POLITECNICO DI MILANO
School of Industrial and Information Engineering
Master of Science in Automation and Control Engineering

Implementation, comparison, and advances in global planners using Ackerman motion primitives

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Supervisor: Prof. Matteo Matteucci
Co-Supervisor: Prof. Luca Bascetta

Thesis by:
Carolina Arauz Villegas
ID 862405

Academic Year 2017-2018

Contents

List of Figures

List of Tables

Acronyms

A*	A* Algorithm
AD*	Anytime Dynamic A* Algorithm
ANA*	Anytime Nonparametric A* Algorithm
ARA*	Anytime Repairing A* Algorithm
OMPL	Open Motion Planning Library
PRM	Probabilistic Roadmap Algorithm
PRM*	Optimal Probabilistic Roadmap Algorithm
ROS	Robot Operating System
RRT	Rapidly-Exploring Random Tree Algorithm
RRT*	Optimal Rapidly-Exploring Random Tree Algorithm
SBPL	Search Based Planning Library

Abstract

Motion planning plays an important role in autonomous navigation, it can be defined as the search for a collision-free path from an initial state to a final state.
A good path planner, specially when dealing with non-holonomic vehicles, should also take into considerations the capabilities of the vehicle (kinematic and dynamic constraints), so that the path is not only collision-free, but feasible for the robot.
Path planning has been generally divided in two parts, a global path planning and a local path planning, both parts are complementary and in most cases used together.

This thesis is focused on the global planning problem for non-holonomic vehicles in an (x, y, θ) space, with the by product of implementing novel global planners in the Robot Operating System (ROS) framework, nowadays, widely used in the robotics community.

To tackle our goal, two ROS nodes are created, the first uses the search-based algorithms implementations by SBPL (Search Based Planning Library [3]). The second uses random sampling planners implementations provided by OMPL (Open Motion Planning Library [1]).

To deal with the non-holonomic constraints of the vehicle, motion primitives can be used; which for the SBPL environment has already support to access motion primitives.
In the case of OMPl planners, even though spaces like Dubins and Reeds-Shepp exist, the use of motion primitives is not given. Therefore an implementation of RRT* with Motion Primitives done by Basak Sakcak [4] was modified, to work with the same motion primitives used in the SBPL approach. Then, three new methods of setting a sample set were created to improve the performance of the planner, and achieve real-time performance.
Finally, performance comparisons were done between the different planners in terms of length of the path and planning time on two real world use cases.

Sommario

La pianificazione del movimento svolge un ruolo importante nella navigazione autonoma, può essere definita come la ricerca de un percorso senza collisioni da uno stato iniziale a uno stato finale.
Un buon pianificatore di percorsi, specialmente quando si tratta di veicoli non-olonomici, dovrebbe anche prendere in considerazione le caratteristiche del veicolo (vincoli cinematici e dinamici), in modo che il percorso sia percorribile per il robot.
La pianificazione del percorso viene generalmente suddivisa in due parti, una pianificazione globale e una locale, che sono complementari e nella maggior parte dei casi utilizzate insieme.

L'obiettivo principale della tesi è lo studio di pianificatori globali per veicoli non-olonomici nello spazio (x, y, θ).
Per affrontare il nostro obiettivo, abbiamo creato due pianificatori, il primo, che utilizza gli algoritmi di ricerca su grafo, basato sulla libreria SBPL (Search Based Planning Library [3]). Il secondo sfrutta la libreria OMPL (Open Motion Planning Library [1]) che implementa algoritmi basati sul campionamento.
Per gestire i vincoli non-olonomici del veicolo, è possibile utilizzare le primitive di movimento.Nel caso di SBPL l'uso delle primitive è supportato mentre nel caso dei pianificatori OMPL, anche se esistono spazi come Dubins e Reeds-Shepp, l'uso delle primitive di movimento non è previsto. L'implementazione di RRT* che sfrutta le primitive di movimento realizzata da Basak Sakcak[4] è stata usata come punto di partenza per la tesi, e modificata per lavorare con le stesse primitive di movimento utilizzate nell'approccio con SBPL e ottenere prestazioni real-time.

I risultati presentano il confronto delle prestazioni tra i diversi pianificatori in termini di lunghezza del percorso e tempo di pianificazione.

Acknowledgements

First, I must express my very profound gratitude to my parents and my sisters, mostly for their emotional support during this two years abroad, without them this accomplishment would not have been possible.

To my fiancé Pablo, for always been there for me and for encouraging me during this adventure by his side. To all the new friends for introducing me to their culture and for making these two years an amazing experience. To my friends back home for being there no matter the distance.

Last, but not least I would also like to thank Prof. Matteo Matteucci, first for giving me the opportunity of the thesis, but specially for the challenge, to keep pushing me to learn more. As well as Prof. Luca Bascetta for accepting to be my co-supervisor. To Gianluca Bardaro, Basak Sacak and Alessandro Gabrielli for all the support during the development of the thesis, for their availability to answer my questions and doubts.

A mi familia, por su esfuerzo y apoyo incondicional

Chapter 1

Introduction

"Success is the ability to move from one failure to another without loss of enthusiasm."

Winston Churchill

Mobile robot path planning is a key research area. An autonomous robot should have the capability of generating a collision free path between a state A and a state B in its environment

The path planning problem can be formulated as follows: given a mobile robot and a model of the environment, find the optimal path between a start position and a final position without colliding with obstacles. The constructed path must satisfy a set of optimization criteria including processing time, energy consumption, and shortest traveled distance.

Path planning is influenced by two factors the environment and the knowledge the robot has of its environment, therefore, it has been generally divided in two parts, a global path planning and a local path planning. Both parts are complementary and in most cases used together. The global planner requires prior knowledge of the environment and the constraints of the robot to create the plan, the local planner, also called trajectory follower, generates the control values to follow the path generated by the global planner taking into account the current state of the vehicle as perceived by the sensors.

For autonomous vehicles path planning is quite important since the vehicle has limited capabilities and it can only handle certain maneuvers, therefore a simple plan that connects two states, without colliding with obstacles, might not be enough. Instead, a good plan would take into consideration the kinematic and dynamic constraints of the vehicle so that the path is not only collision-free, but feasible for the robot.

This work focuses on global planners, in particular on the implementation of tools that facilitate the analysis of different path planning algorithms, under the Robot Operating Sys-

tem (ROS).
ROS is a framework nowadays widely used by the robotics community. Its main purpose is to make the development of robot software more flexible. It is a collection of tools, libraries and conventions used to interact with different robotic platforms performing complex tasks.

The navigation stack of ROS already contains both global and local path planners, however only simple algorithms are defined as A* and Dijkstra for the global one and DWA for the local planner.
The planning problem, is not only required to generate plans which are collision-free and feasible, but they might satisfy a constraint in time, length or energy consumption, there the need to switch to optimal planners taking into account the vehicle kinematics, and the ones provided by ROS do not.

The path planning algorithms of interest in this work can be categorized as Search-Based and Sampling-Based; the first group builds a graph of discretized states for the robot and searches for an optimal solution on the graph, while the second group samples some possible states of the robot in the continuous space, trying to connect them to find a sequence of states-connections from the start state to the goal state. For the latter, algorithms as RRT* and PRM* have been proven to be asymptotically optimal.

To tackle our objective of creating modules for these two categories of algorithms, two nodes have been implemented using open source C++ libraries, SBPL (Search Based Planning Algorithm) for the search based algorithms and OMPL (Open Motion Planning Algorithm), for the random sampling algorithms.
Furthermore with the aim to use the planners for Ackerman vehicles, motion primitives have been implemented. In the case of the SBPL library, motion primitives are already supported, thus a modification to the library is done in order to work without motion primitives, so as to have versions of AD* and ARA* without motion primitives and have the possibility to compare with other algorithms.

On the other hand OMPL offers implementations like the Dubins path and the Reeds-Shepp curves to deal with kinodynamic constraints. Since these two do not give enough constraints to construct feasible paths, it was decided to also include motion primitives as in SBPL to perform the path planning.

To do so the thesis is based on an implementation of RRT* with Motion Primitives done by Basak Sakcak [4]. However that work has been modified to work with the same motion primitives used in the SBPL approach and to reach real-time performance.
With this aim, three new methods of setting a sample set are created to improve the performance of the planner, indeed the one provided used the whole space for the random sampling,

which lead to wasted time discarding samples due to the limited amount of motion primitives. Our novel proposals instead of sampling all the space randomly, select one node of the tree and then sample around it, either around a cube (taking certain values of (x, y, θ), or sampling by the possible available motions for the node (with or without removal of the checked primitives)).

The thesis consists of **6** chapters, the **5** chapters that follow this introduction are outlined below

Chapter 2: the Second chapter discusses the state of the art of path planning, which is categorized in Search Based Algorithms and Sampled Based Algorithms, therefore the main concepts of each category and their main algorithms are explained.

