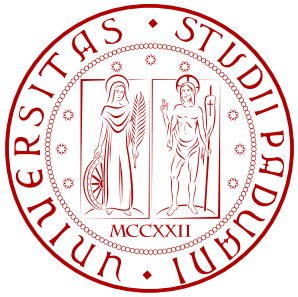


University of Padova

Department of Information Engineering
Master of Science in Computer Engineering

A Sampling-Based Tree Planner for Robot Navigation Among Movable Obstacles

Student:
Nicola Castaman

Supervisor:
Prof. Enrico Pagello

Co-supervisor:
Dott.ssa Elisa Tosello

Academic Year 2015 - 2016

Abstract

Over the last decades, many Motion Planning algorithms have been developed in order to find a continuous robot motion connecting a robot start configuration S and a goal configuration G. Traditional algorithms limit the search of the path within the collision-free space, while avoiding contacts with obstacles in the scene. This is in contrast with how humans naturally act, utilizing their manipulation capabilities to modify the environment to assist locomotion. If necessary, humans do not hesitate to move objects, such as chairs, out of their way to reach an otherwise unreachable goal. This thesis aims to bring robots closer to such capabilities proposing a planner that solves *Navigation Among Movable Obstacles* (NAMO) problems giving robots the ability to reason about the environment and choose when manipulating obstacles. It finds a path from a robot start configuration S to a goal configuration G taking into consideration the possibility of moving objects if G cannot be reached or if moving objects may significantly shorten the path. The planner combines the A*-Search and the exploration strategy of the *Kinodynamic Motion Planning by Interior-Exterior Cell Exploration* (KPIECE) algorithm. It is locally optimal and independent from the size of the map and from the number, shape, and position of obstacles. It assumes full world knowledge but the world is completely reconfigurable and it can be easily extended in order to explore unknown environments.

Contents

1. Introduction

Over the last decades, many Motion Planning algorithms have been developed in order to find a continuous robot motion connecting a robot start configuration S and a goal configuration G. Traditional algorithms limit the search of the path within the collision-free space, while avoiding contacts with obstacles in the scene: the robot sees environment objects as obstacles. Moreover, these algorithms are not designed for systems with complex dynamics such as humanoids or mobile manipulators, which have manipulation capabilities. This is in contrast with how humans naturally act, utilizing their manipulation capabilities to modify the environment to assist locomotion. If necessary, humans do not hesitate to move objects, such as chairs, out of their way to reach an otherwise unreachable goal. Future robots should demonstrate similar behavior, using their manipulation capabilities to move or even use environment objects.

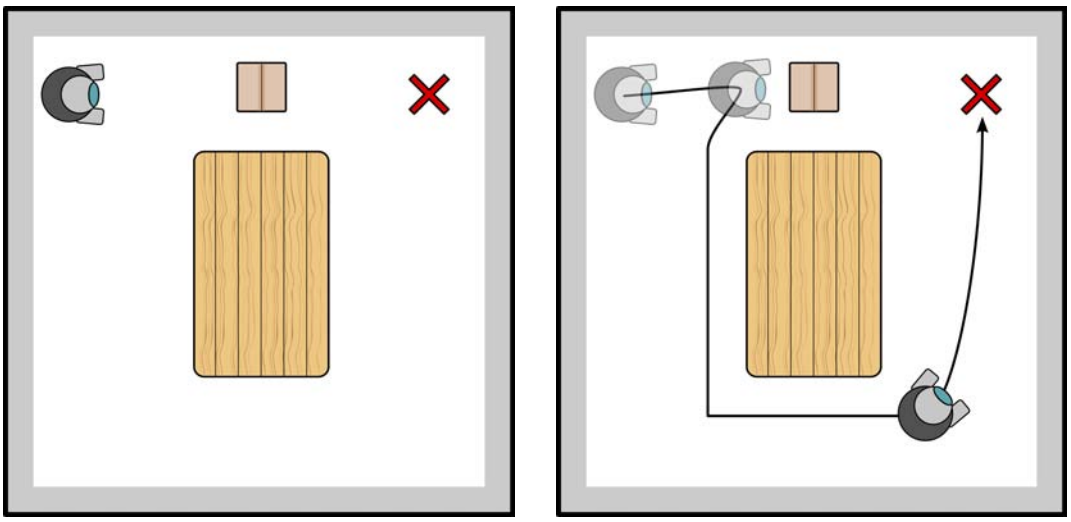
This work, based on a work already published in [2], aims to bring robots closer to such capabilities, to enable robots to autonomously reason about the environment and to change it to reach the goal. For a robot, not all environment objects have to be obstacles and it should be able to move each manipulable object out of the way to archive a task.

All developed packages are public released as open-source and available at `https://bitbucket.org/account/user/iaslab-unipd/projects/NAMO`.

1.1. Motivation

Most of existing motion planning algorithms implemented for actual robots search for collision-free paths. Assume, for example, the scenario of Figure 1.1a where the robot has to reach a Goal but the shortest way is restricted by a box. Figure 1.1b depicts the actual robots behavior: when the robot sees the obstacle that occludes the goal it search for a collision-free path that circumnavigate the collision space.

Humans behave in a different way, if it is necessary or less strenuous, they manipulate objects in order to create free spaces while minimizing efforts and time necessary to reach the goal. If the manipulation is too onerous (e.g., object is heavy, time required to manipulate the object exceeds the time to perform the collision-free path), then humans will interrupt the task and walk through the existing free space. Figure 1.2b depicts the human behavior in the same scenario of the robot. The human sees the obstacle and first tries to move it instead of circumnavigating the



(a) Configuration (b) Solution obtained with actual planner

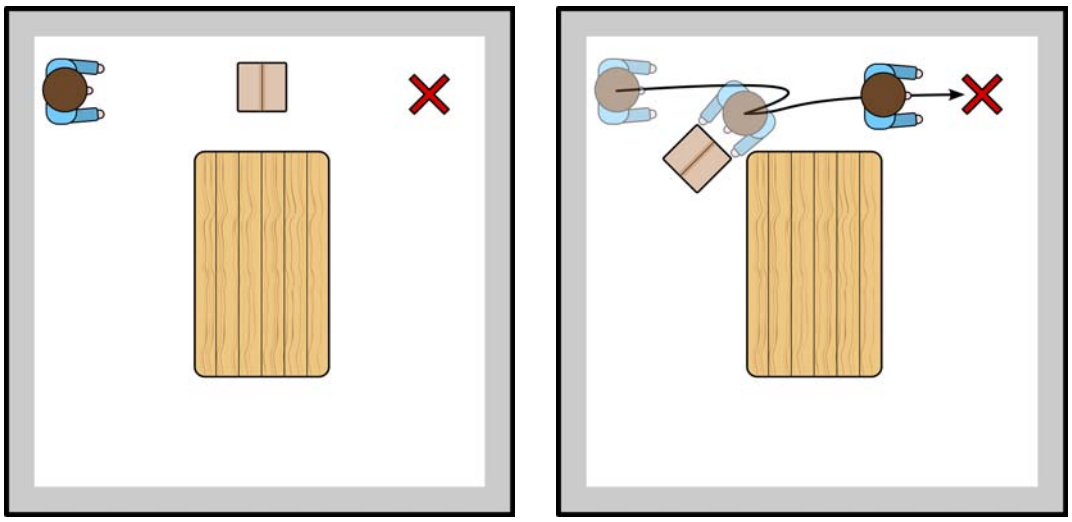
Figure 1.1.: Robot behavior: walk to the Goal; detect the obstacle; search a collision-free path; reach the Goal.

collision space that occludes the goal.

Robots that collaborate with humans in future industries or future rescue robots that save humans from disasters should behave in the same way. Robots should quickly decide which objects must be moved and where to move them in order to create an accessible path or clear a path to reach the Goal. Moreover robots should choose how to manipulate objects and compute a motion plan that integrate manipulation and navigation.

In fact, in a real scenario, such as in case of earthquake disasters, it could be impossible to find a collision free path connecting two states because of obstructions. Moreover, the minimum cost path allowing a robot to reach a planned pose could not involve a carefully collision-free navigation around the clutter. Sometimes, creating gaps among obstacles can be necessary or can considerably reduce navigation costs.

Think for example to a service robots, it will have to be able to autonomously navigate inside a home or an office while opening doors, moving chairs, etc. Rescue robots, should instead, be able to act in areas affected by disasters such as mining accidents, floods, and earthquakes; they should be able to decide when moving obstacles out of the way, in order to reach and save human lives as soon as possible. An example could be the scenario of an incident at a nuclear power plant, where the radiation levels is too high for workers to enter in damaged buildings. The use of robots turn out to be the right choice, however, current robot technologies prove to be limited. Robots require constant teleoperation, which in turn require operators to be within close proximity to the power plant at all times in full protection suits. In this way, there is an increase of the difficulty of operating the robots



(a) Configuration (b) Solution obtained by human

Figure 1.2.: Human behavior: walk to the Goal; detect the obstacle; try to remove the obstacle; reach the Goal.

teleoperation command terminal while workers are exposed to radiation. In addition, if communication between the operation terminal and the robot breaks, the robot will be lost and the task could not be completed. If robots are able to operate more autonomously and reach target areas without constant supervision and teleoperation, radiation exposure of workers could be lower and task completion rate increase.

Navigation Among Movable Obstacles (NAMO) gives robots the ability to reason about the environment and choose when manipulating obstacles [21, 23, 22]. It plans robot movements taking into consideration the possibility of moving objects if the goal cannot be reached or if moving objects may significantly shorten the path to the goal.

1.2. Challenges

The goal of this thesis is to develop a practical system that allow a robot to reason about the environment and to move obstacles away. There are two challenges that address NAMO planners: the very high dimensional of the state space in NAMO planning and the uncertainty that outcome on real systems and in physical interactions between a robot and its environment.

To understand the state space dimensionality, consider the navigation in a room with manipulable obstacles such as the ones depicted in Figure 1.1a. In contrast to planning for a single mobile robot with a fixed number of degrees of freedom, NAMO requires to consider the displacement of any movable object. The full state space

has the same dimension of that of a robot that has as many joints as the number of obstacles in the environment. In order to obtain an efficient solution, the robot will need heuristics for deciding which objects should be moved or where to move them.

In reality, robots have incomplete world knowledge and can only perceive the environment through limited sensory input, resulting in state and action uncertainty. To better understand why this might be a problem, consider again the example in Figure 1.1a. Maybe the robot knows that the shortest path to the goal involves moving the box, but it does not know whether the box is too heavy to be moved. How might the robot weigh the cost of moving the box? Moreover, the pose of the objects in the scene may be subject to uncertainty as well as the motion of the robot.

All these concepts are the base that motivate the proposed planning algorithms. The algorithm aims to be scalable: independent from the size of the map and from the number, shape and pose of objects. It has not to impose restrictions on actions to be performed: the robot can both push and grasp every object. Moreover, to be really useful the algorithm must ensure a real-time computation: the computation time cannot take more time than the time that the robot take to avoid obstacles. Fast planning and replannig is preferable to work in a real scenario. In the end, the implementation of the algorithm aims to achieve three important features: portability, reusability and flexibility. The algorithm has to be immediately compatible with a lot of widely used robots, that obviously have different features and DOF. Moreover, it has to work in different environment and not to be a problem specific solution.

The challenges of recognition and localization of objects are beyond the scope of this thesis. For this reason existing tools have been used in order to test to efficiency of the proposed planing algorithm on a simulated robot acting on a simulated scene.

1.3. Approach

In order to reach the objective proposed by this thesis, an algorithm that solves the NAMO problem is proposed; this algorithm combinine benefits of existing NAMO solvers with that of *Kinodynamic Motion Planning by Interior-Exterior Cell Exploration* (KPIECE) [25, 24], a promising sampling-based algorithm. Sampling-based motion planning algorithms explore the state space of the robotic system by growing a tree of valid motions from the start state of the system towards a goal region, using a model of motion [10]. In detail, complex systems motion planning can be solved by whose tree planners that only depend on forward propagating the model of motion, numerically evaluating motions only forward in time. KPIECE, in particular, is designed for systems with complex dynamics, where physics-based simulation is necessary and it simplifies the problem resolution by projecting the robot states space into a discretized Euclidean space. Moreover, KPIECE focuses exploration on the less covered areas by assigning an importance function that gives priority to unexplored cells. In a similar way to KPIECE, the developed algorithm

projects the three-dimensional cluttered workspace into an Euclidean space and discretize it by a grid. Consequently, the algorithm considers both objects in contact and not in contact with the ground. In the same way the algorithm projects the robot states space into the Euclidean space. An importance value is assigned to every cell. With this purpose, the importance function of KPIECE, was reformulated in order to assign an importance value to both free and occupied cells. The value corresponds to the robot effort required to reach the cell and eventually manipulate the obstacles inside it. As in KPIECE, the importance function favors the exploration of less explored areas. This behavior makes the robot more "curious", in a way similar to that used by humans.

