when using the IK with base the task planner doesn't have any geometric information of the final base position. This causes that for the task planner is the same to place an object close to his position than far from it. To solve this problem an option is to implement two different kind place actions, one with the fixed IK and another one with the base IK with a higher cost. But this increases the computational time and FF doesn't allow cost in the functions.

6.2.3 Move Base

The action move base is computed as follows in PDDL:

Listing 8: Move Base action

```
 (:action  move_base
   :parameters  (?curr_pos  ?next_pos  -  base_pos)

   :precondition  (and  (base_path  ?curr_pos  ?next_pos)
   (base_at  ?curr_pos))

   :effect   (and  (base_at  ?next_pos)  (not  (base_at  ?curr_pos)))
)
```

In the preconditions we check that:

- The parameter curr_pos is the current base position. ⟹ *(base_at ?curr_pos)*

- There exists a path between the current pose and the desired pose. ⟹ (base_path ?curr_pos ?next_pos)

In the effects we update the position of the base. ⟹ *(base_at ?next_pos)* + *(not (base_at ?curr_pos))*

6.2.4 Move Arm

The action move arm is computed as follows in PDDL:

Listing 9: Move Arm action

```
(:action move_arm
    :parameters (?curr_pos ?next_pos - arm_pos)
    :precondition (and (arm_at ?curr_pos))
    :effect  (and (arm_at ?next_pos) (not (arm_at ?curr_pos)))
)
```

We check that the parameter curr_pos is the current pose of the arm *(arm_at ?curr_pos)*. And we update the arm position once executed *(arm_at ?next_pos* + *(not (arm_at ?curr_pos))*.

6.3 Simulating an environment with collision objects

To visualise our experiments we have created a Kinematic simulator using the markers of RVIZ. Commonly the simulations are run with Gazebo but since we ignore the physics and perception of the robot we have created our own simulator.

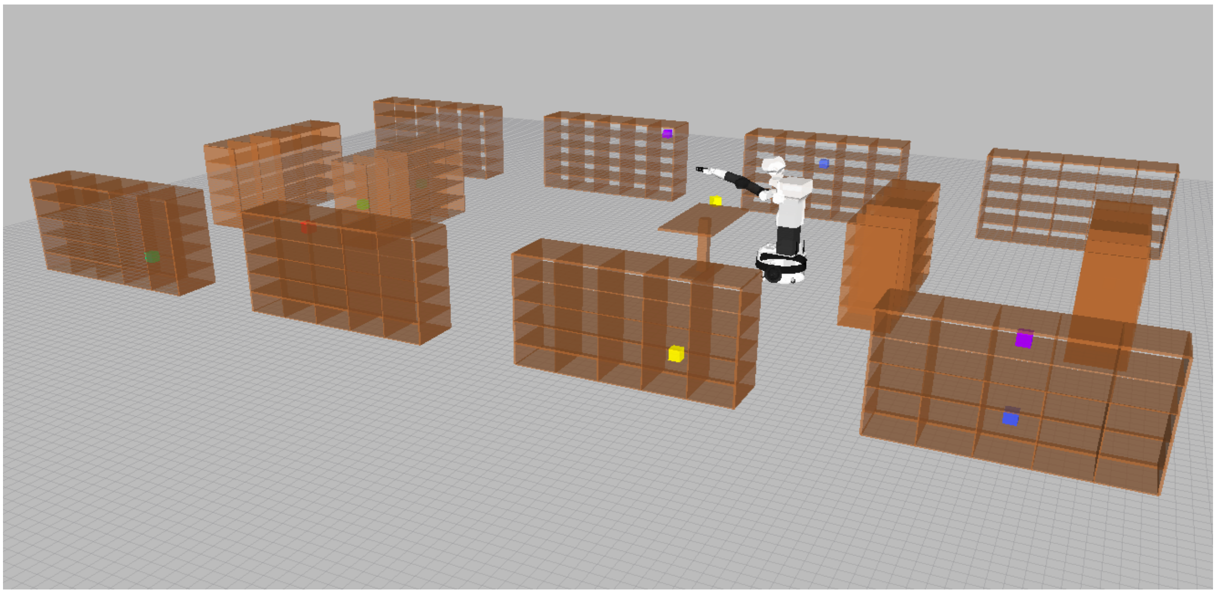


Figure 20: Environment with collision objects

The first step is to create the objects and associate it to collision objects. We have created two different classes in C++ one for the robot objects and one for the environment objects, that inherit from a base class, and store the basic properties of the objects and create the collision objects. To create the collision objects we use Bullet [1] or FCL [17][2] that are libraries in C++ that compute collisions between objects. When computing the IK or RRT the program calls to this collision objects to check if there exists solution. Also we have created a method that visualises the collision objects as markers in RVIZ.

Since we want to create many planning problems to test our framework, we create a parser in XML that reads the objects and his properties from a file, and create them as C++ objects. The objects are defined as it follows in the XML:

Listing 10: XML object description

```
    <object>
        <name name="cylinder_on_table-1"/>
        <color r="1.0" g="0.1882" b="0.1882"/>
        <size radius="0.05" height="0.2"/>
        <position x="0" y="0" z="0.1"/>
        <orientation x="0.0" y="0.0" z="0.0" w="1.0"/>
        <reference ref="table-1"/>
        <shape shape="cylinder"/>
        <graspable graspable ="true"/>
    </object>
```

Although there is no perception in this step, all the properties described and stored in the C++ object could be extracted using a laser or a camera. To make real environments we have created synthetic objects as a shelf or a table combining several collision objects.

6.3.1 Updating the environment

As our purpose is to interact with the objects of the environment we need to update the position of those objects. To simplify it we have created a tree structure containing all the objects and his positions, ordered by the reference objects. If the cube is on the table the reference of cube is table, and the reference of table is world since it is on the world. Structuring in this way has many advantages. First if we reference the object to the gripper of the robot, it automatically updates his position when we move the gripper. Second allows to compute the positions in the local frame. Is much more intuitive to set the goal at 10cm at the left of the center of the table than to define it in the global frame. Otherwise the IK and

RVIZ use global coordinates. As we have defined our structure obtain the global coordinate is as simple as compose the homogeneous transformations from the local coordinates.

In all our problems the first step is be to parse the XML of the environment creating the collision objects, and create a tree with all the objects to visualise the environment.

6.4 Creating a C++ interface that relates task planning with motion planning

This section is the most important of the project since it explains how the task planning layer communicates with the motion planning layer and vice versa to find a solution.

The task planner layer generates a plan from the current state using the computed actions. This step is done by the PDDL and the solver. While the motion planning executes the actions in simulation. In this way of communication the task planner creates a plan and the motion planner execute it step by step. But even with the proper predicates usually doesn't exists a solution or the motion plan could not execute it. Maybe because in the PDDL there are not enough positions, or because the current state in the PDDL doesn't show all the information from the environment. This usually happens in the real problems in which a priori we don't know if an object is blocking another. Is necessary that the motion planner communicates with the task planner and we call it inference.