Chapter 3: this chapter describes the planning libraries used for the development of the thesis, and the main concepts around their usage. As well as the concepts of ROS middleware and the Navigation Stack.

Chapter 4: the fourth chapter explains the structure and architecture of the designed global planner nodes, to then enter in the details of the SBPL and the OMPL global planners, focusing on the modifications perform to the RRT*_MotionPrimitives planner, and the novel ideas for the sampling modes.

Chapter 5: this chapter contains the different tests performed and the comparisons between the planners.

Chapter 6: the last chapter corresponds to the thesis conclusions and future recommendations.

Chapter 2

State of the art

This chapter shows the state of the art on trajectory planning, starting from historical notes. Then two main categories of planning algorithms are introduced, as well as the most known algorithms. Finally basic concepts with respect to car-like vehicles are explained, specifically about Ackerman Vehicles, in order to understand the need of using motion primitives to achieve good paths.

2.1 Historical Notes

From the past decades path planning has received a considerable amount of attention, since robots keep getting bigger roles in the modern industry and in our daily life. Researchers have tackled many challenges in different areas, such as trajectory calculation, allowing a robot (e.g. autonomous vehicle) to move from one point to another by also considering its kinematic and dynamic constraints.

Even though the problem of motion planning has been recognized since the late 60s, it can be said that time marking the beginning of modern motion planning came until 1979 when Lozano-Pérez and Wesley [5] introduced the concept of configuration space (C-space), by the time, they expressed the requirement of: "growing the obstacles and shrinking the moving object to a point", as the mechanisms needed to obtain the path.

The motion planning problem can be informally described as a sequence of control inputs to drive the robot from its initial state to one of the goal states while obeying the rules of the environment, e.g., not colliding with the surrounding obstacles [6]. An algorithm to address this problem is said to be complete if it terminates in finite time, returning a valid solution if one exists, or failure otherwise.

The first basic version of motion planning, called piano movers problems, is known to be very hard from the computational point of view, therefore it was unsuitable for practical applications. Once the completeness requirement was relaxed into a resolution completeness (return a valid solution, if one exists) practical planners came, as the cell decomposition methods [7], and the potential fields method [8].

These first approaches were based on local planning, they did not attempt to solve the problem in its full generality, but used only the information available at the moving robot to determine the next motion command. These planners had some critical drawbacks as the difficulty to generate complex maneuvers, they were also limited to state spaces with up to five dimensions since the decomposition-based methods suffered from large number of cells, and the potential field methods from local minima [6].

Research efforts were addressed to create powerful global planners, so that the path could be easily followed by the vehicle, since even though some local planners approach use also some global information, the problem still remained that the solution was not always optimal.

Several classifications including different levels can be made for describing current global path planning algorithms as shown on Figure 2.1 [9]; from a high level point of view they can be classified according to the Holonomic, Nonholonomic or Kynodynamic problems they face. The second level classification, and the one we focus on, is the differentiation between the two strategies used to execute the path planning, for the search-based algorithms a first essential step is to model the environment as a graph before searching the optimal or feasible path, while for the sampled-based algorithms the environment model is not necessary. A third and forth level refer to Off-line/On-line algorithms and Deterministic/Probabilistic, respectively. In this thesis we only consider off-line deterministic planners.

2.2 Path Planning Algorithms

In this section the basic elements of planning are mentioned, then in the next section the two categories of path planning algorithms are explained; Search-Based Planning and Sampled-Based Planning, as well as their most known algorithms.

Basic Elements of Planning

There are several basic elements that will arise as part of planning [10]:

- **State** Planning problems involve a state space that captures all the possible situations that could arise, it can also be said that the State Space is the set that contains all the possible configurations of the robot, it can be continuous or discrete. Every element of

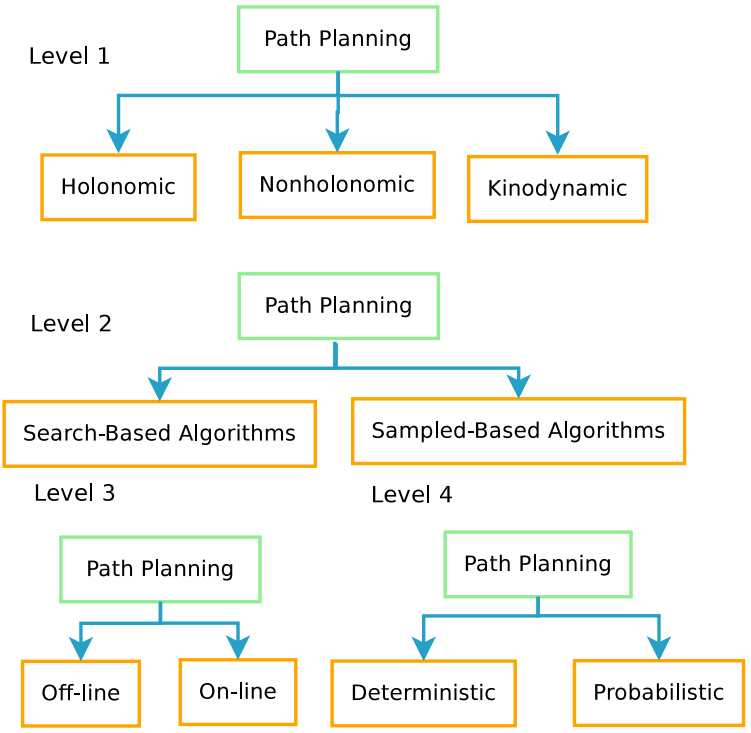


Figure 2.1: Classification of Path Planning Levels

the state space is called state, in a Cartesian environment it can represent the position and orientation of a robot.

- **Time** All planning problems involve a sequence of decisions to be applied over time, it could be explicitly modeled or implicit (actions are done in a succession, particular time is unimportant, but the proper sequence must be maintained)

- **Actions** Actions represent the transitions between the states. The plan generates actions that manipulate the state, in the planning formulation it must be specified how the state changes when actions are applied.

- **Initial and Goal states** The planning problem is generally specified starting in some initial state, and trying to achieve a specified goal, therefore the actions will be selected in order to satisfy this.

- **Criterion** Two kinds of planning criteria exist, Feasibility, which entitles to find a plan that arrives to a goal state regardless of its efficiency, and Optimality that finds a feasible plan optimizing performance in some manner. A planning algorithm is *optimal* if it always finds an optimal path, a path is said to be optimal if the sum of its transition costs is minimal across all possible paths leading from the initial state to the goal state.

The planning algorithm is *complete* if it always finds a path in finite time when one exists.

- **Plan** The plan can specify a sequence of actions to be taken as a function of a state. The appropriate action must be determined from the available information.

2.3 Search-Based Planning

Search-based planning has, in the last years, dominated path planning in the robotics field, mostly because of its relative ease in implementation. This category uses graph search methods to compute paths, or trajectories, over a discrete representation of the problem. Therefore, search-based planning can be seen as a two stage procedure: how to turn the problem into a graph, and how to search the graph to find the best solution [3].

The typical components of a Search-based Planner are mentioned below.

- Graph Construction: in the graph each node will correspond to a state and each edge corresponds to an action, which links two states. The graph gives the information of the successor states for a given state. It can be constructed by using motion primitives, this is positive because it would lead to a feasible path and the possibility to incorporate a variety of constraints, but could also lead to an incomplete graph. Since a graph is inherently discrete, all graph-search algorithms require this discrete representation.

- Cost function: a cost associated with every transition in the graph.

- Heuristic function: indicated as $h(s)$, gives the estimates of cost-to-goal from a given state s. This function must be admissible, meaning that the cost estimated by the heuristic function must be less or equal to the real cost to reach the final state, and to obtain an optimal solution the function must also be consistent, i.e., the cost estimated from a state to each other must be less or equal to the real cost. An heuristic function could be the Euclidean distance from the state s to the goal state.

- Graph search algorithm: there are different algorithms available, some can return an optimal path like Dijkstra and A*, while others return an ϵ sub-optimal path, e.g., weighted A*, ARA*, AD*, R*. Each algorithm must take the decision about which action to execute; to do so they use a function $f(s) = g(s) + h(s)$, where $g(s)$ is the real cost of the path to the current state. Once a state is processed, the possible destinations states are found, this process is called expansion.

2.3.1 Main Search Based Planning Algorithms

Before explaining the different categories of heuristic path planning algorithms some notation must be introduced to understand the pseudo-code to be shown below.