Initially, the algorithm is developed to project the robot as a point-like robot in the two-dimensional space. This approach is a good step to prove the validity of the solution but is not enough. Indeed, the planner does not consider the occupancy of the robot and its state. Then, the algorithm was improved to take into consideration first the footprint of the robot, then its simulated state. The algorithms assumes full world knowledge but the environment is reconfigurable and it can be easily extended in order to solve NAMO problems in unknown environments. In fact, it is able to handle sensor feedbacks and correct uncertainties regarding obstacle poses and robot actions. Performed tests prove that it is locally optimal.

The obtained algorithm addresses two NAMO challenges: scalability and real-time computation. It is scalable because of its independence from the size of the map and from the number, shape, and pose of obstacles. It does not impose restrictions on actions to be performed: the robot can both push and grasp every object. In addition, it is real-time as proved by the performed experiments where the path was computed in a time less than a millisecond.

The implementation of the algorithm has been made ROS-compliant. The open-source Robot Operating System (ROS) [16] is a flexible framework for develop robot software. In fact, it creates an abstraction of the robot to the developer, so the developed software is instantly usable in all compatible robots. Moreover, it is extensively used among the robotics community and is compatible with a lot of widely used robots. This allow the proposed solution to be portable, reusable and flexible. The proposed NAMO algorithm extends the ROS navigation package allowing the assignment of different weights to whose cells of the 2D costmap that are populated by obstacles.

1.4. Overview

The rest of this thesis work is organized as follows:

Chapter 2 - Related Work provides an overview of related work in NAMO planning, robotic planning under uncertainty, and Sampling-Based Motion Planning;

Chapter 3 - NAMO Planning Domain introduces NAMO as a robot planning domain and presents its technical challenges;

Chapter 4 - NAMO Planner gives a practical solution for NAMO, describe in depth the algorithm proposed for the navigation and gives an overview on the object manipulation and sensors feedbacks;

Chapter 5 - Implementation Details describes the implementation of the proposed algorithm in a ROS-compliant package and gives an overview on what is ROS and how it works;

Chapter 6 - Experiments describes the performed experiments and discusses the obtained results, moreover describes the simulation performed and robot used for this purpose;

Chapter 7 - Conclusions and future works provides concluding remarks and outlines future research directions;

Appendix A - A* gives an overview on how the A* algorithm works;

Appendix B - KPIECE describe how the KPIECE algorithm works.

2. Related Work

The work presented in this thesis aims to improve existing NAMO algorithms by adding a Sampling-based Motion Planning. In this chapter the state of the art of both NAMO and Sampling-Based Motion Planning algorithm is discussed to provide an overview of researches now available.

2.1. NAMO Planning

Navigation Among Movable Obstacle (NAMO) has always been a challenge, even considering complete environment knowledge. As stated in [21, 23, 28, 14], Wilfong [27] first proved that deterministic NAMO with an unconstrained number of obstacles is NP-hard. Demaine [3] showed that even when considering only unit square obstacles the problem remains NP-hard.

In [21], Stilman solved a subclass of NAMO problems, namely LP_1, where disconnected components of free-space could be connected independently by moving a single obstacle. As stated in [23], the planner was successfully implemented on the humanoid robot HRP-2. Subsequentely, Van Den Berg with Stillman presented a probabilistically complete algorithm for NAMO domains [26]. However, all these planners solved NAMO problems assuming full world knowledge.

Following researches by Wu [28] and Kakiuchi [6] bring a solution to NAMO problems even in unknown environments. Wu, in [28], discretizes the environment in a grid and calculates a plan from Start to Goal through an A*-Search using the Euclidean distance as heuristic. Authors presented a baseline as well as an optimized approach. The baseline approach calculates plans for all possible actions on all known objects for any change in the environment. The method does not scale for larger environments. The optimized method does not automatically recompute plans if new information becomes available. When encountering new obstacles, [28] recomputes plans only when the current one becomes invalid due to collisions. For every new obstacle considered for the re-plan, it evaluates push actions limiting their number by the cost of just avoiding the obstacle itself. It maintains an ordered list of costs relative to the manipulation of each object and, at every step, it selects the minimum cost one. This approach reduces the number of objects that have to be evaluated during the selection but it does not guarantee local optimality, it supports only push actions, and it constrains obstacle shapes to be rectangular. Levihn, in [14], improves [28] guaranteeing local optimality. It supports a larger action set

(both pushes and grasps) and arbitrary object shapes. It introduces a dynamic bound that limits the number of obstacles evaluations, and it maintains two lists of cost underestimates for every plan. The first list maintains the minimum distance that the robot will have to cover if it moves a specific obstacle. This estimate is less informed but is fast to evaluate and remains constant throughout the entire execution. The second list is populated with the information of the obstacles already evaluated. Furthermore, a dynamic upper bound of the effort required to avoid all obstacles is stored. The use of the two lists and of the upper bound increases the computational efficiency and guarantees local optimality. Authors of [14] proved that the algorithm guided the robot to the goal in a time between 18 and 50 seconds in an environment containing between 2 and 70 randomly generated obstacles in randomized configurations. Kakiuchi [6] presented a solution for NAMO in unknown environments and experiments it on the humanoid robot HRP-2 using only on-board sensors. Movable obstacles are detected using active sensing and a color range sensor, and when an obstacle is moved, the perception of the environment is reconstructed.

2.1.1. NAMO Planning with Uncertainty

As stated in [11], the uncertainty in state and action outcome on real systems is a primary challenge in developing practical systems that allow a real robotic system to reason about environment objects. In literature there is a class of algorithms designed for mobile robot navigation with initial uncertainty about the poses of obstacles in its workspace. These algorithms discretized the states space, and employ methods based on A* search. They assume that space is free until it discovers to be blocked, and replan when the current plan is made impracticable by newly discovered obstacles. Early versions used a relatively straightforward planning strategy instead more advanced versions, e.g. D* [19], employ algorithmic techniques to ensure computational efficiency and limit the frequency of replanning.

A modern and interesting approach was proposed by Levihn [13, 12, 11]. In these works, authors uses ideas from decision theory to formally represent the uncertainty in NAMO for scenarios that require the robot to move a single object out of the way to connect two free-space regions. Authors define the NAMO problem as a *Markov Decision Process* (MDP).

General strategy is to construct a MDP that looks like the real problem, but can be solved in linear time for typical environments. The construction of this MDP builds on two insights of the domain. First, the state space is abstracted into a small set of states. These states are free-space regions that indicate that the robot can move collision-free between any two configurations within the region. Second, there is a small number of actions that lets the movement within this state space: since each free-space region is circumscribed by obstacles, an action that creates an opening to the neighboring free-space is defined for each obstacle. Together these two ideas permit the construction of a MDP. This representation by itself, however,

is insufficient. To solve the MDP, Levihn proposed to use *Monte Carlo* methods.

2.2. Sampling-based Motion Planning

One of the first successful sampling-based motion planners was the *Probabilistic Roadmap Method* (PRM) [7]. The algorithm subdivides the search of a valid path into a learning and a query phase. The former takes random samples from the configuration space of the robot, tests them for whether they are in the free space, and uses a local planner in order to attempt to connect these configurations. The latter adds the starting and goal configurations to the graph and applies a graph search algorithm in order to determine a path. Starting from PRM, many other algorithms were developed in order to better guide the tree expansion. *Rapidly-exploring Random Trees* (RRT) [8] expand from states closed to randomly produced states, *Expansive Space Trees* (EST) [5] and *Single-query Bidirectional probabilistic roadmap planner with Lazy collision checking* (SBL) [18] attempt to detect less explored regions and expand from them. *Kinodynamic Motion Planning by Interior-Exterior Cell Exploration* (KPIECE) [25, 24] finnaly improves the decision phase by making better use of the information collected during the planning process. This information is used to decrease the amount of forward propagation the algorithm needs. As consequence, both runtime and memory requirements decrease making the algorithm suitable to handle high dimensional systems with complex dynamics. As stated in [24], the exploration strategy of KPIECE projects the state space to a lower dimensional Euclidean space and discretizes it by using a grid. The discretization is used to estimate the coverage of the state space and to evaluate cells goodness: for every cell, the algorithm saves the number of times it has been explored and the progress achieved by exploring it. Combining collected information, KPIECE is able to deterministically select the regions to explore: the best less explored ones.

3. NAMO Planning Domain

This chapter defines the NAMO domain and gives an interpretation of that domain in *configuration space*. Consequently, it gives a description on the action that a robot can execute in NAMO domain.

3.1. Problem Statement

As stated by Stilman in [20], the NAMO domain is analyzed as an instance of geometric path planning [9].

In path planning, a complete description of the geometry of a robot $\mathcal{A}$ and of a workspace $\mathcal{W}$ is provided. The workspace $\mathcal{W} = \mathbb{R}^N$, in which $N = 2$ or $N = 3$, is a two-dimensional or three-dimensional Euclidean space.

Objects and robot are represented as polyhedrons in the three-dimensional space. The environment objects are classified as either fixed or movable.

The workspace is populated by the following items:

- $\mathcal{O}_{\text{fixed}}$, a set of *Fixed Obstacles* that must be avoided.
- $\mathcal{O}_{\text{movable}}$, a set of *Movable Obstacles* that the robot can manipulate.
- $\mathcal{A}$, a robot with n degrees of freedom with manipulation capabilities.

The goal for a path planning algorithm is to find a path for $\mathcal{A}$ from an initial position and orientation (pose) to reach a goal position and orientation. To achieve that, a complete specification of the location of every point on the robot geometry, or a *configuration* $\boldsymbol{q}$, must be provided. The *configuration space*, or C-space ($\boldsymbol{q} \in \mathcal{C}$), is the space of all possible configurations. The C-space represents the set of all transformations that can be applied to a robot given its kinematics. Motion planning researches recognize that the C-space is a useful way to abstract planning problems in a unified way. The advantage of this abstraction is that a robot with a complex geometric shape is mapped to a single point in the C-space. The number of degrees of freedom of a robot system is the dimension of the C- space, or the minimum number of parameters needed to specify a configuration.

Let the closed set $\mathcal{O} \subset \mathcal{W}$, where $\mathcal{O} = \mathcal{O}_{\text{fixed}} \cup \mathcal{O}_{\text{movable}}$, represent the obstacle region, which is usually expressed as a collection of polyhedrons. Let the closed set $\mathcal{A}(\boldsymbol{q}) \subset W$ denote the set of points occupied by the robot when at configuration

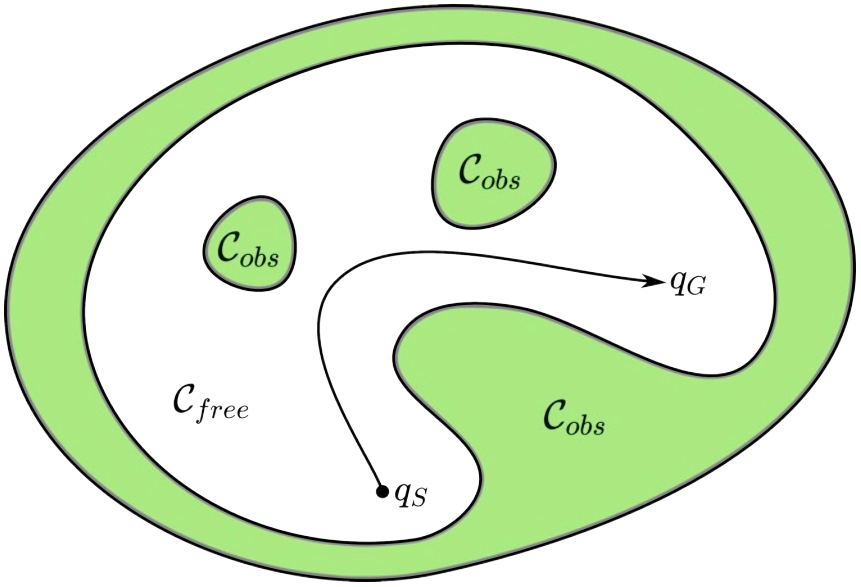


Figure 3.1.: Motion planning in configuration space. Obstacles are part of $\mathcal{C}_{obs}$, meaning that placing the robot in a configuration within that region would lead to collision. The remaining space represents $\mathcal{C}_{free}$, the collision-free region.

$\boldsymbol{q} \in \mathcal{C}$; this set is usually modeled using the same primitives used for $\mathcal{O}$. The C-space obstacle region, $\mathcal{C}_{obs}$, is defined as

$$\mathcal{C}_{\text{obs}} = \{\boldsymbol{q} \in \mathcal{C} | \mathcal{A}(\boldsymbol{q}) \cap \mathcal{O} \neq \emptyset\} \tag{3.1}$$

Since $\mathcal{O}$ and $\mathcal{A}(\boldsymbol{q})$ are closed sets in $\mathcal{W}$, the obstacle region is a closed set in $\mathcal{C}$. The set of configurations that avoid collision is $\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}$, and is called the *free space*.