But first is necessary to crate a structure in C++ that contains the geometric information of the environment, that updates it when necessary, and parse it to a PDDL in order to call the solver. In a first attempt (the Hanoi Problem) we created a class containing all the attributes with a method set and get for all of them. It worked for this problem, but since we have to generate a new class for every different problem we discard it. The second approach was to create one class for the PDDL objects, one for the PDDL predicates and other for the PDDL problem. It worked but it was complex to understand and only allowed to generate predicates with one or two parameters. Finally we create a single class problem that contains all the objects and predicates. His main functionality is to update the problem (changing some predicates or adding/deleting objects) and print the true predicates in a PDDL problem file format.

We also have implemented a ROS service that receives the generated problem file and the parsed domain file, and returns the solution. This service creates two temporary files and calls the executable of the solver. The solution is parsed in a string and send it back to the main program, that parse it and tries to execute the actions by order. Since FD is slower in comparison with FF specially when doesn't exists solution, but FF is not optimal, in some experiments we call the service using FF until a solution is found when we call again the service using FD to obtain the optimal solution. This step produces a higher cost in most of the cases but we ensure that the obtained solution is optimal.

To generate the different experiments we only modify the environments and the goal. The goal is in fact a PDDL problem object that remains constant.

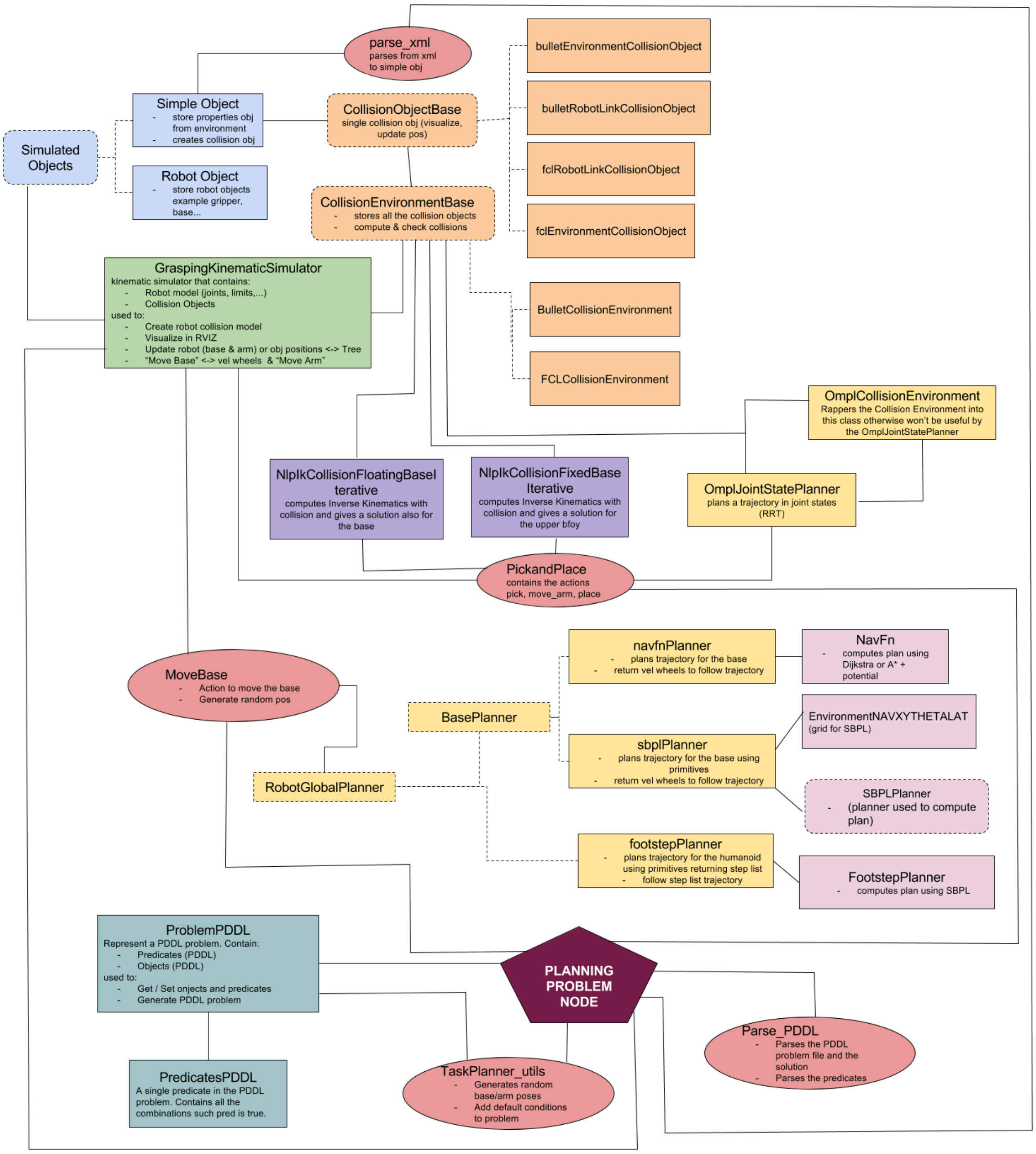

parse_xml
parses from xml
to simple obj
bulletEnvironmentCollisionObject
Simple Object
- store properties obj from environment
- creates collision obj
CollisionObjectBase
single collision obj (visualize, update pos)
bulletRobotLinkCollisionObject
Simulated Objects
Robot Object
- store robot objects example gripper, base...
fclRobotLinkCollisionObject
CollisionEnvironmentBase
- stores all the collision objects
- compute & check collisions
fclEnvironmentCollisionObject
GraspingKinematicSimulator
kinematic simulator that contains:
- Robot model (joints, limits,...)
- Collision Objects
used to:
- Create robot collision model
- Visualize in RVIZ
- Update robot (base & arm) or obj positions <-> Tree
- "Move Base" <-> vel wheels & "Move Arm"
BulletCollisionEnvironment
FCLCollisionEnvironment
OmplCollisionEnvironment
Rappers the Collision Environment into this class otherwise won't be useful by the OmplJointStatePlanner
NlpIkCollisionFloatingBaseIterative
computes Inverse Kinematics with collision and gives a solution also for the base
NlpIkCollisionFixedBase Iterative
computes Inverse Kinematics with collision and gives a solution for the upper bfoy
OmplJointStatePlanner
plans a trajectory in joint states (RRT)
PickandPlace
contains the actions pick, move_arm, place
navfnPlanner
- plans trajectory for the base
- return vel wheels to follow trajectory
NavFn
- computes plan using Dijkstra or A* + potential
MoveBase
- Action to move the base
- Generate random pos
BasePlanner
EnvironmentNAVXYTHETALAT
(grid for SBPL)
sbplPlanner
- plans trajectory for the base using primitives
- return vel wheels to follow trajectory
RobotGlobalPlanner
SBPLPlanner
- (planner used to compute plan)
footstepPlanner
- plans trajectory for the humanoid using primitives returning step list
- follow step list trajectory
FootstepPlanner
- computes plan using SBPL
ProblemPDDL
Represent a PDDL problem. Contain:
- Predicates (PDDL)
- Objects (PDDL)
used to:
- Get / Set onjects and predicates
- Generate PDDL problem
PLANNING PROBLEM NODE
Parse_PDDL
- Parses the PDDL problem file and the solution
- Parses the predicates
PredicatesPDDL
A single predicate in the PDDL problem. Contains all the combinations such pred is true.
TaskPlanner_utils
- Generates random base/arm poses
- Add default conditions to problem

Figure 21: Chart class in C++

6.4.1 Inference

This subsection explains how the motion planner updates the geometric information when it could not execute the action, or the task planner doesn't found a solution.