- S: denotes the finite set of states in the graph.
- $succ(s)$: denotes the set of successors of a state $s \in S$. All the states reachable from a state s by performing one of the available actions.
- $pred(s)$: denotes the set of all predecessors of a state $s \in S$. All the states that can reach a state s by performing one of the available actions.
- s_{start}: denotes the start state.
- s_{goal}: denotes the goal state.
- $\pi(s)$: represents a path from state s to s_{goal}, it is a sequence of states $s, s_1, s_2, ..., s_{goal}$ with $s_i \in succ(s)$.
- $c(s, s')$: cost of transitioning from s to s'. Where $s, s' \in S$ such that $s' \in succ(s)$. The cost must be a positive finite value. The cost of a path is the sum of the costs of every transition $\sum_{i=1}^{k} c(s_{i-1}, s_i)$
- $c^*(s, s')$: least cost path from s to s'. If $s = s'$ then $c^*(s, s') = 0$.
- $g^*(s)$: denotes the least-cost path from s_{start} to s
- $f(s) = g(s) + h(s)$: Node evaluation function, to make expansion decision. With $g(s)$, the real cost of the path to the current state and $h(s)$, the heuristic function.

Static Algorithms

Static Algorithms refer to the path planning algorithms that do not consider changes in the environment, a very known static algorithm is the A*, that operates like the Dijkstra algorithm, but the search looks for the most promising states, saving computational resources. An algorithm can have two possible directions, *forward*, it starts expanding from start state to goal, and *backward* from goal state towards start state. A Pseudo-code for the A* forward version is shown in Algorithm 1 and explained below.

According to [11], the algorithm initially sets $g(s) = \infty$ for all $s \in S$, and begins updating the path cost of the start to zero. Then places on the OPEN list the state, this list is actually

Algorithm 1 A* Algorithm (forward version)

```
function ComputeShortestPath()
    while (argim_{s∈OPEN}(g(s) + h(s, s_goal)) ≠ s_goal) do
        remove state s from the front of OPEN;
        for all s' ∈ succ(s) do
            if g(s) > g(s) + c(s, s') then
                g(s') = g(s) + c(s, s');
                insert s' into OPEN with value (g(s') + h(s', s_goal));

function Main()
    for all s ∈ S do
        g(s) = ∞;
    g(s_start) = 0
    OPEN = ∅
    insert s_start into OPEN with value (g(s_start + h(s_start, s_goal));
    ComputeShortestPath()
```

a priority queue. The elements on the list are ordered according to the sum of their path cost and a heuristic estimate of their path cost to the goal ($g(s) + h(s, s_{goal})$). The state with the minimum sum is at the front of the priority queue. The heuristic function typically underestimates the cost of the optimal path from s to s_{goal}. It is used to focus the search. Then the algorithm takes the first element on the queue and updates the cost of all states reachable from this state (from a direct edge), and checks if $g(s) + c(s, s') \leq g(s')$ then the cost of s' is set to this lower value, if this happens then it is placed on the OPEN list. This continues until we reach the s_{goal}. If the heuristic is admissible, then it guarantees an optimal path.

If the search problem is particularly complex, then it might take a lot of time for A* to find an optimal solution, therefore an approximate solution could be useful if presented faster [12]. If the h-values used by A* are consistent, then the algorithm always leads to an optimal solution but a lot of states are expanded. To overcome this problem researchers have explored the effect of weighting the heuristics value, allowing A* to find a bounded optimal solution with less computational effort.

This approach is known as Weighted A*, it uses an inflation factor $\epsilon \geq 1.0$ in its node evaluation function, $f(s) = g(s) + \epsilon \cdot h(s)$. The ϵ factor bounds the suboptimality, if the heuristic used is consistent then the solution will not cost more than ϵ times the cost of the optimal path [11].
Having weighted heuristics accelerates the search since it makes the nodes closer to the goal more attractive, it can be said that, implicitly it, adjusts a tradeoff between search effort and solution quality.

Figure 2.2 shows an example of A* algorithm with inflated heuristic (weighted A*). The black cells of the grid represent the obstacles, the grey cells represent the expanded state, while

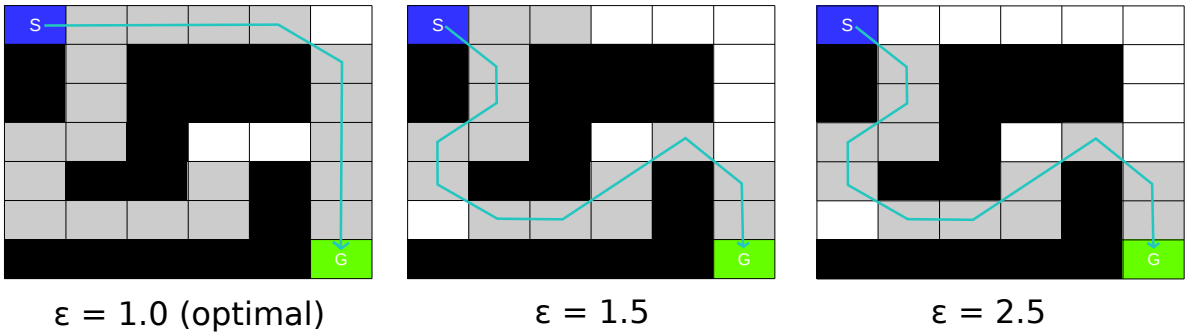


Figure 2.2: A search with weighted heuristic*

the blue cell and the green cell represent the starting state and the goal state respectively. The cost of moving from one cell to its neighbor is one. Finally the path is shown in a light blue color, it can be seen that the higher the inflating factor the less expanded cells can be seen, however the solution is suboptimal.

Anytime Algorithms

Having an optimal search can be infeasible for real world problems, since there is a little amount of time for obtaining the best solution. Anytime algorithms, can be expressed as time bounded algorithms, and they are useful in order to find a good solution that can be generated in the available time. Anytime algorithms typically construct an initial, possibly highly suboptimal, solution very quickly, then improve the quality of this solution while time permits [11].

The Weighted A* algorithm that we discussed previously can be transformed into an anytime algorithm, by running a succession of A* searches with decreasing inflation factor.This approach wastes a lot of computation since each iteration duplicates most of the efforts of the previous searches. There are several algorithms that deal with this problem, like the ARA* (Anytime Repairing A*) algorithm proposed in [13], and the ANA* (Anytime Nonparameteric A*) algorithm in [14].

ARA* is an efficient anytime heuristic search that runs A* with inflated heuristics in succession; it starts with a large ϵ and decreases ϵ prior to each execution until $\epsilon = 1$, but it reuses search efforts from previous executions so that the sub-optimality bounds are still satisfied. Pseudo-code for the ARA* is shown in Algorithm 2 and explained hereafter. Differently from A* it introduces a new function ImprovePath that brings the notion of *local inconsistency*, a decrease in the cost path g(s) introduces a local inconsistency between the g-value of s and the g-values of its successors. The local inconsistency is propagated to the

Algorithm 2 ARA* Algorithm

```
function f(s)
    return g(s) + ε · h(s)

function ImprovePath()
    while (f(s_goal) ≥ min_{s∈OPEN}(f(s))) do
        remove s with the smallest f(s) from OPEN;
        CLOSED = CLOSED ⋃ {s}
        for all s' ∈ succ(s) do
            if s' was not visited before then
                g(s') = ∞;
            if g(s') > g(s) + c(s, s') then
                g(s') = g(s) + c(s, s')
                if s' ∉ CLOSED
                    insert s' into OPEN with value (f(s'));
                else insert s' into INCONS;

function Main()
    g(s_goal) = ∞;
    g(s_start = 0;
    OPEN = CLOSED = INCONS = ∅;
    insert s_start into OPEN with f(s_start);
    ImprovePath()
    ε' = min(ε, g(s_goal)/min_{s∈OPEN⋃INCONS}(g(s) +
h(s)));
    publish current ε'-suboptimal solution;
    while ε' > 1 do
        decrease ε;
        Move states from INCONS into OPEN;
        Update the priority for all s ∈ OPEN according
to f(s);
        CLOSED = ∅;
        ImprovePath();
        ε' = min(ε, g(s_goal)/min_{s∈OPEN⋃INCONS}(g(s)+
h(s)));
        publish current ε'-suboptimal solution;
```

children of s via a series of expansions. Now the OPEN list can be viewed as a set of states from which it is needed to propagate the local inconsistency. Also in order to hold the sub-optimal bound of ϵ each state is restricted to be expanded no more than once, to implement this restriction, any state whose g-value is lowered is checked and inserted into OPEN if it has not been previously expanded. Since OPEN only holds the locally inconsistent states that have not yet been expanded, then a new set is defined as INCONS which holds the locally inconsistent that are not in OPEN. The ImprovePath function stops as soon as $f(s_{goal})$ is equal to the minimal f(s) among all the states on OPEN list.