For a traditional path planning algorithms, the goal is to solve the problem of finding a continuous path that moves the robot $\mathcal{A}$ gradually from a start configuration $\boldsymbol{q_S}$ to a goal configuration $\boldsymbol{q_G}$ while never touching any obstacle:

$$\tau : [0, 1] \rightarrow \mathcal{C}_{\text{free}} \tag{3.2}$$

with:

$$\tau(0) = q_S \text{ and } \tau(1) = q_G \tag{3.3}$$

A NAMO planning algorithm solves the same problem but it is possible to move one or more obstacles in $\mathcal{O}_M$:

$$\tau : [0, 1] \rightarrow \mathcal{C}_{\cancel{\text{free}}} \tag{3.4}$$

with:

$$\tau(0) = q_S \text{ and } \tau(1) = q_G \tag{3.5}$$

this means that all possible configurations $C = C_{\text{free}} \cup C_{\text{obs}}$ are allowed.

An instance of the NAMO problem can be formally defined by a tuple $S = (\mathcal{C}, \mathcal{U}, q_S, q_G, f)$. Where:

- $\mathcal{C}$ is the configuration space. As seen before $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}$;
- $\mathcal{U}$ is the control space;
- $q_S \in \mathcal{C}_{free}$ is the robot initial configuration;
- $q_G \in \mathcal{C}_{free}$ is the robot goal configuration.
- $f : \mathcal{C} \times \mathcal{U} \to \text{Tg}\mathcal{C}$ is the forward routine describing the dynamics, where Tg$\mathcal{C}$ is the tangent bundle of $\mathcal{C}$.

A solution of the NAMO problem consists of a sequence of controls $u_1, ..., u_n \in \mathcal{U}$ and times $t_1, ..., t_n \in \mathbb{R}^{\geqslant 0}$ such that $q_0 = q_S$, $q_n = q_G$ and $q_k \in \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{movable}}$, $k = 1, ..., n-1$ can be obtained sequentially by integrating f. This means that the motion plan can iterates walking, grasping and moving obstacles until the robot is at goal q_G.

During the planning, it is assumed that the geometry and kinematics of the environment and the robot are known. Also, it is assumed that there is no uncertainty in sensing and effects of robot actions.

3.2. Operators

In order to achieve the goal configuration, the robot is permitted to change its own configuration and possibly the configuration of a grasped obstacle. It is possible distinguish between two primitive operators or actions: *Navigate* and *Manipulate*. Each action is parameterized by a path $\tau(q_i, q_j)$ that defines the motion of the robot between two configurations: $\tau : [0, 1] \to \mathcal{C}$ where $\tau(0) = q_i$ and $\tau(1) = q_j$.

Let $\mathcal{W}^t$ the world state at any time t that defines the position and orientation of the robot links and each object. The world state can be represented as follows:

$$\mathcal{W}^t = (t, q^t, \mathcal{O}^t_{movable}) \tag{3.6}$$

where q^t is the robot configuration at instant t and $\mathcal{O}^t_{movable}$ are the positions of all movable obstacles at instant t.

The *Navigate* operator refers to contact-free motion. While the robot may be in sliding contact with an object, its motion must not displace any objects by collision or friction. Navigate simply moves the robot joints as specified by τ.

$$Navigate : (\mathcal{W}^t, \tau(q^t, q^{t+1})) \to \mathcal{W}^{t+1} \tag{3.7}$$

When the robot motion affects the environment by displacing an object, $\mathcal{O}_i$, it refers to the action as *Manipulate*. The manipulate operator consists of two paths: one for the robot and one for $\mathcal{O}_i$. Since the object is not autonomous, the object path is parameterized by the robot path and the initial contact or grasp $\mathcal{G}_i \in \boldsymbol{G}(\mathcal{O}_i)$. The set $\boldsymbol{G}(\mathcal{O}_i)$ consists of all possible point of contact between the robot end-effector and the object. Distinct $\mathcal{G}_i$ lead to different object motions.

$$Manipulate : (\mathcal{W}^t, \mathcal{O}_i, \mathcal{G}_i, \tau(q^t, q^{t+1})) \rightarrow \mathcal{W}^{t+1} \tag{3.8}$$

Manipulate maps a state, contact and path to a new world state where the obstacle $\mathcal{O}_i$ have been displaced. The action is valid when neither the robot nor object collide or displace other objects.

The two action descriptions point to a general formulation for interacting with environment objects. The robot iterates a two step procedure. First, it moves to a contact state with the *Navigate* operator and then applies a *Manipulate* operator to displace the object. The robot also uses *Navigate* to reach a goal state.

3.3. Actions

In Section 3.2, *Manipulate* operators has not receive a precise definition. In this section we give two type of actions for manipulating objects. In each case, the actions translates the trajectory of the robot into a motion for the object.

Grasping The simplest method for move and manipulate an object is when the object is rigidly grasped by the robot. A grasped object remains at a fixed transform relative to the robot end-effector. To move an object, the robot must first *Navigate* to a grasping configuration and then *Manipulate*. In addition to requiring collision-free paths, a valid *Manipulate* operator constrains the initial state of the robot and object. Typically the contact must be a grasp that satisfies form closure. These criteria indicate that a desired motion of the robot will not cause it to release the object. Moreover, it is also possible to constrain grasps with regard to robot force capabilities. Grasping allow to easily predict the possible displacements for an object by looking at robot and object geometry. However, some objects, such as large boxes, are difficult or impossible to grasp. Constrained environments may also restrict robot positions to make grasping impossible or require the end-effector to move with respect to the object.

Pushing A manipulation action that does not require a grasp with closure is called non-prehensile. Given any contact between the robot and object, pushing manipulation restricts the path that the manipulator can follow in order to maintain a fixed transform between the end-effector and the object. The most studied version

of pushing manipulation is based on static friction during pushing [15]. At sufficiently low velocities, static friction prevents the object from slipping with respect to the contact surface. Given the friction coefficient and the friction center of the object we can restrict robot paths to those that apply a force inside the friction cone for the object. Pushing manipulation is more general than Grasping manipulation, however it needs more detailed modeling. In addition to geometry, this method requires knowledge of friction properties. An incorrect assessment would bring to slip causing an unplanned behavior.

4. NAMO Planner

In Chapter 3 was defined the NAMO domain and highlighted the complexity of motion planning with movable obstacles. Defining the domain in terms of configuration space allows to make useful observations about the planning problem.

While complete planning for NAMO may be very difficult to achieve, it is possible look at *Navigate* and*Manipulate* operators independently. However, the most interesting aspect of NAMO is the interdependence of the two actions. For instance, in order for the robot to manipulate an object it must be within reach of the object. It is not always possible to make contact with an object without previously moving another one.

The core of this thesis is to propose an algorithm that aims to resolve NAMO problems focusing on the *Navigate* operator independently from the *Manipulate* one.

Afterwards, for further information, in Section 4.2 a *Manipulate* method is proposed ti accomplish to grasp and push actions. In Section 4.3 the use of sensors feedback, in particular vision sensors, is depicted to correct uncertainties regarding robot actions and obstacle poses.

4.1. Navigation

Implementing a navigation algorithm is the focus of this thesis. This section depicts the proposed solution for navigation that can be used then in combination with manipulation. The proposed algorithm for navigation combined existing A*-based NAMO algorithms with the exploration strategy of KPIECE to create a NAMO planner. A detailed study of A* and KPIECE algorithms can be found in Appendix A and Appendix B.

4.1.1. Proposed Solution

Without loss of generality the domain is restricted to a planar projection of the three-dimensional environment. As for KPIECE, every $\boldsymbol{q} \in \mathcal{C}$ is projected into an Euclidean space E through a projection $Proj$. If $\boldsymbol{p} = Proj(\boldsymbol{q})$, then the coordinates of $\boldsymbol{p}$ in E will be:

$$Coord(\boldsymbol{p}) = Coord((p_1, ..., p_k)) = \left(\left\lfloor \frac{p_1 - o_1}{d_1} \right\rfloor, ..., \left\lfloor \frac{p_k - o_k}{d_k} \right\rfloor \right) \tag{4.1}$$

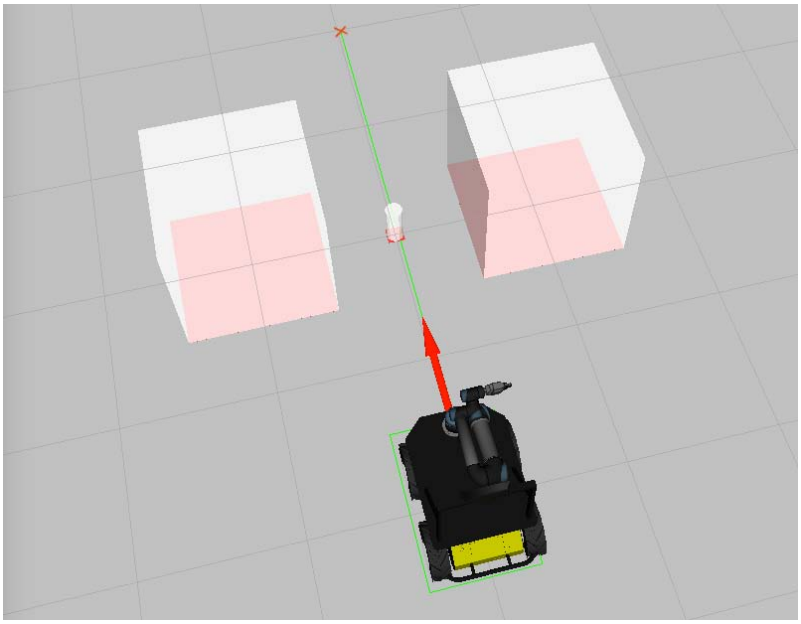


Figure 4.1.: The path generated by the algorithm.

where $(p_1, ..., p_k)$ are the components of $\boldsymbol{p}$, $(o_1, ..., o_k)$ is the origin of the Euclidean map, and $(d_1, ..., d_k)$ is its resolution.

E is discretized through a grid G of $N \times M$ cells of length d. Without loss of generality, d is chosen as propagation step size of the expansion tree. This means that $d = d_1 = ... = d_k$ will be the resolution of the map.

$$Cell(\boldsymbol{p}) = \{\boldsymbol{q} \in \mathcal{C} | Coord(Proj(\boldsymbol{q}) \in Cell(\boldsymbol{p}))\} \tag{4.2}$$

defines, for every $p \in E$, the corresponding cell of G.

A tree data structure T is defined. Every vertex $v_i \in T$ refers to the cell $Cell(i) \in G$; v_i points to the state of Q projected into $Cell(i)$ and used for the propagation. The algorithm proceeds as described in Algorithm 1.

T is initialized with v_S referring to $S \in Cell(S)$. At every iteration, the importance of cells referring to the current node and to its neighbors is updated. The node of the tree referring to the most important cell is selected and a state of it is chosen for the expansion. The process iterates until T reaches $Cell(G)$.

The importance of $Cell(i)$ is defined as:

$$Importance(i) = \frac{1}{\frac{1 + Distance(i) + Weight(i)_{\text{TOT}}}{1 + Selection(i) + Visits(i)}} \tag{4.3}$$

where:

- $Distance(i)$ is the effort done to cover the Euclidean distance separating $Cell(i)$ from $Cell(G)$; it reflects the A* search;

- $Selection(i)$ is the number of times that $Cell(i)$ was selected for expansion;

- $Visits(i)$ refers to the number of times that $Cell(i)$ was considered during the selection phase, namely it is the coverage of $Cell(i)$;

- $Weight(i)_{\mathrm{TOT}}$ is the cost of the path to be performed in order to reach $Cell(i)$, i.e., the sum of the weights of the cells that NAMO sampled as parents of $Cell(i)$.

$Weight(i)$ assigned to a cell $Cell(i)$ is defined as follows:

$$Weight(i) = \alpha \cdot Reach(i) + \sum_{k} (\beta \cdot Move(k, i) + \gamma \cdot Return(i)) \tag{4.4}$$

where:

- $Reach(i)$ is the effort done by the robot in order to reach q_i within $Cell(i)$ from the current state;

- $Move(k, i)$ is the effort required to remove the k-th obstacle from $Cell(i)$ and place it out of the Euclidean distance separating $Cell(S)$ and $Cell(G)$ ($0 \leq k \leq n$, n number of obstacles in $Cell(i)$);

- $Return(i)$ is the effort required to come back. In order to homologue data, efforts are represented as time variables.

Importance can be computed in constant time since all the values it depends on can be made readily available. Once visited a cell, its coverage is updated. Once selected a cell from which continuing the expansion, its selection rate is incremented and a robot state within it is sampled. A chain of states $Path = (q_0, ..., q_k, ..., q_n)$ results, with $q_0 = S$ and $q_n = G$. The robot real-time performs the $Path$.