If the solver doesn't found a solution is because the problem is unsolvable, or because there are not enough positions. For example if we want to move a set of objects to an empty table is necessary to create as many poses as objects we want to move and a pose for the base close to the empty table. As our objective is that the robot a priori just have the basic information, the goal, and it has to discover all by itself; by default this poses are not generated. To solve this issue, we generate random base poses and random arm poses for each object referenced in the world. The base poses are calculated close to the objects, and the arm poses correspond to place positions in the upper surface of the object. If we have two tables with three cylinders at each execution without solution we generate one random base pose for each table and one placing pose fir each table. All the random poses are generated using Gaussians centred in the object. Moreover, the base poses must be in a free cell of the grid map, while the arm positions must be inside the limits of the object and free from collision. To generate poses free from collision we generate a random cube with a fixed size in the given pose and check that there is no collision. Generate poses until the problem found a solution spends a lot of time in some cases during the task planning and generate unnecessary objects, but is a price we have to pay if we want the robot to work autonomously.

The second problem appears when some motion fails because there exists a mismatch between the real state and the current state of the PDDL. In such case we have to update some precondition. By default all the base_path predicates between positions are set to true, and the unknown predicates as the block predicates are set to false.

Table 1 summarises which predicate we update if the action pick fails

If the RRT doesn't found solution we could not reach the object and we consider that the problem is not solvable. As the IK always consider a valid position without collision for all the joints if the RRT fails is because the solution is very complex and we didn't give enough time, which never happens. Usually fails when it tries to move the arm back to the up position, but we have generated a set of different up positions to fix it.

Table 1: Pick action fails

Step where if fails	Predicate of the PDDL to update
Couldn't compute IK	(base_close_to) = false for the current base pose and we generate a new base pose
IK collides with the reference object	(base_close_to) = false for the current base pose and we generate a new base pose
IK collides with an object (not the reference)	(block block_obj grasp_obj ...) = true and we generate a new place pose for each blocking object
No solution for the RRT	In this case the problem could not be solved

There is a special case in which an object blocks another one and is blocked by the other. This happens sometimes when there are a lot of objects above the table, because the robot doesn't grasp from the same side due to the grasping candidates. If such a case appears, setting the block condition the problem doesn't found solution, cause to move object one needs to move object two but to move object two needs to move object one. In this situation we move the base setting base_close_to as false.

Table 2 summarises which predicate we update if the action place fails:

Table 2: Place action fails

Step where if fails	Predicate of the PDDL to update
Couldn't compute IK and count < 10	Search for a new place position, and count ++
Couldn't compute IK and count >= 10	(base_close_to) = false for the current base pose and we generate a new base pose
IK collides with the reference object	(base_close_to) = false for the current base pose and we generate a new base pose
IK collides with an object (not the reference)	(block block_obj grasp_obj ...) = true and we generate a new place pose for each blocking object
No solution for the RRT	In this case the problem could not be solved

The counter is used to search new place positions without moving the base. The action place tries to place the object in the new positions without calling the solver, because we don't care about the exact position where it leaves the object. We can not set this counter as infinite because as the given base positions are random, maybe the current position is to far from the reference object to find IK.

If the action move arm fails is due to the RRT or because we are not giving it a valid position. In both cases there is no solution.

The action move base fails because the base planner didn't found a solution from the current pose to the desired one. We set base_path as false and generate a new base position. Sometimes we are so close to a invalid position that the planner detects the start as an invalid position. In this case we "magically" translate the robot to the attached free square in the grid.

We also generate new base positions or place positions when we update the predicates. If the base_close is set to false we have to generate a new base pose close to the obstacle to move there. And if several objects are blocking another one we need to move them and place them on new place positions.

The IK with base introduce a lot of changes to the inference. Since the predicates base_close_to and base_path doesn't exists those updates disappear. If the action move base fails we consider that there is no solution. If the action pick and place fails cause the IK didn't found any solution means that we can't grasp or place the object so the problem is unsolvable. Finally when the problem founds a double blocking we clear all the blocking conditions and call again the solver. Is like restarting the problem from the current state.

7 Simulation and generated examples

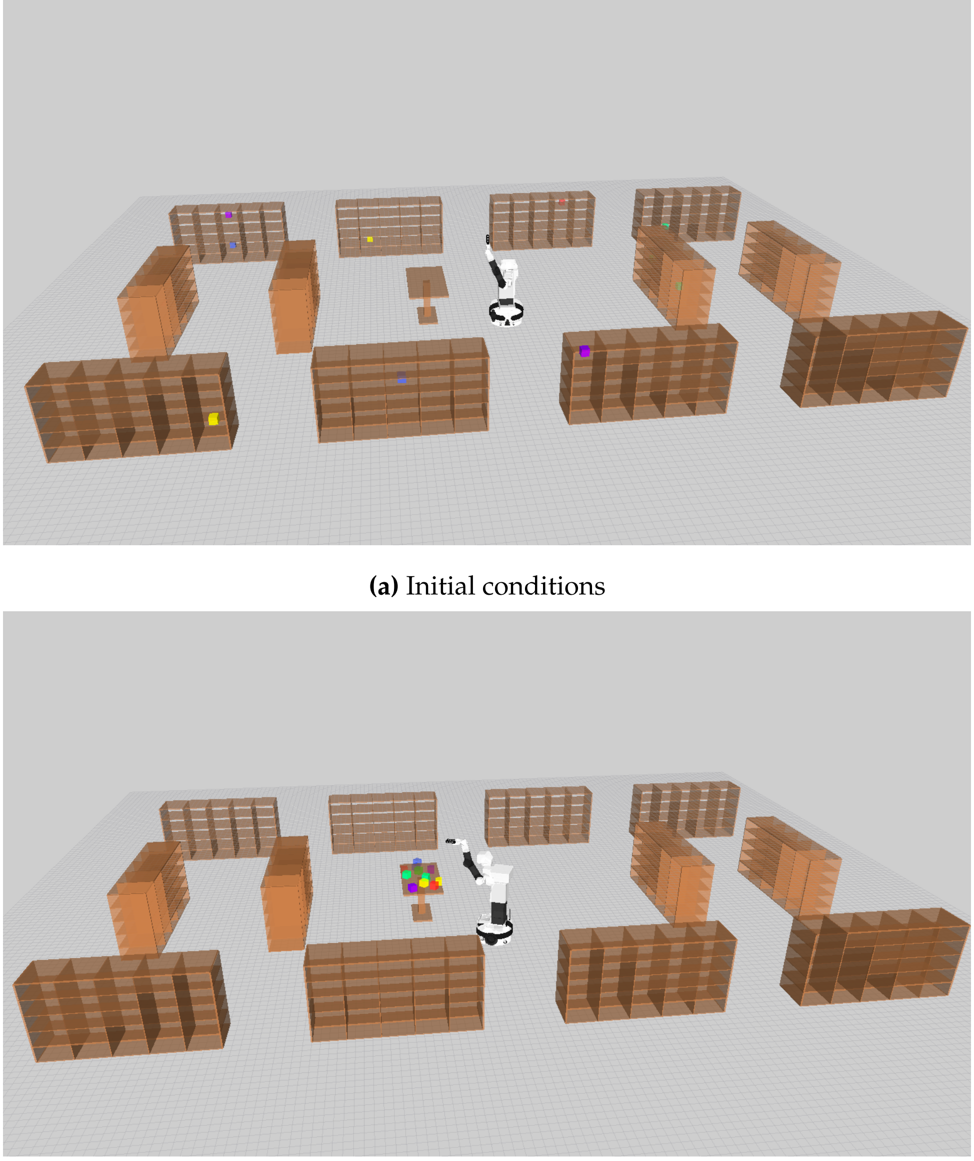
This section presents different planning problems solved with our algorithm. In all the problems the robot has to move objects from position. We used the last implementation of our algorithm that uses SBPL as a base planner and computes the IK with base.