The main function of ARA* makes the initialization, and then repetitively calls the ImprovePath function, with decreasing values of ϵ. A new OPEN list is constructed, before each call to ImprovePath, by moving the contents of the set INCONS, and then it is reordered by the f(s) values.

On the other hand, ANA* does not use an ϵ based on a linear trajectory with ad-hoc parameters chosen by the user like ARA*, instead, adaptively reduces ϵ to expand the most promising nodes per iteration, adapting the greediness of the search as path quality improves. Pseudo-code for the ANA* is shown in Algorithm 3 and explained hereafter.
This algorithm maintains a global variable G that stores the cost of the current best solution (initially $G = \infty$), its ImproveSolution function implements a version of A* that is adapted such that the expansion of the states in the OPEN list are done according to the maximal value of $e(s) = \frac{G-g(s)}{h(s)}$. When a state is expanded g(s) is checked to see if it can be decreased for each of the successor states, if so then g(s') is updated and its predecessor is set to s so

Algorithm 3 ANA* Algorithm

```
function ImproveSolution()
    while OPEN is not empty do
        s = max_{s∈OPEN}{e(s)};
        OPEN = OPEN{s};
        if e(s) < E then
            E = e(s);
        if IsGoal(s) then
            G = g(s);
        for each succ(s') of s do
            if g(s) + c(s, s') < g(s') then
                g(s') = g(s) + c(s, s');
                pred(s') = s;
                if g(s') + h(s') < G then
                    Insert or update s' in OPEN with key e(s');

function Main()
    G = ∞;
    E = ∞;
    OPEN = Ø;
    g(s) = ∞;
    g(s_start = 0;
    insert s_start into OPEN with keye(s_start);
    ImproveSolution()
    while OPEN is not empty do
        ImproveSolution();
        Report current E-suboptimal solution;
        Update keys e(s) in OPEN and prune if g(s) + h(s) ≥ G;
```

that once a solution is found it can be reconstructed. Then s' is inserted into the OPEN list with a key e(s'), if it is already in OPEN then only the key is updated. If $e(s) \leq 1$ then the state is not put in the OPEN list, so that it can be guaranteed that when the goal is extracted from this list the solution has lower cost than the current-best solution.

Incremental Replanning Algorithms

The previous algorithms have a good performance when dealing with static environments, meaning that the graph is known by the agent. But in real scenarios the agents are typically equipped with sensors that provide updated information of the environment to compensate the errors or changes of the initial graph.

A plan with the initial graph might not be valid with the updated data, therefore it is important to have the possibility to replan when the graph is updated with the new information in order to reduce the total cost of the traverse. Thus, a possibility would be to replan from scratch with the new graph, but this becomes computationally expensive, a better solution is to take the previous solution and repair it considering the changes of the graph. During the replanning process, the robot can stop and wait for a new path, or it could also continue along a suboptimal path; in either ways rapid replanning is essential.

In [15] the Dynamic A* (D*) algorithm is presented, this algorithm resembles A*, except that is dynamic in the sense that the arc costs can change during the traverse of the solution path. It is capable of planning optimal trajectories in real time by taking into account that most of the arc cost corrections occur near the robot and that the path needs only to be replanned out to the robot's current state.

According to [11] the Focused Dynamic A* (D*) and D* Lite are the most widely used of the incremental replanning algorithms. They both maintain a least-cost path between a start state and any number of goal states as the cost of arcs changes, they handle increasing or decreasing arc cost and dynamic start states. Since both algorithms are very similar we explain below the D* Lite version since is easier to analyze.

D* Lite stores an estimate of the cost from each state to the goal, and also a one-step lookahead cost rhs(s), such that:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ min_{s' \in succ(s)}(c(s, s') + g(s')) & \text{otherwise} \end{cases} \tag{2.1}$$

A state is called consistent if and only if its g-value equals its rhs-value, otherwise is over-consistent ($g(s) > rhs(s)$) or under-consistent ($g(s) < rhs(s)$). As A*, it also uses a heuristic and priority queue to focus the search and to order efficiently the cost updates. The OPEN list will holds the inconsistent states, which are the states that need to be updated and made consistent. If the arc cost changes after the initial solution, then the algorithm will update the rhs-values of the states affected by the changed arc cost, and place them on the queue. The key value now incorporates two values:

$$key(s) = [k_1(s), k_2(s)] = [min(g(s), rhs(s)) + h(s_{start}, s), min(g(s), rhs(s))] \tag{2.2}$$

By incorporating the second value, the algorithm can ensure that the states that are along the current path and on the queue are processed in the most efficient order. The pseudo-code of D*-Lite is shown in Algorithm 4, [16].

Anytime Incremental Replanning Algorithms

As explained before there are algorithms to deal with complex planning problems that require to get suboptimal solutions due to time constraints e.g., ARA*, and algorithms to deal with changing environments like D*. Anytime Incremental Replanning Algorithms are able to deal with both a dynamic environment and complex planning problems. In [16] the Anytime

Algorithm 4 D* Lite Algorithm

```
function KEY(S)
    return[min(g(s), rhs(s)) + h(s_start, s), min(g(s), rhs(s))];

function UPDATESTATE()
    if s was not visited before then
        g(s) = ∞;
    if s ≠ s_goal then
        rhs(s) = min_{s'∈succ(s)}(c(s, s') + g(s'));
    if s ∈ OPEN then
        remove s from OPEN;
    if g(s) ≠ rhs(s) then
        insert s into OPEN with key(s);

function COMPUTESHORTESTPATH()
    while (min_{s∈OPEN}(key(s) < key(s_start) OR rhs(s_start) ≠ g(s_start)) do
        remove state s with the minimum key from OPEN;
        if (g(s) > rhs(s)) then
            g(s) = rhs(s);
            for all s' ∈ pred(s) : UpdateState(s');
        else
            g(s) = ∞;
            for all s' ∈ pred(s) ∪ s : UpdateState(s');

function MAIN()
    rhs(s_start) = ∞;
    g(s_start = ∞;
    g(s_goal) = ∞;
    OPEN = ∅;
    rhs(s_goal) = 0;
    insert s_goal into OPEN with key(s_goal);
    forever
    ComputeShortestPath();
    Wait for the changes in the edge costs;
    for all directed edges (u,v) with changed edge costs do
        Update the edge cost c(u,v);
        UpdateState(u);
```

Dynamic A* (AD*) algorithm is proposed, it performs a series of searches using decreasing inflation factors to generate a series of solutions with improved bounds, and, if changes are detected in the environment, the locally affected states are placed on the OPEN queue with priorities equal to the minimum of their previous key value and their new key value (as seen previously with $k_2(s)$ in D* Lite). The solution is guaranteed to be ϵ-suboptimal.

Algorithm 5 shows a pseudo-code of AD* implementation. It starts by setting ϵ_0 to a sufficiently high value, so that a first suboptimal solution can be generated very quickly. If no changes are detected in the environment then it behaves as ARA* gradually decreasing ϵ until an optimal solution is found. If there are changes in the edges cost, it checks how substantial the changes are, if not so much, then acts like D* updating the edges, if too much has change, then the ϵ is increased or it can also replan from scratch. Unlike in ARA* the states need to be reinserted into OPEN because they may become underconsistent; once inserted, states key values must use admissible heuristic values. To guarantee that the underconsistent

states propagate their new cost to their affected neighbors, the key values must be computed differently for underconsistent states and for overconsistent[11].

Algorithm 5 AD* Algorithm

```
function KEY(S)
    if g(s) ¿ rhs(s) then
        return [min(g(s), rhs(s)) + ε · h(s_start, s);
        min(g(s), rhs(s))];
    else
        return [min(g(s), rhs(s)) + h(s_start, s);
        min(g(s), rhs(s))];

function UPDATESTATE()
    if s was not visited before then
        g(s) = ∞;
    if s ≠ s_goal then
        rhs(s) = min_{s'∈succ(s)}(c(s, s') + g(s'));
    if s ∈ OPEN then
        remove s from OPEN;
    if g(s) ≠ rhs(s) then
        if s ∉ CLOSED then
            insert s into OPEN with key(s);
        else
            insert s into INCONS;

function COMPUTEORIMPROVEPATH()
    while min_{s∈OPEN}(key(s) < key(s_start) OR rhs(s_start) ≠
g(s_start)) do;
        remove state s with the minimum key from OPEN;
        if ( theng(s)¿rhs(s))
            g(s) = rhs(s);
            CLOSED = CLOSED ∪ s;
            foralls' ∈ pred(s) : UpdateState(s');
        else
            g(s) = ∞;
            foralls' ∈ pred(s) ∪ s : UpdateState(s');

function MAIN()
    g(s_start = rhs(s_start) = ∞;
    g(s_goal) = ∞;
    rhs(s_goal = 0; ε = ε_0;
    OPEN = CLOSED = INCONS = Ø;
    insert s_goal into OPEN with key(s_goal);
    ComputeOrImprovePath();
    publish current ε-suboptimal solution;
    forever;
    if changes in edge costs are detected then
        for all directed edges (u,v) with changed edge
costs do
            Update the edge cost c(u,v);
            UpdateState(u);
    if significant edge cost changes are observed then
        increase ε or replan from scratch;
    else if ε > 1 then
        decrease ε;
    Move states from INCONS into OPEN;
    Update the priorities for all s ∈ OPEN according to
key(s);
    CLOSED = Ø;
    ComputeOrImprovePath();
    publish current ε-suboptimal solution;
    if ε = 1 then
        wait for changes in edge costs;
```

2.4 Sampling-Based Planning

Instead of using an explicit representation of the environment, like search-based algorithms, sampling-based algorithms rely on a collision checking module, providing information about feasibility of candidate trajectories, and they connect a set of points randomly sampled from the obstacle-free space in order to build a graph (roadmap) of feasible trajectories. The roadmap is then used to construct the solution to the original motion-planning problem [6].