It is easy to observe that, as KPIECE, NAMO prefers expanding from cells that are less covered rather than from cells that are well covered. Cells that have been selected for expansion fewer times are preferred over cells that have been selected many times. Moreover, NAMO gives priority to cells closer to the goal, i.e., less explored areas; and it prioritizes cells that carry the robot to make the least effort, combining navigation and manipulation efforts. Studies show that considering these heuristics in the selection of cells work well in practice [25]. Formulating these heuristics could facilitate the resolution of the NAMO problem in unknown environments.

Algorithm 1 NAMO

Input: A collision map Map, a start state $Start$, a goal state $Goal$
Output: A set of states $Path$ that minimize the cost of reaching $Goal$ from $Start$

Discretize Map through a grid G
Create a tree data-structure T of nodes n where $cell(n)$ is the cell of G of which n collects the ExpansionData(n)
Let S and G be respectively the $Start$ and $Goal$ nodes
T.pushBack(S)
$S.visits$++
$current \leftarrow S$
while $current \neq G$ **do**
    $current \leftarrow$ SelectBestCurrentNode(T)
    $current.selection$++
    $next \leftarrow$ SelectBestNextNode($current$)
    $next.visits$++
    T.pushBack($next$)
end while
for every n in T **do**
    Select a state s in $cell(n)$
    Add s in $Path$
end for
Let the robot perform $Path$

Algorithm 2 Select Best Current Node

function SelectBestCurrentNode(T)
    $current \leftarrow T$.begin()
    $max\_importance \leftarrow$ GetImportance($current$)
    $best\_current \leftarrow current$
    while $current \neq T$.end() **do**
        $current \leftarrow T$.next()
        $importance \leftarrow$ GetImportance($current$)
        if $importance > max\_importance$ **then**
            $max\_importance \leftarrow importance$
            $best\_current \leftarrow current$
        end if
    end while
    return $best\_current$
end function

Algorithm 3 Select Best Next Node

```
1: function SelectBestNextNode(n)
2:     best_next ← a random neighbour of n
3:     max_score ← 0
4:     while n has non-selected neighbours do
5:         next ← a random neighbour of n
6:         score ← GetScore(next)
7:         if score > next_score then
8:             max_score ← score
9:             best_next ← next
10:        end if
11:    end while
12:    return best_next
13: end function
```

4.2. Manipulation

Once detected the obstacle on the path, the robot has to decide which manipulation action to apply in order to move it. Generally, if the object is small enough (width or length less than the gripper maximum opening), the robot tries to grasp it, otherwise it proceeds with a push. Depending on the action, a different method has been implemented in order to geometrically compute the new object position (the re-estimation of the orientation is unnecessary). Every approach refers to the obstacle position (current or next) as the position of their Center of Mass. These actions are generated for a generic mobile manipulator robot.

4.2.1. The relocation routine of graspable objects

Figure 4.2 depicts the new positions generation process of a graspable object. Starting from the manipulator origin, n positions are generated around the robot. Every position $i(0 \leqslant i < n)$ has the following coordinates:

$$x_i = r_i * \sin\theta_i \tag{4.5}$$

$$y_i = r_i * \cos\theta_i \tag{4.6}$$

where

$$r_i = \begin{cases} \min & i = 0 \\ r_{i-1} + \Delta r & \text{otherwise} \\ \max & i = n-1 \end{cases} \tag{4.7}$$

$$\theta_i = \begin{cases} 0 & i = 0 \\ \theta_{i-1} + \Delta\theta & \text{otherwise} \\ 2\pi & i = n-1 \end{cases} \tag{4.8}$$

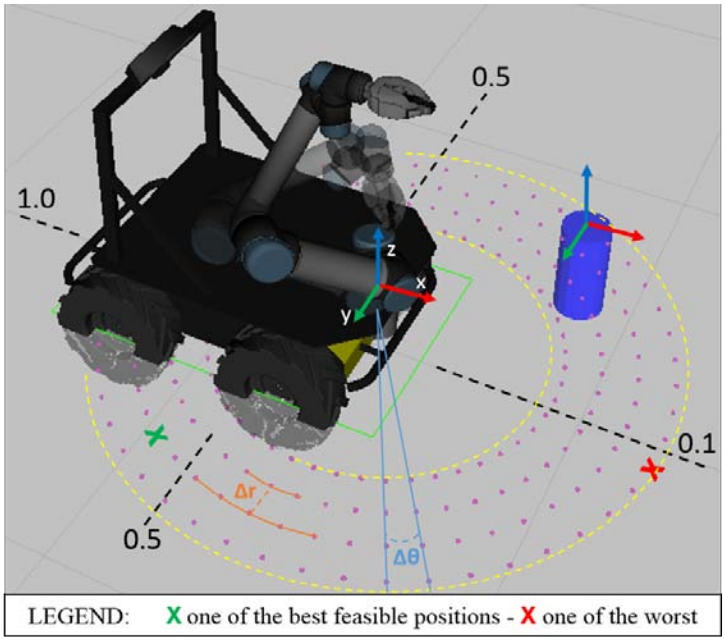


Figure 4.2.: The positions generation routine of a graspable obstacle.

Δr and $\Delta\theta$ are the angle and radius resolution respectively. min is the minimum reach of the robotics arm allowing the placement of the object out of the footprint polygon of the mobile base and max is its maximum reach.

In order to facilitate the new positions selection, a weight w_i is assigned to every i:

$$w_i = \begin{cases} 1 & (x_i, y_i)\text{behind the robot} \\ 0.1 & (x_i, y_i)\text{in front of the robot} \\ -\cos\theta & \text{otherwise} \end{cases} \tag{4.9}$$

Positions are inserted in aa ordered list L depending on w_i and on the distance d_i from the manipulator origin:

$$L = [(x_1, y_1, w_1, d_1), (x_2, y_2, w_2, d_2), ..., (x_n, y_n, w_n, d_n)] \tag{4.10}$$

with $w_1 > w_2 > ... > w_n$. If $w_i = w_j$, than $d_i \leqslant d_j$.

Typically, human navigation routines prefer a forward motion instead of a backward one. Inspired from this behavior, the implemented positions generation process prefers to relocate encountered obstacles behind the robot. This decision should minimize the probability of reconsidering the object again while solving the NAMO problem. If, during the displacement, no i is kinematically feasible and collision-free, than the constraint $ri \leqslant$ max is relaxed allowing the motion of the mobile base. Depending on the relaxation, L is reformulated.

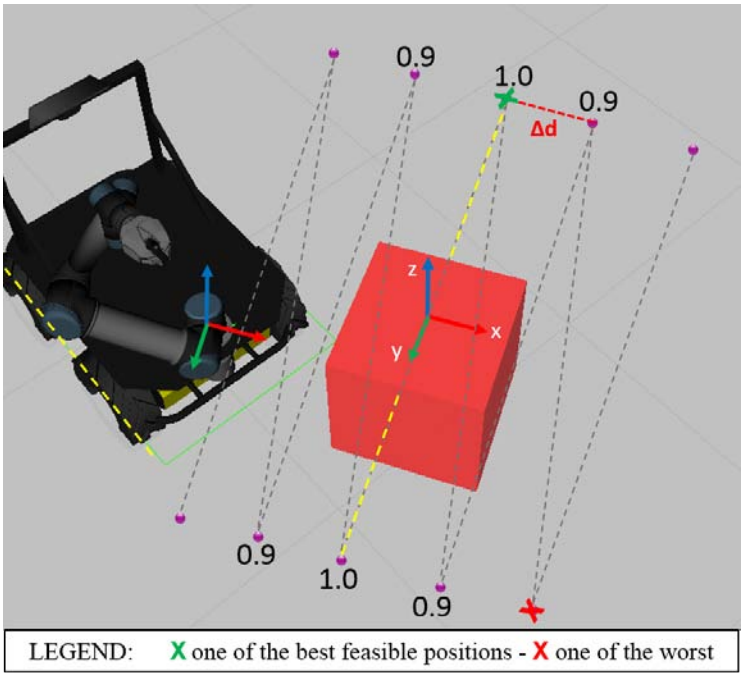


Figure 4.3.: The positions generation process of a pushable obstacle.

4.2.2. The relocation routine of pushable objects

Figure 4.3 depicts the positions generation process of a pushable object. A *Zig-Zag* mode is adopted: new positions are generating to the right or the left of the current one taking into account the space that the robot requires to move. i $(0 \leqslant i < n)$ has coordinates:

$$x_i = x_0 + i\Delta d \tag{4.11}$$

$$yi = y_0 \pm l \tag{4.12}$$

where (x_0, y_0) is the current position of the object, Δd is the distance resolution, and l is the robot maximum side.

As for the *Grasp* routine, a weight w_i is assigned to every i. The adopted rule follows:

$$w_i = \begin{cases} 1 & x_i = x_0 \\ \dfrac{1}{d_i} & \text{otherwise} \end{cases} \tag{4.13}$$

where $d_i = D((x_i, y_i), (x_0, y_0))$ is the distance between the new and the current object position with respect to the object reference system. An ordered list

$$L = [(x_1, y_1, w_1), (x_2, y_2, w_2), ..., (x_n, y_n, w_n)] \tag{4.14}$$

is formulated with $w_1 \geqslant w_2 \geqslant ... \geqslant w_n$.

In case of failed displacement, the routine increases l. L is reformulated.

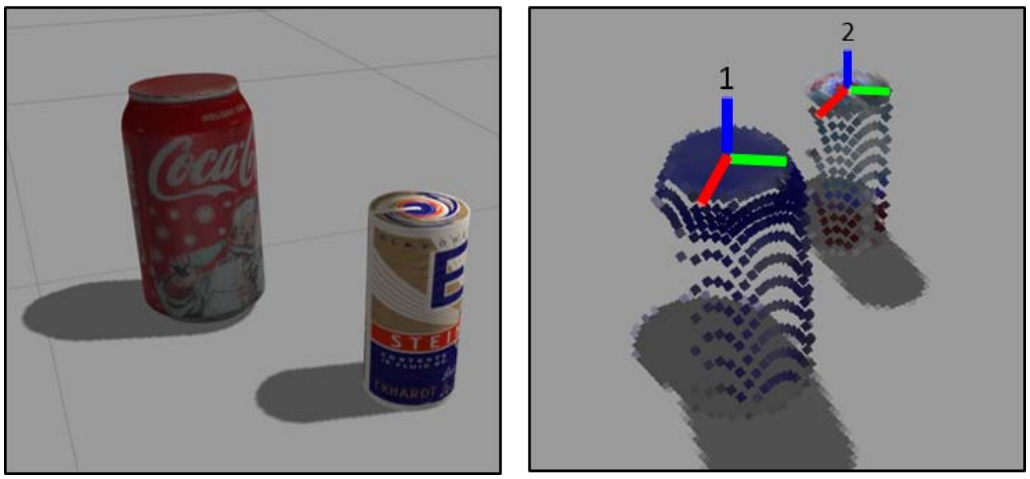


Figure 4.4.: Objects in the scene and their segmentations. A marker is visible for every reference system. Objects are ordered depending on the distance from the robot.

4.2.3. Displacement

The current state of the work requires that objects in the scene are known as well as the gripper poses necessary to grasp or push them. The tuples (*object*, *gripperpose*) are stored in a data set and retrieved when needed. Once L has been computed, the displacement routine starts: it extracts the positions in L starting from the first and try to place there the object eventually combing navigation and manipulation actions. The routine provides a collision checking of the object during its motion from the current to the goal position.

4.3. Sensors feedback

As stated before, while moving in a workspace, robots may have to deal with unexpected events. Moreover, the pose of the objects in the scene may be subject to uncertainty as well as the motion of the robot. For these reasons, sensors should be mounted on every robot and every automaton should be able to correct its actions based on sensor feedbacks. In this work, the Point Cloud Library [17] has been exploited in order to implement a routine able to read signals of a vision sensor and to process them in order to detect the scene, segment the obstacles, extract their coordinates and eventually recognize them (See Figure 4.4). The proposed algorithm takes this information in order to update the occupancy map and eventually recompute the NAMO path.

5. Implementation Details

Simulation and experiments are essential in order to evaluate the effectiveness of the proposed solution. As mentioned before in the introduction, the implementation of the algorithm have to fulfill three important features: portability, reusability and flexibility. Moreover, it has to be immediately usable with a lot of widely used robots. For these reasons, the implemented solution has been made ROS-compliant.

5.1. ROS

ROS[1] (*Robot Operating System*) [16] is a framework that is widely used in robotics. ROS makes available libraries and tools to help software developers to create robot applications that is immediately usable in a lot of robots. The philosophy is to make a piece of software that could work in other robots with only little changes to the code.

ROS is designed to be modular and is organized in software packages that can contain one or more nodes, which are processes where computation is done. To understand the ROS modularity, thinks to a system that control a movable robot: one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on.