All experiments were carried out on Intel Core i7-2.93GHz machine with 8GB of RAM. In each experiment the results exhibit the accumulated execution times when running the solver (i.e the task planner layer) and the total execution time. We also show the total time moving the base, considering the plan and the execution time with the real speed of the robot. Moreover, we consider the total execution times when executing the RRT trajectories, considering feasible joint velocities. Excepting the plan for the base, the execution times of the RRT trajectories and the base should be zero if we want to analyse the total time spend by the motion layer. But since we want to visualise it as in the real robot we have to take those times in account since they are considerably large.

7.1 Emptying a library

In the first problem the environment simulates a library with 12 shelf's and a table in the centre of the scenario. In this experiment the robot has to pick 11 cubes distributed in the different shelf's and has to place all of them in the table.

At the beginning the PDDL contain the objects of the environment, the position of the objects, the arm of the robot, and the initial position for the base and the arm. As initial conditions we consider where the objects and the robot is located, and also that the gripper.



(a) Initial conditions

(b) Goal conditions

The following code describes it in PDDL:

```
(:objects cube10_on_table-1 - object cube10_on_table-1_pos - arm_pos
cube1_on_table-1 - object cube1_on_table-1_pos - arm_pos
cube2_on_table-1 - object cube2_on_table-1_pos - arm_pos
cube3_on_table-1 - object cube3_on_table-1_pos - arm_pos
cube4_on_table-1 - object cube4_on_table-1_pos - arm_pos
cube5_on_table-1 - object cube5_on_table-1_pos - arm_pos
cube6_on_table-1 - object cube6_on_table-1_pos - arm_pos
cube7_on_table-1 - object cube7_on_table-1_pos - arm_pos
cube8_on_table-1 - object cube8_on_table-1_pos - arm_pos
cube9_on_table-1 - object cube9_on_table-1_pos - arm_pos
```

```
    cube_on_table-1 - object cube_on_table-1_pos - arm_pos
    initial_base_pos - base_pos robot - arm shelf-0 - object
    shelf-1 - object shelf-10 - object shelf-11 - object shelf-2 - object
    shelf-3 - object shelf-4 - object shelf-5 - object
    shelf-6 - object shelf-7 - object shelf-8 - object
    shelf-9 - object table-2 - object )
(:init
   (arm_at up_pos )
   (attached_to cube10_on_table-1 shelf-11 )
   (attached_to cube1_on_table-1 shelf-11 )
   (attached_to cube2_on_table-1 shelf-3 )
   (attached_to cube3_on_table-1 shelf-4 )
   (attached_to cube4_on_table-1 shelf-5 )
   (attached_to cube5_on_table-1 shelf-6 )
   (attached_to cube6_on_table-1 shelf-7 )
   (attached_to cube7_on_table-1 shelf-8 )
   (attached_to cube8_on_table-1 shelf-8 )
   (attached_to cube9_on_table-1 shelf-9 )
   (attached_to cube_on_table-1 shelf-1 )
   (base_at initial_base_pos )
   (free robot )
   (graspable cube10_on_table-1 )
   (graspable cube1_on_table-1 )
   (graspable cube2_on_table-1 )
   (graspable cube3_on_table-1 )
   (graspable cube4_on_table-1 )
   (graspable cube5_on_table-1 )
   (graspable cube6_on_table-1 )
   (graspable cube7_on_table-1 )
   (graspable cube8_on_table-1 )
   (graspable cube9_on_table-1 )
   (graspable cube_on_table-1 )
   (obj_at cube10_on_table-1 cube10_on_table-1_pos )
   (obj_at cube1_on_table-1 cube1_on_table-1_pos )
```

```
    (obj_at cube2_on_table-1 cube2_on_table-1_pos )
    (obj_at cube3_on_table-1 cube3_on_table-1_pos )
    (obj_at cube4_on_table-1 cube4_on_table-1_pos )
    (obj_at cube5_on_table-1 cube5_on_table-1_pos )
    (obj_at cube6_on_table-1 cube6_on_table-1_pos )
    (obj_at cube7_on_table-1 cube7_on_table-1_pos )
    (obj_at cube8_on_table-1 cube8_on_table-1_pos )
    (obj_at cube9_on_table-1 cube9_on_table-1_pos )
    (obj_at cube_on_table-1 cube_on_table-1_pos )
)
```

While the goal is written as:

```
(:goal (and (attached_to cube10_on_table-1 table-2 )
            (attached_to cube1_on_table-1 table-2 )
            (attached_to cube2_on_table-1 table-2 )
            (attached_to cube3_on_table-1 table-2 )
            (attached_to cube4_on_table-1 table-2 )
            (attached_to cube5_on_table-1 table-2 )
            (attached_to cube6_on_table-1 table-2 )
            (attached_to cube7_on_table-1 table-2 )
            (attached_to cube8_on_table-1 table-2 )
            (attached_to cube9_on_table-1 table-2 )
            (attached_to cube_on_table-1 table-2 )))
 )
```

At the beginning the solver doesn't found a solution because it has to generate 11 new positions above the table to place the objects. But once it founds a solution it never replans since there is no blockage situation. Table 3 shows the time for each execution of the solver.

Each iteration without solution the problem increases his complexity generating more actions and evaluating in more states, which produces a high execution time. In the last iteration the evaluation of states is lower because the solver founds a solution. In this case we have only used the FF solver, since it returns the optimal solution and we save execution time.