Sampling Based planning has emerged as a way to avoid explicit constructions of the obstacles in the C-Space; since there is no explicit model of the obstacles there is no need to characterize all possible collision conditions [17].

Sampling Based algorithms are well-suited to solve high-dimensional problems as the size of the graph they construct is not explicitly exponential in the number of dimensions, thus they are particularly useful for planning of serial manipulators and in other situations where higher-dimensional planning is required.

A problem that these methods experience is their lack of repeatability, each run should return a different path. This could be something positive, in the fact that unlike a deterministic algorithm it is possible that if solving a problem at some moment takes a long time, it could do better next time. But it is negative in the fact that the algorithm performance can be obscured, and it is difficult to detect possible flaws.

In [18] a general scheme is proposed for the problem of planning collision-free paths for an arbitrary holonomic robot with n-degrees of freedom with randomized planners. They made the following considerations:

- The configuration space of the robot is taken as C, and the subset of collision free configurations as C_{free}.
- The plan problem is specified as the start and goal configuration: q_{start} and q_{goal}.
- A path in C_{free} that joins q_{start} and q_{goal} is a solution of the problem.
- Each configuration q is described as an n-tuple. In our particular interest the tuple is (x, y, θ).
- There exists a function CLEARENCE which maps the distance between the robot and the obstacles. If CLEARENCE(q) is positive, then that configuration belongs to C_{free}.
- There exists a parameter ρ, that bounds the distance between two consecutive configurations.

From the previous statements, the general planning scheme is as follows:

- A random configuration (q_{rand}) is taken from C.
- Keep the configurations that belong to C_{free}, as the nodes of a graph (G).
- The start and goal configuration will also be nodes of G.
- Connect adjacent configurations by links of the graph.
- If q_{start} and q_{goal} are connected by the links of G, then we have a solution. Else a path has not been found withing the taken configurations (the amount of random configurations to consider is also a parameter given to the planner).

Since these kind of algorithms usually do not sample the whole space, but just a finite amount of random samples are taken, they cannot determine if a solution exists or not, then if a solution is not found, it does not necessarily mean that it does not exist.Therefore, most of the sampling-based planning algorithms are complete provided the sampling is done infinitely long.
These approaches have the a difficulty to find optimal paths, even though they are more rapid and less expensive in term of resources than Search Based planning. The standard sampling-based planners are use to find feasible paths without the notion of cost. To speed up planning usually simple heuristics are employed like sampling towards the goal (goal biasing). However, this does not take any notion of cost into account and is only concerned with finding a path. But, in recent years some Sampling-Based planners have been proven to asymptotically optimal [6].

In synthesis these algorithms rely on some method for generating samples over a finite state space, that computes generally a uniform set of random robot configurations, and connects these samples via collision free paths that would respect the motion constraints of the robot to answer the query.

The main Sampled-Based Planning Algorithms are explained in the next section.

2.4.1 Main Sampling-Based Planning Algorithms

Sampling based methods can be divided into two categories: multi-query and single-query. For the first one the most known algorithm is the Probabilistic Roadmap (PRM), and for the second one the Rapidly-Exploring Random Tree (RRT), these are explained ahead as well as their optimal versions PRM* and RRT*, which are provably asymptotically optimal.

Multi-query Algorithms

This category is said to be multi-query because it can connect multiple start and goal configurations without having to reconstruct the graph structure each time.

PRM was introduced in [19] as a planning method for robots in static workspaces, the method consists of two phases, the *learning phase* and the *query phase*.

1. **Learning phase**, in the first phase two steps are taken:

   (a) The construction of the probabilistic roadmap by randomly sampling free configurations of the robot and connecting them with a motion planner, the graph is undirected and is expressed by the letter $R = (N, E)$. The nodes (N) are free configurations of the roadmap, while the connections obtained with the local planner form the edges (E) of the roadmap, an edge is a feasible path between two configurations.

   (b) Then some post-processing is done to the roadmap to improve its connectivity, this is named expansion step. It selects nodes of the roadmap that are inside difficult regions of the configuration space and it expands the graph creating additional nodes in this area.

   The time to spend in the learning phase is a parameter that must be given, it depends on the difficulty of the planning problem. A pseudo-code of this step is shown in Algorithm 6, to understand it the following symbols are explained:

   - **c**, represents a new node (random free configuration sampled).
   - The distance function D(a,b), returns the distance between the configurations a and b.
   - **Nc**, set of candidate neighbors of c chosen from N, this set is formed selecting nodes in N within a certain maximum distance (fixed radius **r**) from c to do this the function D is employed.
   - $\Delta(a, b)$, returns true if the local planner can compute a path between the two given configurations (a and b).

   Figure 2.3 shows an example of how the PRM Roadmap is constructed.

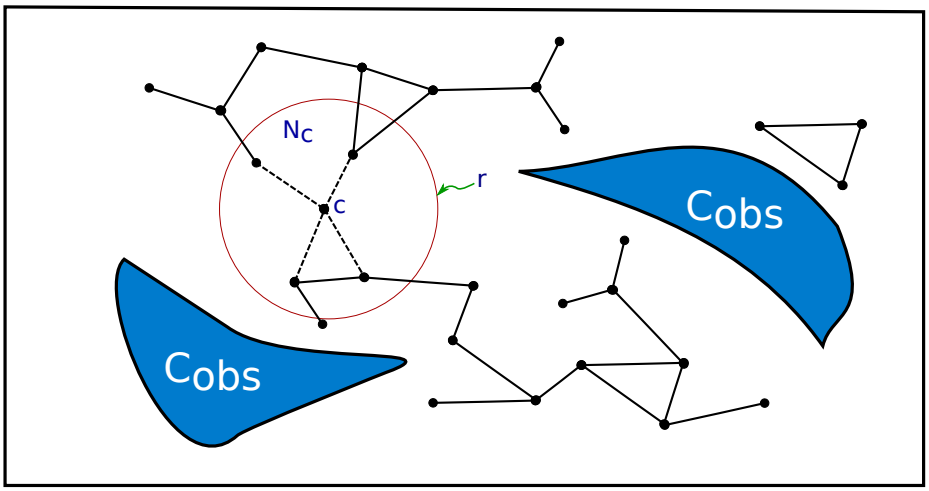


Figure 2.3: PRM Construction of the RoadMap

Algorithm 6 PRM, construction step

```
function ConstructionStep()
    N = ∅
    E = ∅
    while time is not over do
        Get a random free configuration c;
        Get a set of candidate neighbors Nc;
        for all n ∈ Nc in order of increasing D(c,n) do
            if not_already_connected(c,n) and Δ(c, n) then
                Add the edge connecting (c,n);
                Update R with the connected components;
    return R =(N,E);
```

2. **Query phase**, The query is the request to solve a path planning problem between a start and a goal configuration. This process is as follows:

   - Try to find a path from the start and goal configurations (q_{start}, q_{goal}), to nodes of the roadmap $(\tilde{q}_{start}, \tilde{q}_{goal})$.
   - If the previous step is successful, then perform a graph search to find the sequence of edges that connect $(\tilde{q}_{start}, \tilde{q}_{goal})$ in the roadmap.
   - Form the feasible path as the concatenation of the successive path segments

   Figure 2.4 shows how a PRM query would be solved, it shows in green the connections between the start and goal configurations to the nearest nodes in the roadmap, and in yellow the final feasible path.