Usually, a system will have many nodes to control different functions, and it is better to have many nodes that provide only a single functionality, rather than having a large node that makes everything in the system. A ROS node can be written using the ROS `roscpp` for C++ or `rospy` for Python. Nodes communicate with each other sending information using messages. A ROS message is simply a data structure that uses standard types or types developed by the user.

Messages, in ROS, are exchanged thanks to asynchronous nodes as depicted in Figure 5.1.

Topics Messages are exchanged with publish/subscribe model. A node sends out a message publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of message can subscribe to the appropriate topic, and it isn't necessary that the node that is publishing this topic should exist. It is important that

[1] http://www.ros.org/

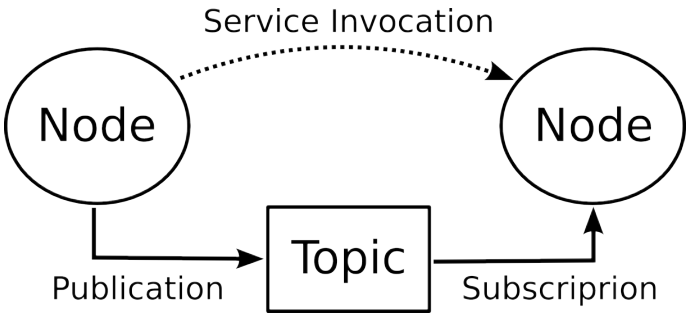


Figure 5.1.: Messages exchange system in ROS

the name of the topic be unique to avoid problems and confusion between topics with the same name. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. This permits to decouple the production of information from its consumption.

Services The publish/subscribe model is a very flexible communication paradigm, but sending data in a many-to-many fashion is not appropriate when is needed a request or an answer from a node. Services implements a request/response model, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. When a node has a service, all the nodes can communicate with it, thanks to ROS client libraries.

Nodes use topics and services by a peer-to-peer connection, but all the nodes have to communicate with the *Master* service to enable the entire connection. It provides name registration and lookup. Without the *Master*, nodes would not be able to find each other, exchange messages, or invoke services.

5.2. NAMO Package

ROS implements a 2D Navigation Stack[2] that takes information from odometry, sensor streams, goal pose and outputs safe velocity commands that are sent to a mobile base. To use the Navigation Stack, the robot must be running ROS (of course), have a `tf` transform tree in place, and publish sensor data using the correct ROS Message types.

This project aims to create a package that extends ROS Navigation Stack to implement the proposed solution for Navigation Among Movable Obstacles.

[2]http://wiki.ros.org/navigation

The 2D costmap (`costmap_2d`) implemented in ROS cannot handle information like the weight of obstacles populating the grid. Moreover, it does not distinguish between different obstacles that occupy the map, in fact this ROS package manage the space as free or obstruct. To overcome this lack, it was necessary to implement a new package that memorizes distinctly every object in the map. In this package, an object is memorized with its weight value. This object is then assigned to cells that occupy. To use this package in a simulated 3D enviroment, the sole of the 3D object shape is remapped into 2D polygons and then projected on the grid.

The packages developed to implement the improved costmap (`map_2d`) and to implement the proposed navigation algorithm (`namo_planner`) and used in the following experiments are explained in details below.

All developed packages are public released as open-source and available at `https://bitbucket.org/account/user/iaslab-unipd/projects/NAMO`.

For the objects segmentation, the package `rail_segmentation`[3], included in the ROS distribution, was used. This package provides tabletop object segmentation functionality for handheld objects given a point cloud. It also allows the segmentation within a robot's coordinate frame, so that objects stored on a robot's platform can be segmented. Under the hood, `rail_segmentation` use the PCL functions in order to segment obstacles. Computer vision routines used to segment obstacles allow to update the occupancy map, control collisions, and handle uncertainty in robot position and manipulation actions.

The package for objects manipulation (pushes and grasps) is provided by IAS-Lab[4] of University of Padova. This package is based on the MoveIt! Simple Grasps tool developed by Dave T. Coleman[5]. This is a simple grasps generator for simple objects for use with the MoveIt! pick and place tool. In this package, to move the end-effector to the object that has to be manipulated, the motion planning algorithm calculates a Cartesian path to the goal. Otherwise, the gripper is moved calculating the movement in joints space.

5.2.1. Map 2D

The `map_2d` package provides an implementation of a 2D costmap that receive information about the environment and builds a 2D occupancy grid of the data that memorize obstacles and their assigned weight.

In this package, an object is memorized through the `map_2d/Obstacle` that store its weight value. To create an obstacle, besides the assigned weight value, it is necessary to have the position with respect to the map and its footprint as polygon in the way of vector of points (`std::vector<geometry_msgs::Point>`). With this

[3]http://wiki.ros.org/rail_segmentation
[4]http://robotics.dei.unipd.it/
[5]https://github.com/davetcoleman/moveit_simple_grasps

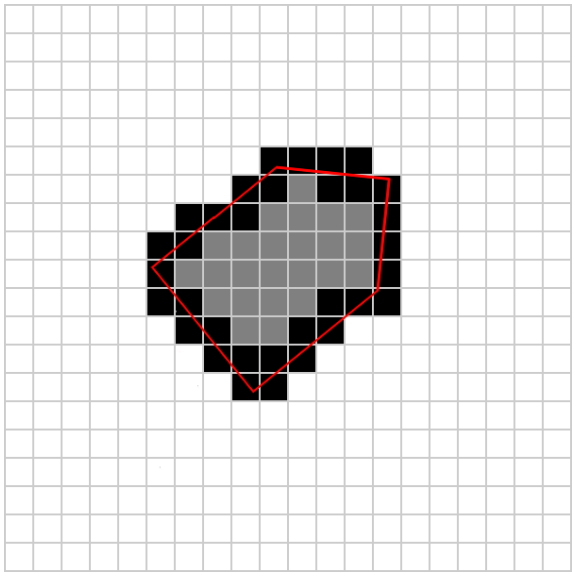


Figure 5.2.: The procedure of extraction of cells corresponding to the footprint.

information the cells occupied by the obstacle are calculated. A link to the obstacle is inserted in every cell that it occupies. This approach allows to distinguish between different obstacles inserted inside the map and know which cells occupy. Therefore, each cells of the grid can be empty or links to one or more obstacles, the unknown space is considered free. A cell has a weight value calculated by the sum of the weight of the obstacles that occupy that cell. If there are no obstacles, the space is considered free and the weight is 0.

`map_2d` provides a purely two dimensional interface, meaning that if two objects are in the same position in the XY plane, but with different Z positions would result linked by same cells, and that cells have the weight equals to the sum of the weight of objects. This approach is designed to help planning in planar spaces.

Footprint and cells

Now, an overview is given on how to find out cells corresponding to an obstacle footprint. As depicted in Figure 5.2, polygons corrispondent to the obstacles are positionated to their relative positions in the maps. Thanks to raytracing algorithm developer by Bresenhamit [1] the cells corresponding to edges of footprint are extracted. Bresenhamit's algorithm determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points and it is commonly used to draw line primitives in a bitmap image. When the boundary cells are extracted , the inner cells are also extracted.

Published Topics

- `costmap`(nav_msgs/OccupancyGrid)
  The values in the costmap, published every time the map is updated, and used only for visualization purposes.

Services

- `add_obstacles` (map_2d/Obstacle)
  Add a new obstacle in the map.

- `remove_obstacles` (std_msgs/String)
  Remove an obstacle present in the map by its ID.

5.2.2. NAMO Planner

The NAMO Planner (`namo_planner`) package provides an implementation of the proposed algorithm for Navigation Among Movable Obstacles. This package is fairly simple. It receives information about obstacles present in the word that are memorized in the costmap generated with the `map_2d` package. Then, the algorithm is able to find the path to reach the goal pose and select which obstacles to move.

In order to find the best orientation of the robot that allows, if it is possible, to pass through the free space while avoiding collisions, at each change of cell, the footprint is rotated by 360 degrees with step of 45 degrees. This approach allows to find a collision free path if it exist and, of course, it is better than moving an obstacle.

The collision check is done by verifying if one or more cells corresponding to the robot footprint are occupied by an obstacle. If an obstacle is detected, it begins to be considered for the displacement.

Published Topics

- `plan` (nav_msgs/Path)
  The last plan computed, published every time the planner computes a new path, and used primarily for visualization purposes.

Subscribed Topics

- `map` (nav_msgs/OccupancyGrid)
  2D costmap originating from `map_2d`. A map change will trigger replanning.

- `initialpose` (geometry_msgs/PoseWithCovarianceStamped)
  Current starting pose (x, y, theta) in the map frame.

- `goal` (geometry_msgs/PoseStamped)
  Goal pose (x, y, theta) in the map frame.

Services

- `plan_namo` (nav_msgs/GetPlan)
  Plan a path from a given start to goal, return the success.

6. Experiments

In order to evaluate the effectiveness of the solution proposed in Chapter 4, an extensive phase of tests was executed. The first series of tests on the proposed algorithm has been focused on a point-like robot. As previously explained, the robot states space is projected into an Euclidean space. Results of these tests allow to evaluate the effectiveness of the proposed solution, in particular the weight of the moved obstacles, the computational time and the scalability of the algorithm. After proving the effectiveness of the proposed solution with the point-like robot model, the following tests have been focused on a real robot model. More precisely was tested the projection of the footprint of the robot into the two-dimensional plane. All tests were performed with full known environment. Anyway, the algorithm works on a full reconfigurable environment and this allows an easily extension to an unknown environment. The workbench for all performed experiments is a MacBook Pro equipped with an Intel Core i7 2.2GHz quad-core processor and 16GB DDR3 RAM memory.

In the end a simulated execution was performed to prove the validity of the proposed solution in a real simulated scenario with a real simulated robot. It was decided to use a robot that combines motion capabilities of a unmanned ground vehicle with manipulation capabilities of a robotic arm.

6.1. Point-Like Robot

First experiments were performed on a point-like robot model. In order to evaluate the effectiveness of the proposed solution, two different versions of the algorithm were tested: the proposed one and its random version. As stated in Chapter 4, the proposed version of the algorithm adds the Euclidean distance, other than the objects weights, to the formulation of the importance. In this solution is evident the combination of A* and KPIECE algorithms. On the other hand, the random version does not consider the Euclidean distance while evaluating the importance of the neighbors of a cell. It randomly selects a neighbor to be added to the tree for the expansion.

Goals of these experiments are:

- proving the timing improvement achieved by the proposed algorithm;

- showing its independence from the size of the map and from the number of obstacles;
- proving that lighter objects are preferred for moving.

6.1.1. Experimental Setup

To reach the goals of this experiment, two different type of tests were made. The first one tests the computational time of the algorithm and it proves the correct choice of the objects to be moved.

In the first setup, two different types of maps were created. The first map was populated by 70 obstacles, than again, the second one was populated by 100 obstacles. Secondly, every map was discretized into a 25x25 and a 50x50 grid. For every configuration, (depending on the size of the grid and the number of obstacles) 100 different maps were randomly generated. Tests were repeated 100 timers for each generated map, creating a sample of 10000 tests for every proposed configuration.

In the second setup, a different approach was used to test how time changes varying the number of the obstacles or the size of the map. Firstly, a map with 50×50 grid was fixed. The number of obstacles populating it varied from 10 to 250. Secondly, the number of the obstacles was fixed to 70 and the size of the grid was varied from 20×20 to 100×100.

The two different setups, had in common that the obstacles were polygons of random random size, randomly placed on the map and with weight randomly assigned. For simplicity, three different weights $[1, 3, 5]$ were considered.

6.1.2. Results

Table 6.1 and Table 6.2 collect the results of the previously explained test, more specifically those performed with setup 1.

Table 6.1 proves that the elaboration time is not influenced by the number of obstacles. The implemented version is faster than the random one and less affected by an increase of the grid size.

On the other hand, Table 6.2 shows that, on average, only obstacles with little weight are chosen. In addition, the implemented version selects less obstacles than the random one. The choice shows the ability of the robot to select those objects whose displacement requires less effort.

Figure 6.2 and 6.1 depict results obtained performing tests with setup 2. As shown, time increase linearly both in random and proposed execution, in both cases, time follow the growth of the number of obstacles but, as depicted in Figure 6.1, the random execution time increases faster than the other one.

The chart of Figure 6.2 shows that the random algorithm time increases exponentially with respect to a change of the grid size. On the other side, the implemented

Table 6.1.: Mean time of 10000 executions (time expressed in ms).

Obstacles	25x25 deterministic	25x25 random	50x50 deterministic	50x50 random
70	0.265	1.021	0.389	3.742
100	0.257	1.350	0.380	3.590

Table 6.2.: Mean obstacles weight of 10000 executions.