Table 3: FF solver execution

Iteration of the FF	Execution time (s)	Number of eval states	Number of gen actions
Iter 1	0.14	601	2175
Iter 2	2.68	14510	3588
Iter 3	5.14	20608	5179
Iter 4	5.16	15928	6948
Iter 5	5.19	12825	8895
Iter 6	5.23	10663	11020
Iter 7	5.26	9004	13323
Iter 8	5.31	7733	15804
Iter 9	5.35	6756	18463
Iter 10	5.42	5903	21300
Iter 11	0.45	78	24315

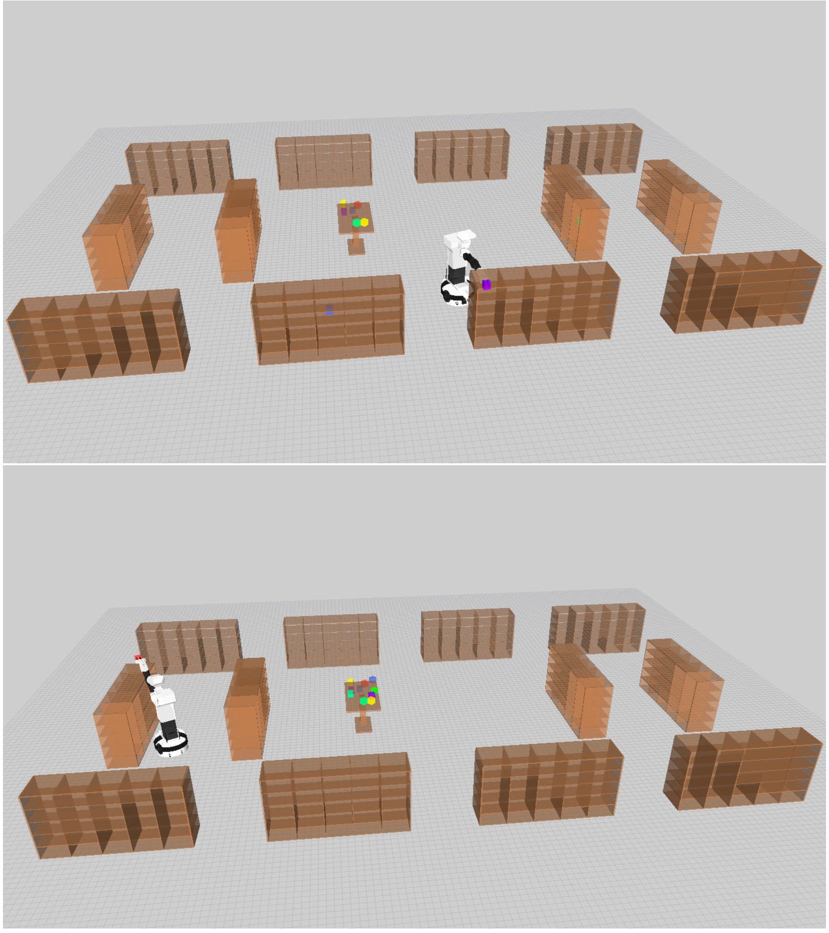


Figure 23: Screenshots of the experiment

The total execution time has been:

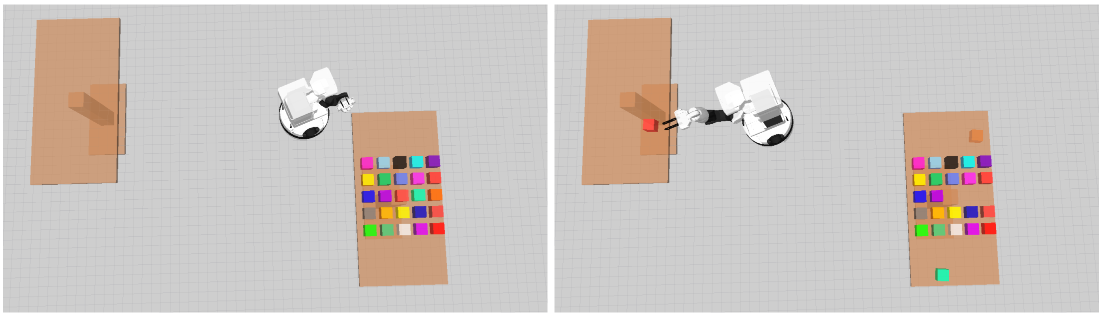
Table 4: Results experiment

Process	Execution time (seconds)
FF solver	45
RRT execution	88
Move Base	196
Total time execution	**437**

The table shows that the simulation spend most of the time in the motion planning and executing the plan. The task planning layer just spends 45 seconds searching for a solution.

7.2 Moving an object hampered by others

In this problem we have created and environment with two tables. One containing 25 cubes forming a square and another empty. The objective is to pick the cube located in the middle of the structure and place it on the other table. As the robot could grasp cubes from above we delete this option to force it to move the cubes around the object.