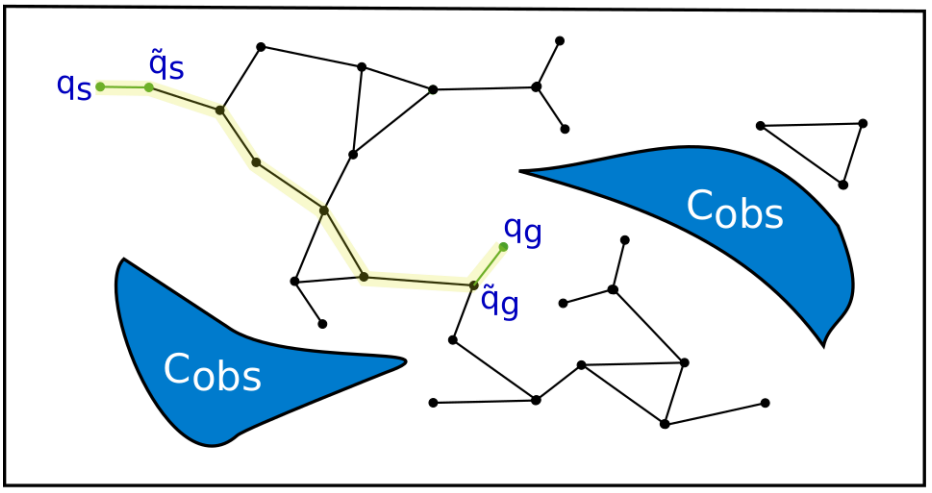


Figure 2.4: PRM Solving a query

It can be noted that there is a tradeoff between the invested time in precomputation and the attempt to solve a query, the invested time is worth if the application requires numerous queries in the same environment.

The **Optimal Probabilistic RoadMaps (PRM*)** was presented in [6], here they show that PRM* is asymptotically optimal.

Unlike its predecessor, PRM* does not make the connections between the roadmap nodes with a fixed radius, instead the radius is chosen as a function of the number of samples n, this connection radius decreases with the number of samples, and also the average number of connections made at each iteration is proportional to the natural logarithm of the amount of samples. The function for the radius is shown in Equation 2.3, having:

- d as the dimension of the space C.
- $\mu(C_{free})$ as the volume of the obstacle-free space.
- ζ_d as the volume of the unitary ball in the d-dimensional Euclidean space.

$$\begin{aligned} r = r(n) &:= \gamma_{PRM}(\log(n)/n)^{1/d} \\ \gamma_{PRM} > \gamma^*_{PRM} &= 2(1+1/d)^{1/d} * (\mu(C_{free})/\zeta_d)^{1/d} \end{aligned} \tag{2.3}$$

A pseudo-code of PRM* is shown in Algorithm 7, as it can be seen the neighbors of the sampled configuration are extracted with a different radius following the previous equation.

Algorithm 7 PRM*, construction step

```
function ConstructionStep()
    N = q_start ⋃ {SampleFree_i}_{i=1,...,n}
    for all v ∈ N do
        Get U ← Near(R = (N, E), v, γ_PRM(log(n)/n)^{1/d}) \ (v)
        for all u ∈ U do
            if (CollisionFree(v,u)) then
                E ← E ⋃ {(v, u), (u, v)}
    return R =(N,E);
```

Single-query Algorithms

The concept of **Rapidly-exploring Random Trees** was introduced by Steven M. LaValle in [20], it is a randomized data structure, whose advantage over PRM is that it can be applied to nonholonomic and kinodynamic planning. Unlike the previous one, this is a single-query algorithm, since for every set of start and goal configurations the tree is built from scratch, with its initial node being the start configuration (x_{init}).

In a standard problem, we would consider our path planning space to be a metric space X, which would be the configuration space, e.g., C (X=C). Since RRTs are meant to solve kinodynamic planning problems, the space X is the tangent bundle of the configuration space ($X = T(C)$) which is assumed to be bounded, so, for instance, X could be composed by the configuration space, and the velocity bounds, thus a state in X_{obs} could be either a state whose velocity is outside of the velocity bounds, or it could be a state inside an obstacle of the environment. The representation of X can vary, but for all kinds of representations, there must be the possibility to check if a certain state lies in X_{obs}, or in its complement X_{free}.

The construction of the tree τ is done as shown below, and a pseudo-code is shown in Algorithm 8.

1. The first node or vertex of the tree is the initial state x_{init}.

2. K iterations are performed. On each iteration a random state, x_{rand} is selected from the space.

3. Then the algorithm finds the nearest neighbor (x_{near}) to the random state, this is done in terms of ρ which is a distance metric on the state space.

4. If x_{rand} is accesible to x_{near}, acoording to ρ, then τ is expanded connecting this two states.

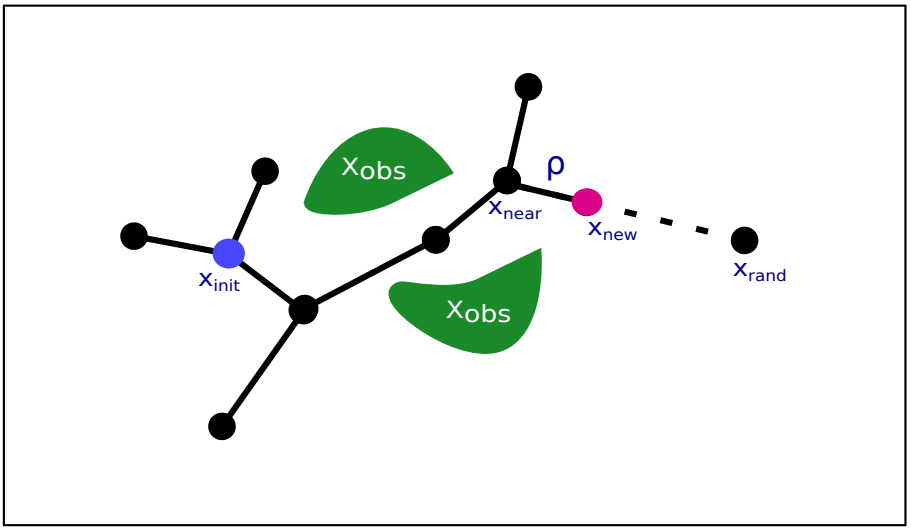


Figure 2.5: Generating RRT

Algorithm 8 Generate RRT

function GENERTATE_RRT $(x_{init}, K, \Delta t)$
  $\tau.init(x_{init})$
  for all $k \in K$ **do**
    $x_{rand} \leftarrow RandomState(k)$;
    $x_{near} \leftarrow Nearest(x_{rand}, \tau)$;
    $(x_{new}, U_{new}) \leftarrow Steer(x_{near}, x_{rand})$;
    if ObstacleFree(x_{new}) **then**
      $\tau \leftarrow InsertNode(x_{new})$
  return τ;

5. If is not possible to connect them, then a new node x_{new} is given by a Steer function that expands x_{near} to x_{new} [21].

6. For connecting this new state to the tree, a Collision Check must be done.

Figure 2.5 shows the process of how the tree is built.

The **Optimal RRT** (RRT*) was introduced in [6] like PRM*. It comes from a modification of the Rapidly-exploring Random Graph (RRG), this algorithm is similar to RRT, in the sense that it tries to connect to the nearest node of the tree, but then unlike RRT, it tries to connect to all the nodes inside a certain radius like PRM, resulting in an undirected graph, which can also contain cycles. RRT, which is a directed graph, is a subgraph of RRG.

RRT* behaves very similar to its predecessor RRT, in the sense that it inherits all of its properties and works similar, but introduces new features; the near neighbor search and the rewiring operation. The first one tries to find which is the best parent among the nodes within a certain radius, defined like in PRM*, as seen in Equation 2.4, having d as the dimensional space and γ as a planning constant based on the environment [21].

$$k = \gamma(\log(n)/n)^{1/d} \tag{2.4}$$

While the second new feature searches among the near nodes of the tree, and checks the cost for each of them, if there is a better parent (a path with less cost) then a rewire is made (the edge is removed form the previous parent and the new one is added). A pseudo-code of the Creation of the RRT* tree is shown in the Algorithm 9, and Figure 2.6 shows an exemplifies this behavior.