Obstacles	25x25 deterministic	25x25 random	50x50 deterministic	50x50 random
70	1.056	1.278	1.000	1.333
100	1.126	1.280	1.000	1.306

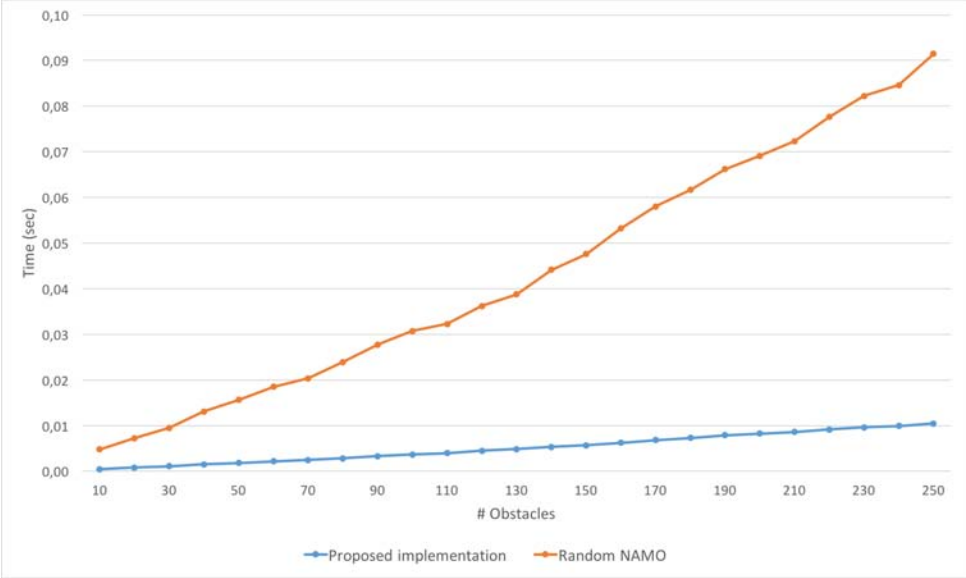


Figure 6.1.: Computational time with respect to the number of obstacles populating the grid.

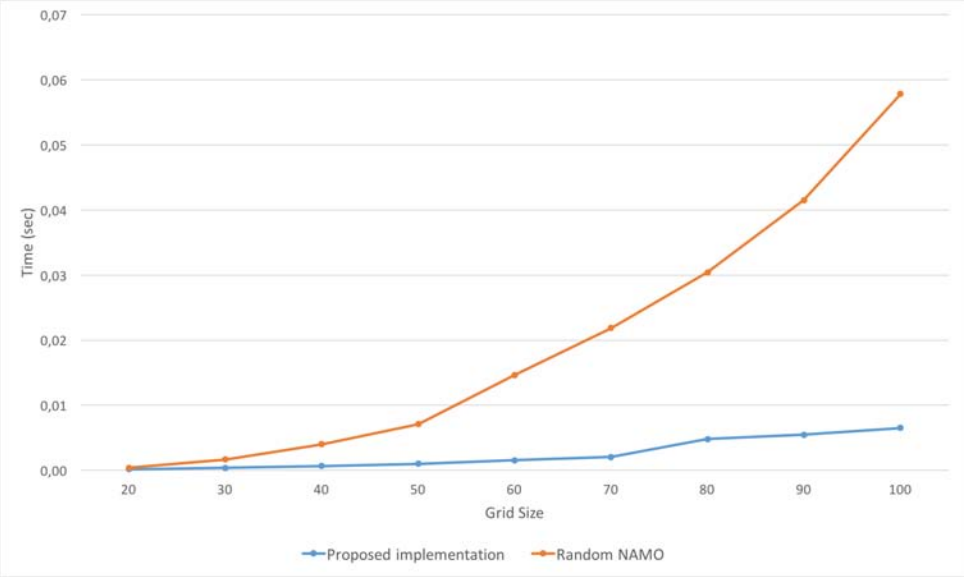


Figure 6.2.: Computational time with respect to the size of the map.

algorithm time increases more slowly and linearly. In summary, these charts prove that the algorithm is independent from the size of the map and the number, weight, size and location of obstacles. In fact, as stated in Subsection 6.1.1, weights are assigned randomly to obstacles and obstacles are located randomly on the map. Beyond tests results, Figure 6.3 depicts the working of the algorithm. The figures show a point-like robot performing a path from a Start state to a Goal while adopting the proposed methodology. As it is possible to see, one light obstacle has been selected for the relocation.

6.2. Point-Like Robot with Footprint

After really good results obtained with the point like robot, another experiment was performed taking in consideration the footprint of the robot. The footprint is the two-dimensional projection of the robot to the ground plane. This experiments aims to confirm the effectiveness of the proposed solution also considering the robot occupation. Furthermore, it aims to verify the accuracy of the algorithm in a simple but possible real scenario.

6.2.1. Experimental Setup

For the purpose of this experiment a simple map with three obstacles was generated. As depicted in Figure 6.4, two heavy obstacles, like tables, were placed in the opposite sides and a small lightweight object, like a small box, was placed in the middle. The

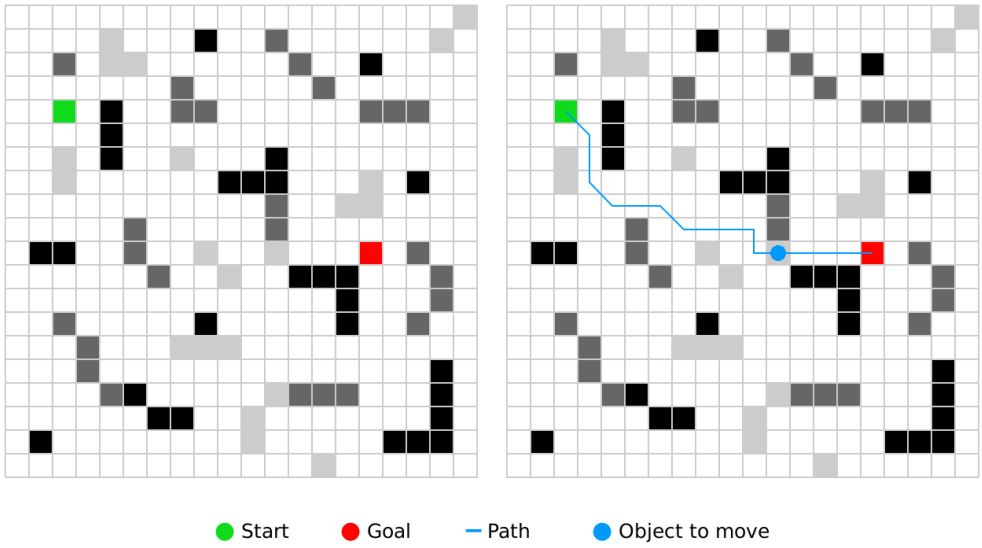


Figure 6.3.: The path connecting a Start state to the Goal state: one obstacle selected for the relocation.

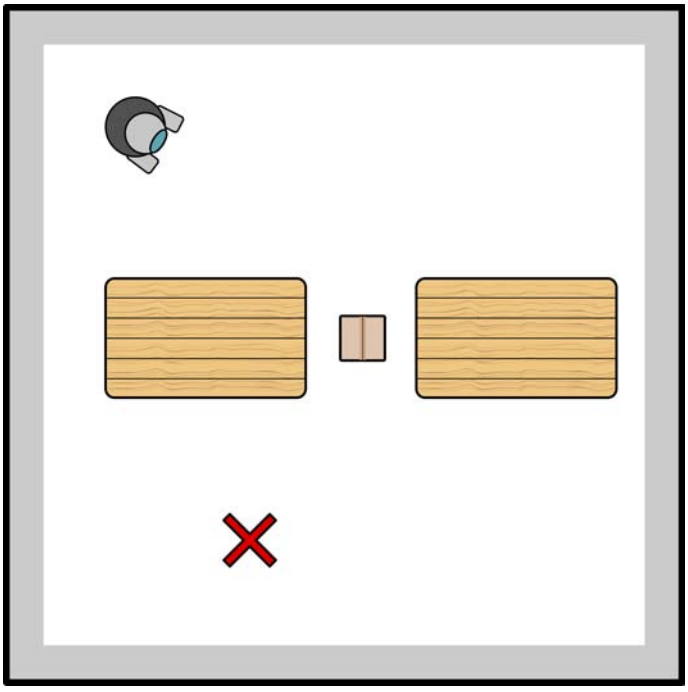


Figure 6.4.: The environment proposed for the experiment with the footprint of the robot.

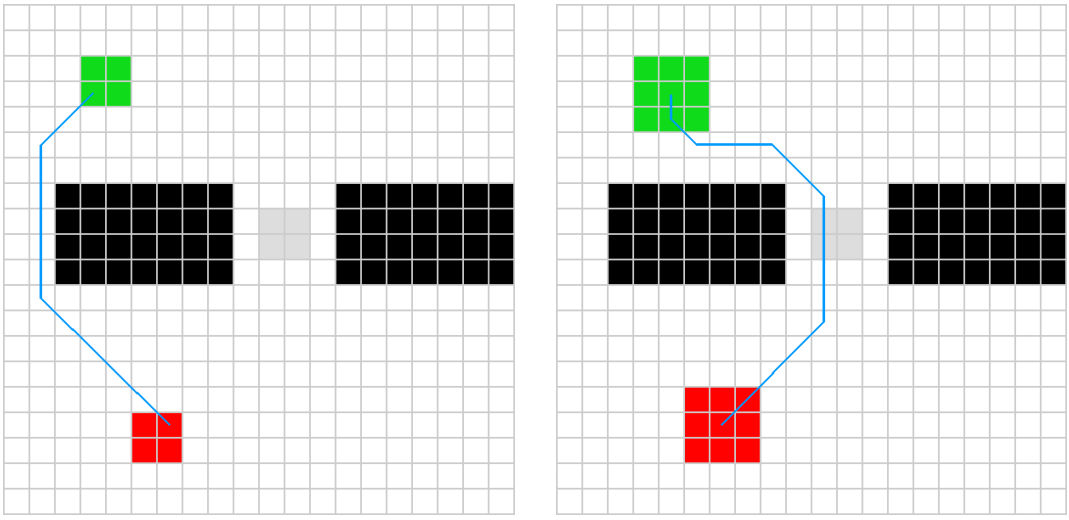


Figure 6.5.: An example of execution with Small Robot and Big Robot. In figures are visible the start footprint in green and the goal footprint in red.

robot, starting from the start position, has to reach the goal position through the free space or moving the box. Two robots of different size were used: a small ones that can obstacles and reach the goal passing through free space and a bigger ones that can not avoid obstacles. The experiment was executed 10 times changing the start or the goal position using both two robots. In Figure 6.5 it is possible to see an example of execution with the Small Robot and another one with the Big Robot.

6.2.2. Results

Results of the test are collected in Table 6.3. For each exectution it were collected the computation time and if the small obstacle was removed or not. As it is possible to see by the mean of the results the computation time is around 20 milliseconds, slightly higher with the Big Robot. Compared to the result obtained without consider the footprint of the robot, computational time is an order of magnitude higher. About the small obstacle, Big Robot obviously removes it every time, on the other hand, Small Robot removes it in the 40% of the case

6.3. Simulated Robot

Once proved the effectiveness of the algorithm for a point-like robot, tests were performed on a simulated one.

The robot with which tests were performed is composed of an Husky mobile robot with an UR5 robot manipulator, a Robotiq 2-Finger gripper, and a Microsoft Kinect vision sensor. A deep analysis of this robot will be given later in this Section.

Table 6.3.: The results obtained in 10 test considering the footprint of the robot. Time is expressed in seconds.

	Small Robot		Big Robot	
	Time	Removed	Time	Removed
	0.0258952	Y	0.0267104	Y
	0.0206904	Y	0.0236292	Y
	0.0213171	N	0.0252328	Y
	0.0217194	N	0.0280514	Y
	0.0199351	N	0.0311342	Y
	0.0200567	Y	0.0230287	Y
	0.0182324	Y	0.0226662	Y
	0.0168608	N	0.0261195	Y
	0.0209737	N	0.0286647	Y
	0.0203648	N	0.0267853	Y
Mean	0.0206046	40%	0.0262022	100%

In order to execute the simulation experiment, Gazebo was used as simulator [19]. For the objects manipulation it was used MoveIt!

6.3.1. The Robot

In order to simulate the behavior of the algorithm in a real scenario it was decided to use a robot that combine motion capabilities of a unmanned ground vehicle with the manipulation capabilities of a robotic arm. The choice, as depicted in Figure 6.6, is an Husky that interact with the world around it with a UR5 robot arm from Universal Robots mounted to the Husky top plate and a 2 Finger Gripper from Robotiq. The arm can extend up to 0.85m and carry a 5kg payload and safe around humans. A Microsoft Kinect vision sensor was added for object segmentation and to correct uncertainty in manipulation operations. Another strong point come out in favor of this choice is that Husky, UR5 and Robotiq gripper and Microsoft Kinect are fully supported in ROS.