(a) Initial conditions

(b) Goal conditions

Figure 24: Initial and final conditions of the experiment

As in the previous section we consider the objects, his positions, the arm of the robot and the initial positions of the robot. As preconditions we don't consider any blockage since in a real situation is unknown. We just consider the gripper as free and the location of the cubes. This initial conditions are formulated in PDDL as follows:

```
 (:objects cube10_on_table-1 - object cube10_on_table-1_pos - arm_pos
 cube12_on_table-1 - object cube12_on_table-1_pos - arm_pos
 cube13_on_table-1 - object cube13_on_table-1_pos - arm_pos
 cube14_on_table-1 - object cube14_on_table-1_pos - arm_pos
 cube15_on_table-1 - object cube15_on_table-1_pos - arm_pos
 cube16_on_table-1 - object cube16_on_table-1_pos - arm_pos
 cube17_on_table-1 - object cube17_on_table-1_pos - arm_pos
 cube18_on_table-1 - object cube18_on_table-1_pos - arm_pos
 cube19_on_table-1 - object cube19_on_table-1_pos - arm_pos
 cube1_on_table-1 - object cube1_on_table-1_pos - arm_pos
 cube21_on_table-1 - object cube21_on_table-1_pos - arm_pos
 cube22_on_table-1 - object cube22_on_table-1_pos - arm_pos
 cube23_on_table-1 - object cube23_on_table-1_pos - arm_pos
 cube24_on_table-1 - object cube24_on_table-1_pos - arm_pos
 cube25_on_table-1 - object cube25_on_table-1_pos - arm_pos
 cube26_on_table-1 - object cube26_on_table-1_pos - arm_pos
 cube2_on_table-1 - object cube2_on_table-1_pos - arm_pos
 cube3_on_table-1 - object cube3_on_table-1_pos - arm_pos
 cube4_on_table-1 - object cube4_on_table-1_pos - arm_pos
 cube5_on_table-1 - object cube5_on_table-1_pos - arm_pos
 cube6_on_table-1 - object cube6_on_table-1_pos - arm_pos
 cube7_on_table-1 - object cube7_on_table-1_pos - arm_pos
 cube8_on_table-1 - object cube8_on_table-1_pos - arm_pos
 cube9_on_table-1 - object cube9_on_table-1_pos - arm_pos
 cube_on_table-1 - object cube_on_table-1_pos - arm_pos
 initial_base_pos - base_pos robot - arm table-1 - object table-2 - object )
(:init
   (arm_at up_pos )
   (attached_to cube10_on_table-1 table-1 )
   (attached_to cube12_on_table-1 table-1 )
   (attached_to cube13_on_table-1 table-1 )
   (attached_to cube14_on_table-1 table-1 )
   (attached_to cube15_on_table-1 table-1 )
   (attached_to cube16_on_table-1 table-1 )
```

```
    (attached_to cube17_on_table-1 table-1 )
    (attached_to cube18_on_table-1 table-1 )
    (attached_to cube19_on_table-1 table-1 )
    (attached_to cube1_on_table-1 table-1 )
    (attached_to cube21_on_table-1 table-1 )
    (attached_to cube22_on_table-1 table-1 )
    (attached_to cube23_on_table-1 table-1 )
    (attached_to cube24_on_table-1 table-1 )
    (attached_to cube25_on_table-1 table-1 )
    (attached_to cube26_on_table-1 table-1 )
    (attached_to cube2_on_table-1 table-1 )
    (attached_to cube3_on_table-1 table-1 )
    (attached_to cube4_on_table-1 table-1 )
    (attached_to cube5_on_table-1 table-1 )
    (attached_to cube6_on_table-1 table-1 )
    (attached_to cube7_on_table-1 table-1 )
    (attached_to cube8_on_table-1 table-1 )
    (attached_to cube9_on_table-1 table-1 )
    (attached_to cube_on_table-1 table-1 )
    (base_at initial_base_pos )
    (free robot )
    (graspable cube10_on_table-1 )
    (graspable cube12_on_table-1 )
    (graspable cube13_on_table-1 )
    (graspable cube14_on_table-1 )
    (graspable cube15_on_table-1 )
    (graspable cube16_on_table-1 )
    (graspable cube17_on_table-1 )
    (graspable cube18_on_table-1 )
    (graspable cube19_on_table-1 )
    (graspable cube1_on_table-1 )
    (graspable cube21_on_table-1 )
    (graspable cube22_on_table-1 )
    (graspable cube23_on_table-1 )
    (graspable cube24_on_table-1 )
    (graspable cube25_on_table-1 )
    (graspable cube26_on_table-1 )
    (graspable cube2_on_table-1 )
    (graspable cube3_on_table-1 )
    (graspable cube4_on_table-1 )
    (graspable cube5_on_table-1 )
    (graspable cube6_on_table-1 )
    (graspable cube7_on_table-1 )
    (graspable cube8_on_table-1 )
    (graspable cube9_on_table-1 )
    (graspable cube_on_table-1 )
    (obj_at cube10_on_table-1 cube10_on_table-1_pos )
    (obj_at cube12_on_table-1 cube12_on_table-1_pos )
    (obj_at cube13_on_table-1 cube13_on_table-1_pos )
    (obj_at cube14_on_table-1 cube14_on_table-1_pos )
    (obj_at cube15_on_table-1 cube15_on_table-1_pos )
    (obj_at cube16_on_table-1 cube16_on_table-1_pos )
    (obj_at cube17_on_table-1 cube17_on_table-1_pos )
    (obj_at cube18_on_table-1 cube18_on_table-1_pos )
    (obj_at cube19_on_table-1 cube19_on_table-1_pos )
    (obj_at cube1_on_table-1 cube1_on_table-1_pos )
    (obj_at cube21_on_table-1 cube21_on_table-1_pos )
    (obj_at cube22_on_table-1 cube22_on_table-1_pos )
    (obj_at cube23_on_table-1 cube23_on_table-1_pos )
    (obj_at cube24_on_table-1 cube24_on_table-1_pos )
    (obj_at cube25_on_table-1 cube25_on_table-1_pos )
    (obj_at cube26_on_table-1 cube26_on_table-1_pos )
    (obj_at cube2_on_table-1 cube2_on_table-1_pos )
    (obj_at cube3_on_table-1 cube3_on_table-1_pos )
    (obj_at cube4_on_table-1 cube4_on_table-1_pos )
    (obj_at cube5_on_table-1 cube5_on_table-1_pos )
    (obj_at cube6_on_table-1 cube6_on_table-1_pos )
    (obj_at cube7_on_table-1 cube7_on_table-1_pos )
    (obj_at cube8_on_table-1 cube8_on_table-1_pos )
    (obj_at cube9_on_table-1 cube9_on_table-1_pos )
    (obj_at cube_on_table-1 cube_on_table-1_pos )
)
```

The goal is that the cube_on_table-1, which is the central one, end on table-2.

```
    (:goal (and (attached_to cylinder_on_table-1 table-2 )))
```

In this experiment we call the FF solver, and if it founds solution we call the FD solver. When there is no solution the FD spends a lot of time until it stops the search. So is more efficient to guarantee the solution with the FF solver and found the optimal one with FD.

In the first iteration the solver doesn't found solution cause it needs to generate a pose on table-2. But in the second iteration it solves the problem as:

```
(pick cylinder_on_table-1 table-1 robot cylinder_on_table-1_pos up_pos initial_base_pos)
(place cylinder_on_table-1 table-2 robot random_table-2_pos2 up_pos ik_base)
; cost = 2 (unit cost)
```

Once it tries to execute the plan, it fails picking the object because the IK is in collision. The inference updates the problem adding the blocking condition and calls again the FD solver. We don't call the FF solver because there exists enough positions to solve it.

```
...
(block cylinder10_on_table-1 cylinder_on_table-1 cylinder10_on_table-1_pos
cylinder_on_table-1_pos )
(block cylinder1_on_table-1 cylinder_on_table-1 cylinder1_on_table-1_pos
cylinder_on_table-1_pos )
...
```

The FD solver outputs the following solution:

```
(pick cylinder10_on_table-1 table-1 robot cylinder10_on_table-1_pos up_pos initial_base_pos
(place cylinder10_on_table-1 table-1 robot random_table-1_pos1 up_pos ik_base)
(pick cylinder1_on_table-1 table-1 robot cylinder1_on_table-1_pos up_pos ik_base)
(place cylinder1_on_table-1 table-1 robot random_table-1_pos3 up_pos ik_base)
(pick cylinder_on_table-1 table-1 robot cylinder_on_table-1_pos up_pos ik_base)
(place cylinder_on_table-1 table-2 robot random_table-2_pos2 up_pos ik_base)
; cost = 6 (unit cost)
```

Which results the valid one.

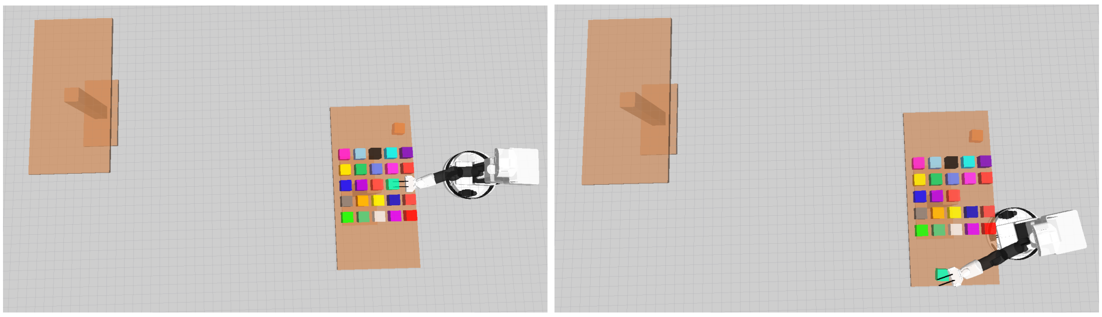


Figure 25: Screenshots of the experiment

The following table summarises the execution of the solvers:

Table 5: Solver execution

Iteration	Solver used	Execution time (s)	Number of eval. states	Number of gen actions
Iter 1	FF	0	-	-
Iter 2	FF	0.86	27	16188
Iter 3	FD	7	84	-
Iter 4	FD	14	263	-

In comparison with the FF the FD solver much more time.