Algorithm 9 Generate RRT*

function GENERTATE_RRT* $(x_{init}, K, \Delta t)$
  $\tau.init(x_{init})$
  for all $v \in V$ **do**
    $x_{rand} \leftarrow RandomState(n)$;
    $x_{near} \leftarrow Nearest(x_{rand}, \tau)$;
    $(x_{new}, U_{new}) \leftarrow Steer(x_{near}, x_{rand})$;
    if ObstacleFree(x_{new}) **then**
      $X_{near} \leftarrow Near(\tau, x_{new}, k)$
      $x_{min} \leftarrow x_{nearest}$
      $c_{min} \leftarrow Cost(x_{nearest}) + c(x_{nearest}, x_{new})$
      //Get best parent
      for $x_{near} \in X_{near}$ **do**
        if CollFree(x_{near}, x_{new}) and Cost(x_{near}) + c($x_{nearest}, x_{new}$) $< c_{min}$ **then**
          $x_{min} \leftarrow x_{near}$
          $c_{min} \leftarrow Cost(x_{near}) + c(Line(x_{nearest}, x_{new}))$
        $E \leftarrow E \bigcup \{(x_{min}, x_{new})\}$
      //Rewire Tree
      for $x_{near} \in X_{near}$ **do**
        if CollFree(x_{near}, x_{new}) and $Cost(x_{new}) + c(x_{near}, x_{new}) < Cost(x_{near})$ **then**
          $x_{parent} \leftarrow Parent(x_{near})$
          $E \leftarrow E \backslash \{(x_{parent}, x_{near})\} \bigcup \{(x_{new}, x_{near})\}$;
  return τ;

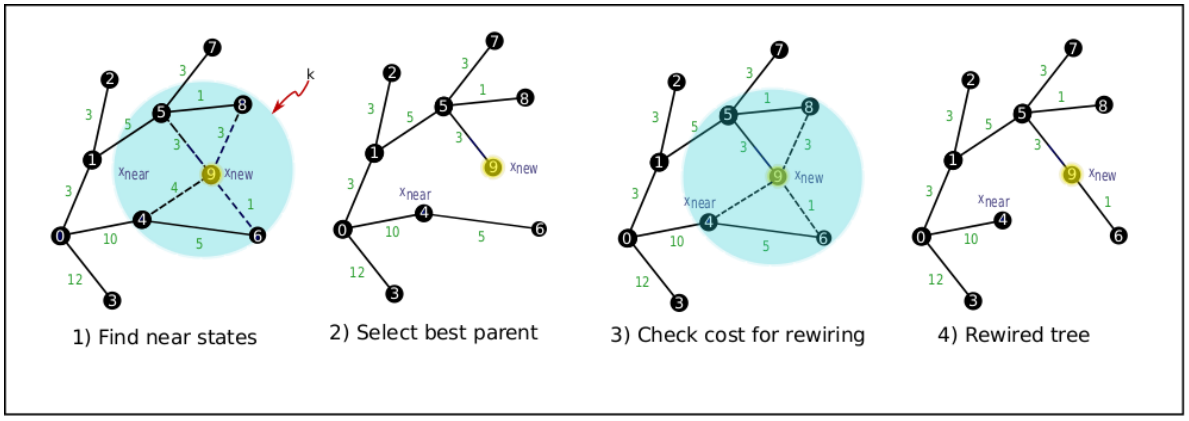


*Figure 2.6: Generating RRT**

It is important to mention that due to the asymptotic quality of RRT*, as the number of iterations increases, the path cost improves gradually. Unlike RRT that does not improve its path, which is suboptimal.

To conclude this section about the Main Sampling-Based Algorithms, the Table 2.1 summarizes some of the main characteristics of the Multi-query and Single-query Algorithms mentioned.

Table 2.1: Comparison between multi-query and single query methods

	Multi-query	**Single-query**
Phases	1. Road construction 2. Searching	Roadmap construction (possible to rewire) and searching online
Typical algorithm	Probabilistic Roadmaps (PRM and PRM*)	Rapidly Exploring Random Trees (RRT and RRT*)
Pros	Fast Searching	No preprocessing
Cons	Can not deal with environment changes	No memory

2.5 Car-like Vehicles

As we mentioned before this thesis focuses in path planning for non-holonomic systems, although the generated implementations are not restricted to these kind of vehicles. In this section the Ackerman vehicle is described to explain the kinematic constraints we take into account in planning.

The notion of nonholonomy appeared in the path planning literature in the 80s with the problem of parallel car parking. The reason why parallel parking is challenging is because of the rolling constraints, i.e., to drive a car sideways the wheels would have to slide instead of roll, which is not possible for a simple car.

Nonholonomic systems are characterized by constraint equations involving time derivatives of the system configuration variables. This usually happens when a system has less controls than configuration variables. A car-like vehicle has two controls, from the driver's point of view the controls are the accelerator and the steering wheel (linear and angular velocities), but it moves in a 3-dimensional space (x, y, θ), thus not every path in the configuration space is a feasible path for the vehicle [22]. Therefore, motion planning gets a second level of difficulty, not only it is required to avoid the obstacles of the environment, but it also has to deal with the non-holonomic constraints of the vehicle.

An Ackerman vehicle has the peculiarity of the steering geometry, named after the men that patented it in 1818. The principle consists of having two pivoted arms on which the wheels rotate instead of the pivoted fore carriage of an ordinary vehicle. This ensures that there is only a rolling action on the tyre of the wheel when turning. Plus this steering model has the characteristic that the axles of the wheels converge to one point during a curve. The front external wheel has a greater turning radius than the internal wheel, thus a lower steering angle, meaning there are two different steering angles, whose mean forms the called ackermann angle. As seen in Figure 2.7 the rear wheels are fixed, then the common point is placed on the extension of the rear wheels axle.

The kinematic model of the vehicle is composed by tree differential equations, as presented in Equation 2.5, having:

- **x** Cartesian position along the x-axis
- **y** Cartesian position along the y-axis
- $\boldsymbol{\theta}$ Orientation of the vehicle
- **v** Linear velocity
- $\boldsymbol{\delta}$ Steering angle
- **L** Wheelbase
- $\boldsymbol{\rho}$ Turning radius

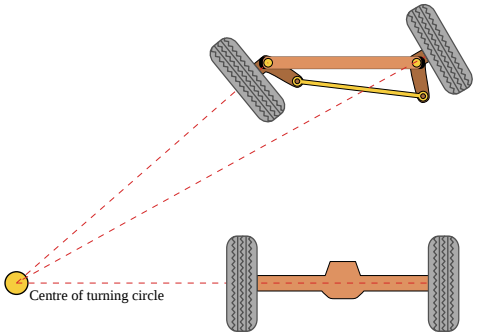


Figure 2.7: Ackerman steering geometry

$$\begin{cases} \dot{x} = v \cdot cos(\theta) \\ \dot{y} = v \cdot sin(\theta) \\ \dot{\theta} = v \cdot \frac{tan(\delta)}{L} \end{cases} \tag{2.5}$$

Figure 2.8 represents the motion of the vehicle.

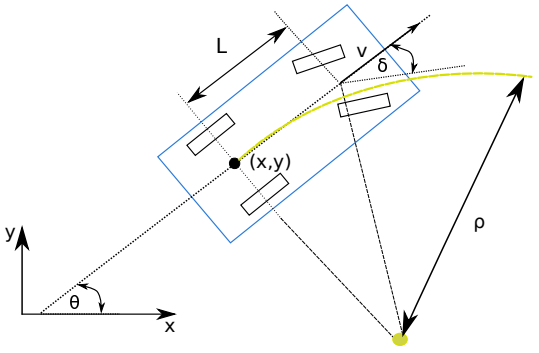


Figure 2.8: Kinematic Model

The previous model can be modified as to only describe the geometrical path travel by the vehicle by excluding the velocity, having as parameter the space travel in the movement. Motion primitives can be obtained by solving Boundary Value Problems (BVPs) by minimizing the space traveled from this model (Equation 2.5). These equations are used for the creation of SBPL primitives, Section 3.1.1 explains how they are used. More information regarding the motion primitives creation can be checked in [23].

$$\begin{cases} \frac{dx}{ds} = cos(\theta) \\ \frac{dy}{ds} = sin(\theta) \\ \frac{\theta}{ds} = \frac{tan(\delta)}{L} \end{cases} \tag{2.6}$$

2.5.1 Car-like spaces

Other than the possibility to use motion primitives for the path planning problem, there are vehicle models that can be used to compute the optimal path between a start and end state, two of the most commonly studied vehicles ,which are first order models,are the Dubins car and the Reeds-Shepp car ([24], [25]).

Dubins gives a sufficient set of paths, i.e. a set which always contains what he called a geodesic, or optimal path, its possible motions are: go straight, turn left and turn right. In the case of Reeds-Shepp car, it has the same motions, but can also execute them in reverse.

In OMPL for these two car models, a geometric planner can be used, but instead of interpolation in a straight line between states, two state spaces are created (ompl::base::DubinsStateSpace and ompl::base::ReedsSheppStateSpace) in order to use the appropriate optimal path in the local planner [26].

An example on the Reeds-Shepp car, as explained in [26], is shown in Figure 2.9, it shows a long corridor with start state in green and goal states in blue, both states are facing the wall. As seen in the end, the path is not globally optimal, they are only optimal between sampled states.