This robot is generally utilized in:

- outdoor autonomous navigation;
- remote inspection and long distance tele-operation;
- larger scale mapping and localization.

Figure 6.6.: The utilized robot. Husky UGV, UR5 robot arm, 2 Finger Gripper and Microsoft Kinect

Clearpath Robotics Husky Husky[1] (see Figure 6.7) is a rugged, outdoor-ready unmanned ground vehicle (UGV), suitable for research and rapid prototyping applications. Its large payload capacity and power systems allow a very wide variety of customization to meet research needs. In fact, Stereo cameras, LIDAR, GPS, IMUs, manipulators and more can be added to this UGV. Moreover, Husky is fully supported in ROS with community driven Open Source code and examples.

Universal Robots UR5 UR5 from Universal Robots[2] is a lightweight, flexible and collaborative industrial robot from Universal Robots (see Figure 6.8). The UR5 is a six-jointed robotic arm with a very low weight that can extend up to 0.85m and carry a 5kg payload. Universal Robots are Collaborative Robots which means that they can work right alongside people without safety guarding. This features make this robot ideal to low-weight collaborative processes, such as: picking, placing and testing.

[1] http://www.clearpathrobotics.com/
[2] http://www.universal-robots.com/

Figure 6.7.: Clearpath Robotics Husky

Figure 6.8.: Universal Robots UR5

Robotiq 2-Finger Robotiq[3] 2 Finger (see Figure 6.9) is a gripper that, as its name says, has two articulated fingers and it is compatible with all major industrial robots. This gripper can engage up to five points of contact with objects (two on each of the phalanges plus the palm). The fingers are under-actuated, meaning they have fewer motors than the total number of joints. This configuration allows the fingers to automatically adapt to the shape of the object they grip and it also simplifies the control of the gripper.

Microsoft Kinect The Kinect sensor is a flat, black box that sits on a small platform when placed on a table or shelf near the television used with Xbox 360. This device has the following three sensors that can be used for vision and robotics tasks:

- a color VGA video camera to see the world in color;
- a depth sensor, which is an infrared projector and a monochrome CMOS sensor working together, to see objects in 3D;

[3]http://www.robotiq.com/

Figure 6.9.: Robotiq 2-Finger 85

Figure 6.10.: Microsoft Kinect 360

- a multiarray microphone that is used to isolate the voices of the players from the noise in the room.

For the scope of this project, it is used only two of these sensors: the RGB camera and the depth sensor.

6.3.2. Tools

Gazebo Robot simulator is an essential tool. Gazebo[4] is a multirobot simulator for complex indoor and outdoor environments. It is capable of simulating a population of robots, sensors, and objects in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects, included an accurate simulation of rigid-body physics. Gazebo is composed of a robust physics engine (ODE), a high-quality rendering engine (OGRE), and a graphical interfaces. Two threads run in Gazebo. The first manages the GUI and rendering engine, and the second thread manages the physics engine.

The worlds rendered in gazebo are described by a XML document where you can use external 3D objects (e.g. meshes). The ROS version of gazebo provides some

[4]http://www.gazebosim.org/

key-features:

- ROS service for loading robot models defined using the URDF language;
- Services whose permit to move directly joints and links resident in the gazebo simulating.

MoveIt! MoveIt![5] is state of the art open-source software for mobile manipulation in ROS. The library incorporates a fast inverse kinematics solver (as part of the motion planning primitives), state-of-the-art algorithms for manipulation, grasping 3D perception (usually in the form of point clouds), kinematics, control, and navigation. Apart from the backend, it provides an easy-to-use GUI to configure new robotic arms with the MoveIt! and RViz plugins to develop motion planning tasks in an intuitive way. MoveIt! can be easily integrated in robotics products for industrial, commercial, research purpose.

Point Cloud Library (PCL) The Point Cloud Library[6] (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing.

Point clouds appeared in the robotics toolbox as a way to intuitively represent and manipulate the information provided by 3D sensors, such as time-of-flight cameras and laser scanners, in which the space is sampled in a finite set of points in a 3D frame of reference. The Point Cloud Library (PCL) provides a number of data structures to easily represent the points of the sampled space. PCL also provides a number of state-of-the-art algorithms to perform data processing, such as filtering, model estimation, surface reconstruction, and segmentation.

ROS provides a message-based interface through which PCL point clouds can be efficiently communicated, and a set of conversion functions from native PCL types to ROS messages. Aside from the standard capabilities of the ROS API, there are a number of standard packages that can be used to interact with common 3D sensors, such as the widely used Microsoft Kinect or the Hokuyo laser, and visualize the data in different reference frames with RViz.

6.3.3. Simulated Execution

The simulation aims to simulate a possible environment in which the robot has to remove a small object to reach quickly the goal. As depicted in Figure 6.11 the presented scenario, similar to the scenario utilized for the experiment in Section 6.2, is formed by two big and heavy objects and a small can placed in the middle. The robot, placed in the start position has to reach the goal placed beyond the can, choosing if moving the can or avoiding obstacles.

[5] http://moveit.ros.org/
[6] http://www.pointclouds.org

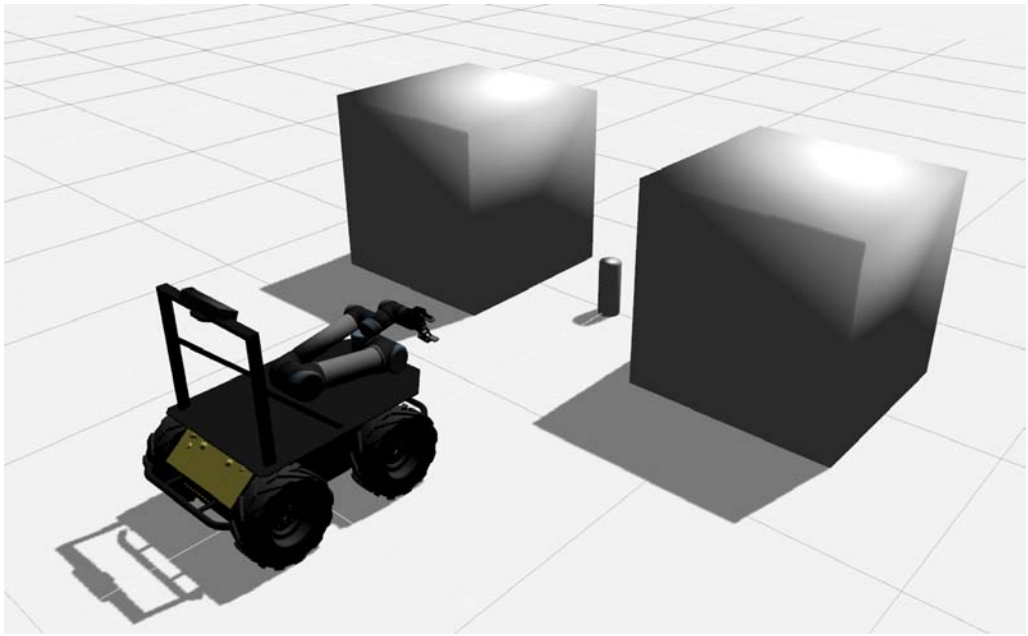


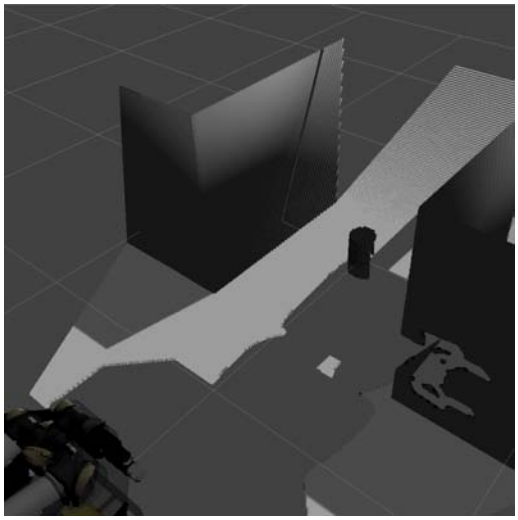
Figure 6.11.: Simulated environment

The software segments obstacles in the scene using the `rail_segmentation` package and assigns a weight value to each one. The footprint with the segmented object is calculated and the costmap, created through the `map_2d` package, is populated. The information about the footprint of the robot is passed to the software as a configuration parameter.

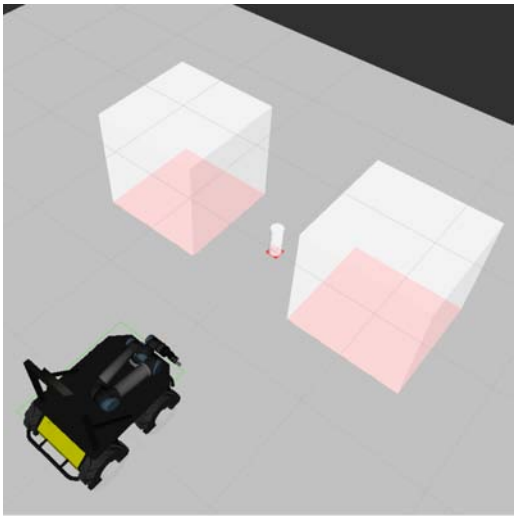
The `namo_planner` package, containing the implementation of the proposed algorithm, is used to find the path that the robot has to follow. As it is possible to see in Figure 6.12c the algorithm chose to remove the obstacle to reach the goal. Indeed this is the shortest way.

The robot, as is visible in Figure 6.12d, follows the calculated path until a distance useful for the manipulation of the can. Then it removes the can from the path using the grasping technique exploiting functions implemented in MoveIt! (see Section 4.2).

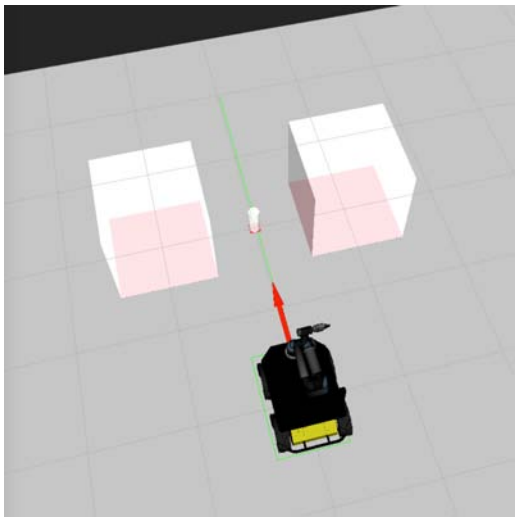
In the end, the robot continues to follow the path until it reaches the goal.



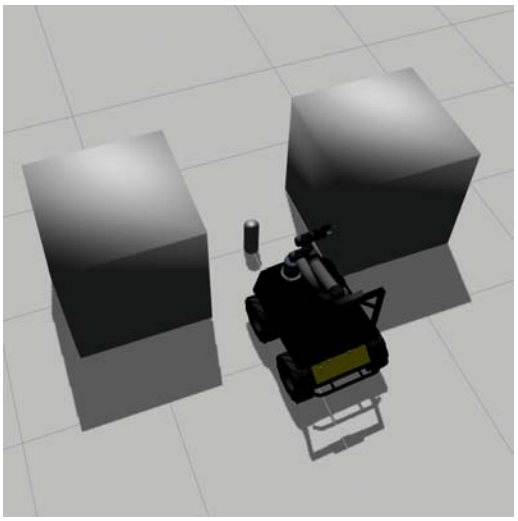
(a) Segmentation



(b) Map creation



(c) Path planning



(d) Execution

Figure 6.12.: Some moments during the simulation execution.

7. Conclusions and Future Work

This thesis presented a NAMO solver combining the existing A*-based NAMO algorithms with the exploration strategy of KPIECE. The obtained algorithm addresses a primary NAMO challenge: scalability. It is scalable because of its independence from the size of the map and from the number, shape, and pose of obstacles. Moreover, it does not impose restrictions on actions to be performed: the robot can both push and grasp every object. It works on a two-dimensional projection of a three-dimensional cluttered workspace letting consider both objects in contact and not in contact with the ground. It assumes full world knowledge but the environment is completely reconfigurable and the algorithms can be easily extended in order to solve NAMO problems in unknown environments. In fact, it is able to handle sensor feedbacks and correct uncertainties regarding obstacle poses and robot actions. Performed tests, discussed in Chapter 6, prove that the solution shown above is locally optimal. Simulations demonstrated that implemented algorithm is ready for being used with a real robot. ROS-compliant developed packages are public released as open-source and available at `https://bitbucket.org/account/user/iaslab-unipd/projects/NAMO`.