The following table summarise the execution times:

Table 6: Results experiment

Process	Execution time (seconds)
Solvers	22
RRT execution	24
Move Base	23
Total time execution	**96**

As in the previous section it spends most of the time executing the motions.

7.3 Closing distances between objects

This experiment is the same as the previous one, but closing the distances between the cubes complicating the problem. In the last experiment we have seen that the IK is very precise since it grasps the cube in a narrow situation. Now we pretend to test the capabilities of the inference and the task planning.

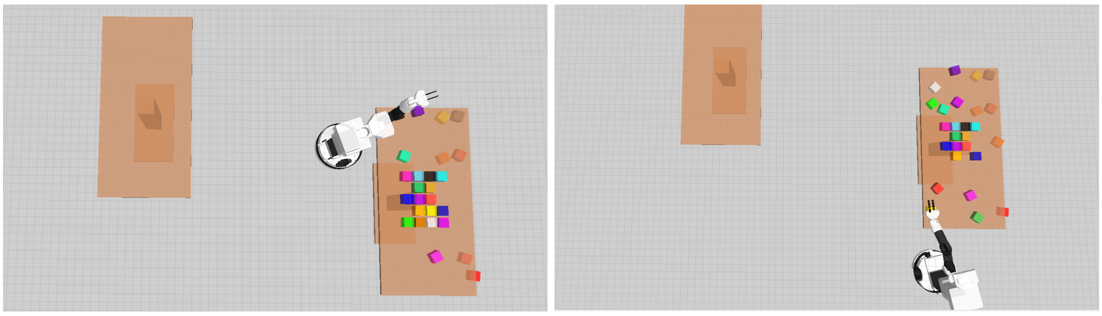
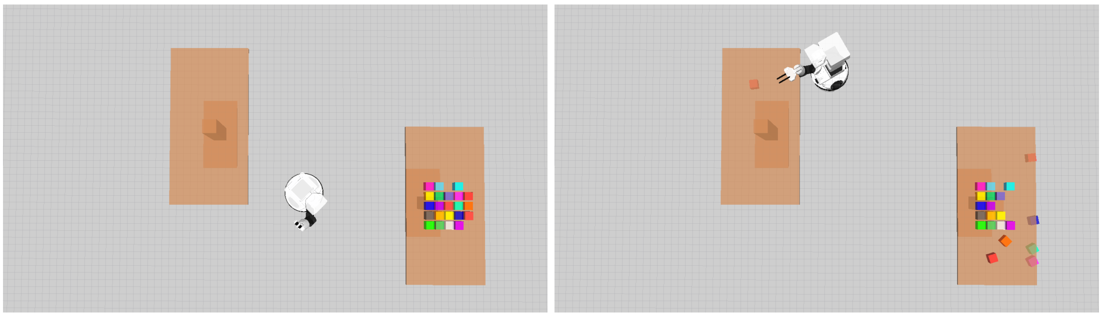


Figure 26: Screenshots of the experiment without finding solution

Since we have tested many times the previous problem reducing the distance without success, we have simplify it a bit maintaining the short distances between cubes. The main issue is that the IK founds double blocking situations when there are a lot of objects. This makes restart the PDDL problem, and causes that the robot tends to "explore" in new directions. Figure 27 shows one of the failed executions in which the problem is almost solved, but the robot decides to explore new directions. Instead of consider a structure of 25 cubes we have consider a structure of 22, and we have enlarged the size of the table to give more space to place the cubes.



(a) Initial conditions

(b) Goal conditions

Figure 27: Initial and goal conditions of the simplified experiment

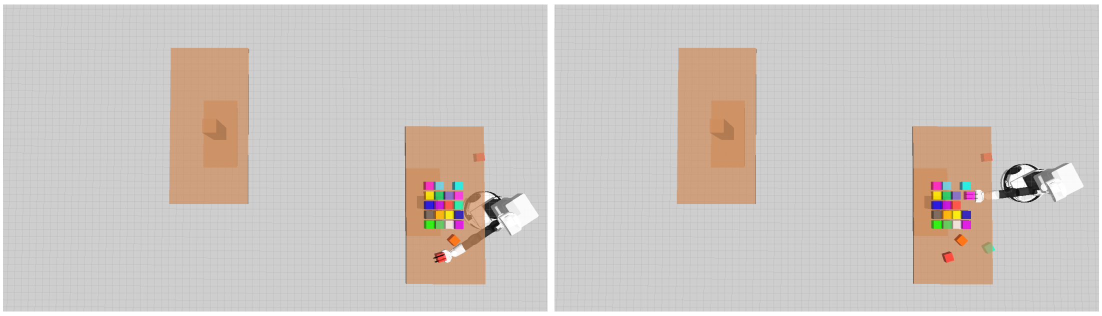


Figure 28: Screenshots of the simplified experiment finding solution

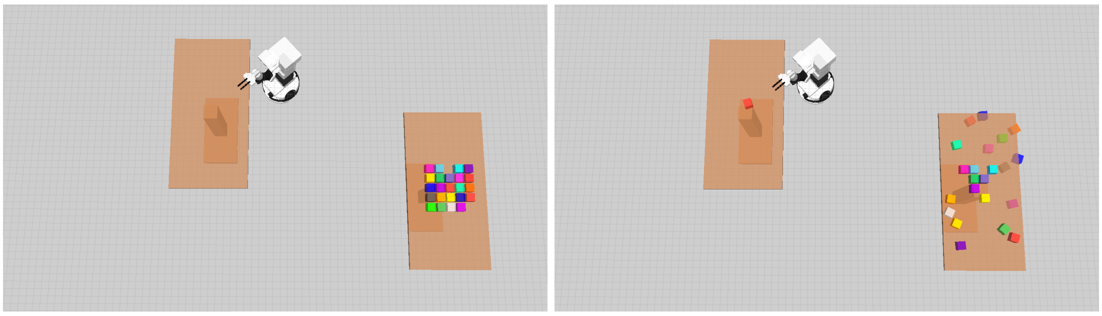
Table 7: Solver execution

Iteration	Solver used	Execution time (s)	Number of eval. states	Number of gen actions
Iter 1	FF	0	-	-
Iter 2	FF	0.64	25	13836
Iter 3	FD	6	78	-
Iter 4	FD	25	639	-

Table 8: Results experiment

Process	Execution time (seconds)
Solvers	32
RRT execution	56
Move Base	43
Total time execution	**594**

Table 8 shows that the main execution time is spent during the motion planning. But this result is not maintained when considering others situations as the ones from Figure 29. Table 9 and Table 10 show the results of this experiment. During this experiment the PDDL problem is restarted twice due to blocking situations. The results evince that the Task planner layer, in particular the solver, spends most of the execution time.



(a) Initial conditions **(b)** Goal conditions

Figure 29: Alternative experiment with solution

Table 9: Solver execution

Iteration	Solver used	Execution time (s)
Iter 1	FF	0
Iter 2	FF	0.65
Iter 3	FD	7
Iter 4	FD	28
Iter 5	FD	22
Iter 6	FD	26
Iter 7	FD	65
Iter 8	FD	84
Iter 9	FD	54
Iter 10	FD	81
Iter 11	FD	90
Iter 12	FD	77
Iter 13	FD	116
Iter 14	FD	116
Iter 15	FD	108
Iter 16	FD	110
Iter 17	FD	125

Table 10: Results experiment

Process	Execution time (seconds)
Solvers	1109
Total time execution	**1898**

The success and efficiency in this complex problems depends on the number of blocking conditions that the IK generates. Till there is no object blocking the grasping object we can not take it, even when the blocking object is rear the grasping object. Is fundamental choose the best grasping candidate (i.e the best solution) as the one which generates less new blocking conditions. But as shown in the simulations this heuristic not always choose the best way.

8 Cost planning

It's hard to describe in detail the total cost of this project. Basically, because the project doesn't have any material cost or rental cost associated, except the hours of work. Moreover it's hard to assign a number of hours, since a lot of tools and libraries were already developed by the company once I arrived, or some of them are open libraries such as Eigen or SBPL.

Anyway Table 11 details the costs for manufacturing each one of the parts for the TIAGo robot.

Table 11: Cost in materials of the TIAGo robot

Part of the robot	Total cost in materials(€)
Mobile base	5.150,43
Torso	4.457,64
Arm	10.909,01
Wrist	3.826,29
Gripper	1.695,59
Head	2.543,28
Total cost	**28.582,23**

9 Sustainability Project

In this section we will analyse the impact of the project from all the points of view.

Starting by the environmental impact, this project doesn't cause any kind of impact, since to develop it we have just used a single computer. It could be analysed in detail the processes and materials used for manufacturing the TIAGo, but nowadays the company doesn't have a study of this type.

From an economic or social point of view is difficult to estimate the impact it may have in our society. Is clear that this project makes the robot more autonomous, and is a first step to make the robots develop human tasks such as cook, do the laundry, layout the table or much other tasks. This makes us ask if the robots are gonna replace employees in the near future, or if they are gonna increase our possibilities and make our life easier.

So the economical or social impact of this project could be seen as a high advance to develop service robots that will help in our daily lives, and will represent a transformation of our jobs. When for detractors this project will represent a way of destroying jobs, and make the companies earn more money by substitute his employees by machines.

But there is also another social benefit around robotics that is rarely mentioned, that is that robots actually could take care of us. Nowadays there are a lot of European projects that try to focus on this topic, by developing technology that will help old people or sick people. All this projects are based on help people that live in remote regions by teleoperate a robot; or alert the emergency service when an accident occurs. Furthermore robots allow to interact with humans in a different way, and could be seen as a friend or a game for the kids.

In all chases, is philosophise around this topic since we can not predict the future and how the robots will affect on it. This project just pretends that robots could develop useful and complex tasks by just giving simple goals. This change will make that robots could be easily "programmable" by any kind of people not necessary engineers.

10 Conclusions

This project presents an approach that combines task planning and motion planning, and is able to solve non-trivial planning problems by just giving simple goals. All the geometric information from the environment is modelled in a PDDL, that tries to be generic for many robot tasks. Creating a single PDDL and giving different objectives, the robot is capable to solve a variety of planning problems, in different environments.

Moreover during the development of the actions we have tested different frameworks trying to compare them, and make the robot more efficient in his actions. This improvements allowed the robot to pick objects from a shelf which is a really hard situation.

Finally we also describe how the FF solver works, since is the one that offers better results compared with the others. In our experiments we present the time of execution for task planning and motion planning. We conclude that when the problem is complex, i.e there are many objects or conditions, the solver spends much of the time task planning. This fades the idea to create a single generic PDDL including all the feasible actions of the robot and his conditions. But in a near future, when computers will be more efficients, this will be possible and we are pretty sure on the application of the project since it will allow robots to work autonomously by just giving simple instructions.

It remains to test all this work on the real robot. And the next step will be to generate the actions for another robots of the company. A positive point on this project, is that it could be extrapolated to any other robot. Furthermore generating new PDDL's and new actions, the robot will be possible to develop completely different planning problems, or even problems in which many multiple robots are involved.

11 References

[1] Bullet Collision Library. `https://github.com/bulletphysics/bullet3`.

[2] Flexible Collision Library. `https://github.com/flexible-collision-library/fcl`.

[3] Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtós, N., and Carreras, M. *Rosplan: Planning in the robot operating system*, vol. 2015-January. AAAI Press, 2015, pp. 333–341.

[4] Coulter, R. C. Implementation of the pure pursuit path tracking algorithm. Tech. Rep. CMU-RI-TR-92-01, Robotics Institute, Pittsburgh, PA, January 1992.

[5] Egerstedt, M. Course about control of mobile robots. Georgia Institute of Technology.

[6] Gockenbach, M. S. Course in optimization theory. `http://www.math.mtu.edu/~msgocken/ma5630spring2003/lectures/sqp1/sqp1.pdf`.

[7] Goodman, B. Sequential quadratic programming. `https://optimization.mccormick.northwestern.edu/index.php/Sequential_quadratic_programming`.

[8] Hoffmann, J., and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Int. Res. 14*, 1 (May 2001), 253–302.

[9] Hoppe, D. R. H. Course in optimization theory. `https://www.math.uh.edu/~rohop/fall_06/index.html`, 2006.

[10] Koh, K. C., and Cho, H. S. A smooth path tracking algorithm for wheeled mobile robots with dynamic constraints. *J. Intell. Robotics Syst. 24*, 4 (Apr. 1999), 367–385.

[11] Kurt Konolige, E. M.-E. Navfn package from ROS. `http://wiki.ros.org/navfn`.

[12] LaValle, S. M. Rapidly-exploring random trees: A new tool for path planning. Tech. rep., 1998.

[13] Likhachev, M. Search based planning with motion primitives. `http://www.cs.cmu.edu/~maxim/files/tutorials/robschooltutorial_oct10.pdf`.

[14] Limpert, N., Schiffer, S., and Ferrein, A. A local planner for ackermann-driven vehicles in ROS SBPL. In *Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech), 2015* (2015), IEEE, pp. 172–177.

[15] Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. PDDL - The Planning Domain Definition Language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[16] Nocedal, J., and Wright, S. J. Numerical optimization, second edition. *Numerical optimization* (2006), 497–528.

[17] Pan, J., Chitta, S., and Manocha, D. FCL: A general purpose library for collision and proximity queries. In *IEEE International Conference on Robotics and Automation (ICRA)* (2012), IEEE, pp. 3859–3866.

[18] Ramirez, M., Lipovetzky, N., and Muise, C. Lightweight Automated Planning ToolKiT. `http://lapkt.org/`, 2015. Accessed: 2016-09-18.

[19] Schulman, J., Duan, Y., Ho, J., Lee, A., Awwal, I., Bradlow, H., Pan, J., Patil, S., Goldberg, K., and Abbeel, P. Motion planning with sequential convex optimization and convex collision checking. *Int. J. Rob. Res. 33*, 9 (Aug. 2014), 1251–1270.

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

[21] Twigg, C. Catmull-Rom splines. `https://www.cs.cmu.edu/~462/projects/assn2/assn2/catmullRom.pdf`.

[22] Based Planning Lab, S. SBPL package from ROS. `http://wiki.ros.org/sbpl`.