7.1. Future Work

With respect to the results obtained, a future goal is to switch from a reconfigurable to an unknown environment. In this way robots are allowed to operate in unknown spaces that is closely to a real scenario; an example could be that robots become helpful in rescue operation. A successive step is to generalize concepts and move from the *Navigation Among Movable Obstacles* (NAMO) domain to the *Navigation Using Manipulable Objects* (NUMO) domain. In NUMO domain, the robot is not restricted to just the moving of environment objects to clear a path, but it can also use them as tools to create a path. For these reasons, NAMO and its generalization NUMO allow to create cognitive robots that can operate autonomously in every scenario and explore unknown regions. Robots understand the world around and find a local optimal solution to achieve the assigned task.

When dealing with unknown objects, a predetermined gripper pose is not available. A solution can be that of exploiting the *Reinforcement Learning* techniques in order to let robots manipulate unknown objects of any shape. An ontology is being formulating allowing the storage of the information necessary for the manipulation. Its Cloud sharing will speed up the robots ability to manipulate objects thanks to the

combined exploitation of their prior knowledge and of the expertise of other robots.

In conclusion, the future goal is to create a system that allows robots to autonomously navigate the world in order to complete an assigned task. Following this idea, robots will be able to recognize objects and obstacles and autonomously take decisions on what is the local optimal way to take. Moreover, robots will be able to learn how to manipulate new objects and memorize this information for the future or to share them with other robots.

A. A*

A* [4] is an informed search algorithm or a best-first search algorithm, this means that it solves problems by searching among all possible paths to the goal for the one that take the smallest cost (i.e. least distance travelled, shortest time, etc.); among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a start node S of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node G.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. This operation is done by using an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n) \tag{A.1}$$

where:

- n is the last node on the path;
- $g(n)$ is the cost of the path from the start node to n;
- $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal.

The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum estimated cost nodes to expand. This priority queue is known as the open set. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue, or until the queue is empty. The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described gives only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node

will point to its predecessor, and so on, until some predecessor of the node is the start node.

As an example, when searching for the shortest route on a map, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points.

If the heuristic h satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently and it is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) - h(x)$.

Additionally, if the heuristic is monotonic (or consistent), a closed set of nodes already traversed may be used to make the search more efficient.

Algorithm 4 A*

Input: start, goal
Output: path

```
1: open ← start
2: closed ← ∅
3: while open ≠ ∅ do
4:     current ← the node in open having the lowest f value
5:     if current = goal then
6:         return reverse path from goal to start
7:     end if
8:     closed ← current
9:     for all neighbor of current ∉ closed do
10:        new_g_score ← current.g_score + dist_between(current, neighbor)
11:        if neighbor ∉ open then
12:            Add neighbor to open
13:        else if g(current) + heuristic(current, neighbor) < g(neighbor) then
14:            Update neighbor's backpointer to point to current
15:        end if
16:    end for
17: end while
18: return failure
```

B. KPIECE

KPIECE, or *Kynodinamic motion Planning by Interior-Exterior Cell Exploration*, is a tree-based planner that uses a discretization to guide the exploration of the state space. It iteratively constructs a tree that connect the start state S to the goal G. Moreover, KPIECE prefers less explored areas of the state space.

In fact, KPIECE discretizes the whole state space by a grid of cells all having the same size. Initially, the grid is populated by a single cell placed in correspondence of the initial state. Then, in turn, the necessary cells are allocated. Each cell includes a list of states, initially empty, in which whole visited states belonging to the cell are insert. This makes possible to identify areas highly and poorly explored (see Figure B.1).

In particular, cells of the grid can be divided into two types: INTERIOR and EXTERIOR. The INTERIOR ones are cells that have $2n$ neighboring cells, where n is the size of the discretization space; they identify areas where the exploration has already occurred. The other ones are the EXTERIOR cells.

Algorithm 5 KPIECE

```
1: Create a grid G = Cell(start)
2: Create a list Cell(start).list = start
3: while time < TIME_MAX do
4:     cell ← select a cell EXTERIOR or INTERIOR
5:     q_old ← select a cell's already visited state
6:     Sample a new state q_rand with probability P such that ditance(q_old, q_rand) ≤ ε
7:     if (q_old, q_rand) is valid then
8:         Add Cell(q_rand) to the grid G
9:         Add q_rand to the list Cell(start).list
10:        Save the path between q_old and q_rand in the solution tree
11:        if q_rand = q_old then
12:            return the path between start and goal
13:        end if
14:    end if
15: end while
```

The algorithm prefers the expansion in less explored areas so, when selecting the cell to be expanded, it prefers to start the expansion from EXTERIOR cells.

Determined the type of the cell to be expanded, the actual cell is selected according to the formula:

$$Importance(i) = \frac{\log_{10}(I) * score}{S * N * C} \tag{B.1}$$

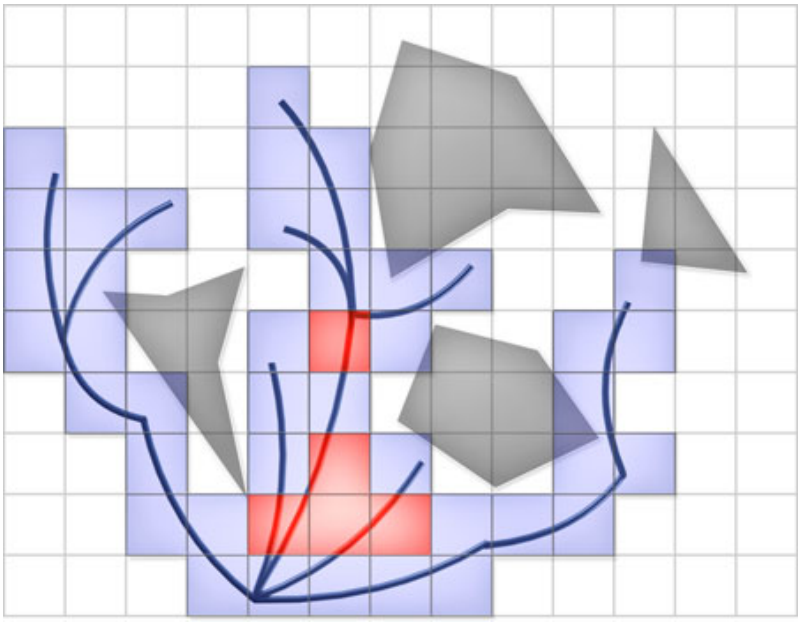


Figure B.1.: KPIECE

where:

- I is the iteration in which the cell i was created
- *score* is the estimated distance to the goal;
- S is the number of times in which the cell i was selected for expansion;
- N is the number of instantiated neighboring cells;
- C is the measure of coverage for the cell i (the sum of its visited states).

Given a cell, a state q_{old} is extracted from the already visited ones and stored inside the relative list according to an half normal distribution. From q_{old}, the tree's expansion starts. A new state q_{rand} is optained with probability with a certain probability P placed to a distance less than ε by q_{old}. The edge that joins the two states is evaluated. If it's valid:

- if necessary the cell containing q_{rand} is added to the grid;
- q_{rand} is added to the newly created cell;
- the search tree is updated by adding the new edge.

Otherwise:

- the last valid state q_{new} belonging to the edge is extracted;

- it is checked that its value is greater than a threshold;
- if so $q_{rand} = q_{new}$.

After adding the edge, it is checked if the goal is reached, otherwise the process iterates. The probabilistic component affects a lot the correctness of the algorithm. It is present in selection of:

- the type of cell;
- the state belonging to a cell;
- a new state q_{rand}.

Bibliography

[1] Jack E Bresenham. "Algorithm for computer control of a digital plotter". In: *IBM Systems journal* 4.1 (1965), pp. 25–30.

[2] Nicola Castaman, Elisa Tosello, and Enrico Pagello. "A Sampling-Based Tree Planner for Navigation Among Movable Obstacles". In: *47th International Symposium on Robotics (ISR 2016)*. Munich, Germany, June 2016.

[3] Erik D. Demaine, Joseph O'Rourke, and Martin L. Demaine. "PushPush and Push-1 are NP-hard in 2D". In: *In Proceedings of the 12th Canadian Conference on Computational Geometry*. 2000, pp. 211–219.

[4] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pp. 100–107.

[5] David Hsu et al. "Randomized kinodynamic motion planning with moving obstacles". In: *The International Journal of Robotics Research* 21.3 (2002), pp. 233–255.

[6] Yohei Kakiuchi et al. "Working with movable obstacles using on-line environment perception reconstruction using active sensing and color range sensor". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. 2010, pp. 1696–1701.

[7] Lydia E Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *Robotics and Automation, IEEE Transactions on* 12.4 (1996), pp. 566–580.

[8] James J Kuffner and Steven M LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 2. IEEE. 2000, pp. 995–1001.

[9] Jean-Claude Latombe. *Robot motion planning*. Springer, 1991.

[10] Steven M. LaValle. *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[11] Martin Levihn. "Autonomous environment manipulation to facilitate task completion". PhD thesis. Georgia Institute of Technology, 2015.

[12] Martin Levihn, Jason Scholz, and Mike Stilman. "Planning with movable obstacles in continuous environments with uncertain dynamics". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 3832–3838.

[13] Martin Levihn, Jonathan Scholz, and Mike Stilman. "Hierarchical Decision Theoretic Planning for Navigation Among Movable Obstacles". In: *Proceedings of the Tenth International Workshop on the Algorithmic Foundations of Robotics (WAFR 2012)*. 2012, pp. 13–15.

[14] Martin Levihn, Mike Stilman, and Henrik Christensen. "Locally optimal navigation among movable obstacles in unknown environments". In: *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. IEEE. 2014, pp. 86–91.

[15] Kevin M Lynch and Matthew T Mason. "Stable pushing: Mechanics, controllability, and planning". In: *The International Journal of Robotics Research* 15.6 (1996), pp. 533–556.

[16] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. 2009, p. 5.

[17] Radu Bogdan Rusu and Steve Cousins. "3d is here: Point cloud library (pcl)". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 1–4.

[18] Gildardo Sánchez and Jean-Claude Latombe. "A single-query bi-directional probabilistic roadmap planner with lazy collision checking". In: *Robotics Research*. Springer, 2003, pp. 403–417.

[19] Anthony Stentz. *The D* Algorithm for Real-Time Planning of Optimal Traverses*. Tech. rep. DTIC Document, 1994.

[20] Mike Stilman. "Navigation among movable obstacles". PhD thesis. Carnegie Mellon University, 2007.

[21] Mike Stilman and James Kuffner. "Navigation among movable obstacles: real-time reasoning in complex environments". In: *Humanoid Robots, 2004 4th IEEE/RAS International Conference on*. Vol. 1. IEEE. 2004, pp. 322–341.

[22] Mike Stilman and James Kuffner. "Planning among movable obstacles with artificial constraints". In: *The International Journal of Robotics Research* 27.11-12 (2008), pp. 1295–1307.

[23] Mike Stilman et al. "Planning and Executing Navigation Among Movable Obstacles". In: *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE. 2006, pp. 820–826.

[24] Ioan A. Şucan and Lydia E. Kavraki. "A sampling-based tree planner for systems with complex dynamics". In: *Robotics, IEEE Transactions on* 28.1 (2012), pp. 116–131.

[25] Ioan A. Şucan and Lydia E. Kavraki. "Kinodynamic Motion Planning by Interior-Exterior Cell Exploration". In: *Algorithmic Foundation of Robotics VIII*. Springer, 2010, pp. 449–464.

[26] Jur Van Den Berg et al. "Path planning among movable obstacles: a probabilistically complete approach". In: *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 599–614.

[27] Gordon Wilfong. "Motion planning in the presence of movable obstacles". In: *Proceedings of the fourth annual symposium on Computational geometry*. ACM. 1988, pp. 279–288.

[28] Hai-ning Wu, Martin Levihn, and Mike Stilman. "Navigation among movable obstacles in unknown environments". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. Oct. 2010, pp. 1433–1438.

Acknowledgments

I am grateful to my supervisor, professor Enrico Pagello, for the passion transmitted and for the opportunity that he gives to me by working in the marvelous world of robotics.

Huge thanks to my co-supervisor Elisa Tosello, for her support and guidance in the development of the work presented in this thesis. Elisa introduced me to motion planning and inspire me to pursue the work on NAMO.

I am grateful to all the people in the IAS-Lab that took their time to help me with my exams and my works. In particular I want say thank to Stefano Ghidoni that is a mentor for me, and he always finds time to give me a good advice.

Thanks to EuRoC team for the fantastic experience in Stuttgart.

I would like to thanks my parents Katia and Giovanni for supporting me in these not always simple years of studies, and to my grandparents for raising me.

Thanks to my oldest friends Mirko and Martina that were always there for me an spur me in there years, and to all other my friends.

I want to thank to all the people that i meet over these years at university and in particular to Matteo with which I share a lot of studying days and also the apartment in Padova.

Finally, I thank the girl that turned my world upside down. Elisa, my girlfriend, that has always been close to me during the work on this thesis and that try to understand my studies in robotics.