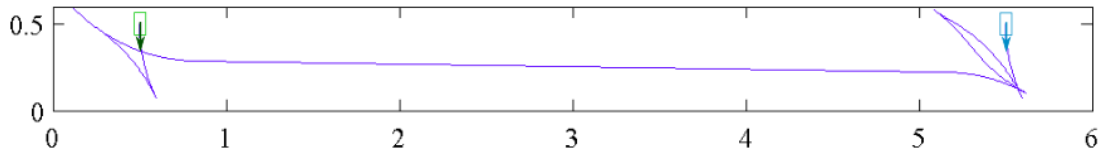


Figure 2.9: Reeds-Shepp car example

Chapter 3

Planning Libraries And The Robot Operating System

This chapter explains two planning libraries that were used for the development of the thesis. The Search Based Planning Library (SBPL) and the Open Motion Planning Library (OMPL). Then the Robot Operating System (ROS) is introduced, as well as its Navigation stack, in particular the move_base package, this will give an overview to the general elements needed for implementing a global planner node.

3.1 Search-Based Planning Library

Search Based Planning Library (SBPL) is a standalone C++ library that provides implementations of various search-based planning algorithms, it can be used under Windows or Linux, and it can also be integrated with ROS. As described in the previous chapter, planning with search-based methods has two parts, representing the problem as a graph, then solving the graph with graph-search methods, therefore SBPL has two objects it uses for these two roles: the environment and the planner.

For our purposes we are mostly interested in the environment object, since most of the configurations are done to it. The environment handles the domain specific details and translates them into a generic form for the planner to handle, which lets the planner find a solution without having to know about the domain. It also implements the heuristic computation. The structure of the SBPL library can be seen on Figure 3.1 that shows a diagram of the communication between the Environment and the Planner (graph search algorithm).

A hierarchy of its two main elements is provided by SBPL. Figure 3.2 shows that Dis-

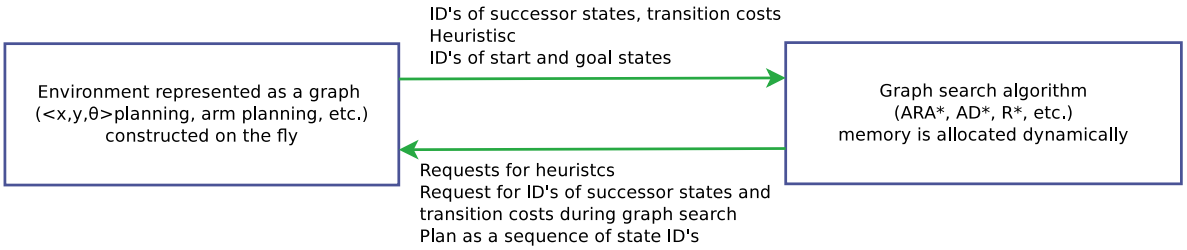


Figure 3.1: SBPL Structure

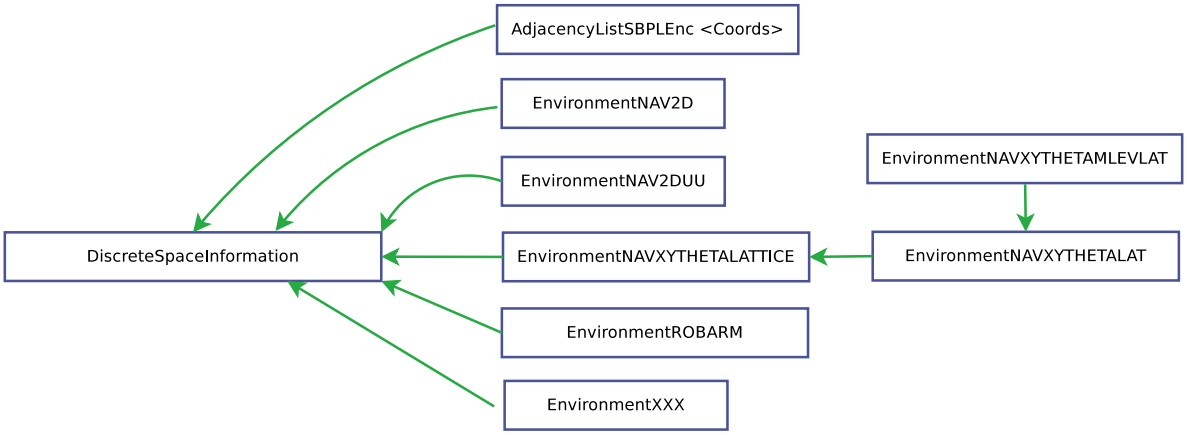


Figure 3.2: Hierarchy of existing Environment classes of SBPL

creteSpaceInformation is the base class for environments defining planning graphs, it is used by the planners to communicate with every type of environment, and the communication is through stateID, so the graph search does not know anything about the actual variables. The Environment is in charge of maintaining a mapping from stateID to the actual state variables (coordinates). The most relevant Environment classes are described below [23],[3].

- **AdjancencyListSBPLEnv** SBPL Environment represented as an adjacency list graph.
- **EnvironmentNAV2D** Grid shaped environment, the robot states are defined as a pair of coordinates (x, y). Useful for simple navigation.
- **EnvironmentNAVXYTHETALATTICE** Base class to formalize the 3D planning using lattice-based graph problem. The state is described by the tuple (x, y, θ). It accepts also motion primitives.
- **EnvironmentNAVXYTHETALAT** Extends the functionality of the previous one, it provides new methods, like setting start and goal states expressed in meters and

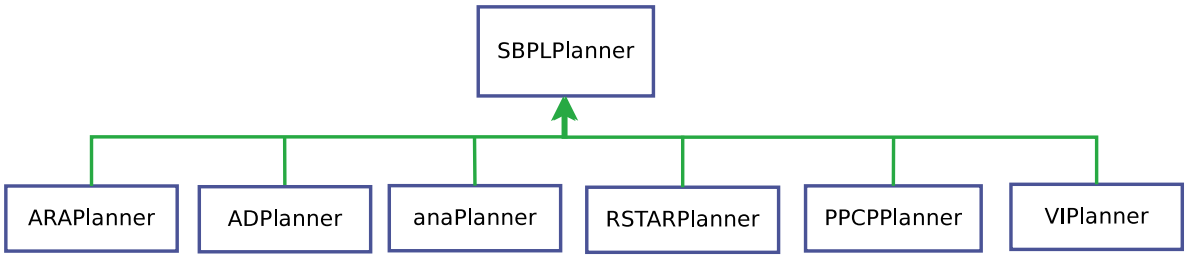


Figure 3.3: Hierarchy of existing Planner classes of SBPL

radians.

- **EnvironmentNAVXYTHETAMLEVLAT** This class also extends from the previous class, but now considers also the elevation (x, y, z, θ), by increasing the number of levels of z a projection of the robot in height is used to avoid collision.
- **EnvironmentROBARM** Implements an environment for a planar kinematic robot arm with variable number of degrees of freedom.
- **EnvironmentXXX** This is a template class that can be used to create a personalized environment.

For the planner the class hierarchy is shown on Figure 3.3

The SBPLPlanner class is pure virtual class for a generic planner, every planner inherits from this class, the rest of the planners are mentioned below.

- **ARAPlanner** ARA*, i.e., Anytime Repairing A*, is an anytime heuristic search which tunes its performance bound based on available search time.
- **ADPlanner** AD*, i.e., Anytime D*, is an anytime incremental replanning algorithm.
- **anaPlanner** ANA*, i.e., Anytime Nonparametric A*, adaptively reduces ϵ to expand the most promising nodes per iteration.
- **RSTARPlanner** R*, i.e., randomized A*, depends less on the quality of guidance of the heuristic function. It tries to execute a series of short-range weighted A* searches toward goal states chosen at random and reconstruct a solution from the paths produced by these searches [27].
- **PPCPPlanner** PPCP, i.e., Probabilistic Planning with Clear Preferences, is applicable to the problems for which it is clear what values of the missing information would result in the best plan. In other words, there exists a clear preference for the actual values of the missing information [28].

For this thesis we focus on the lattice Environments which are explained in detail in the next section, and for the planners we focus on the ARA* and AD*.

3.1.1 State Lattice

As explained on Section 2.3, all search algorithms require a discrete representation of the environment, for which all possible states are discretized, and it could be assumed that two adjacent states have a valid transition between them. This assumption is oversimplifying the problem, since it does not reflect the kinodynamic constraints of the robot, therefore SBPL introduces lattice-based graphs, constructed using motion primitives, leveraging on the precomputed motions that a robot can take.

State Lattice was introduced in [29], it is a discretized set of all reachable configurations of the system. The lattice is constructed by discretizing the C-Space into a hyperdimensional grid and attempting to connect the origin with every node of the grid using a feasible path. It will contain all the feasible paths up to a given resolution, meaning that if the vehicle can travel from one node to another then, the lattice will contain a sequence of paths to perform this maneuver.

In a sense, a state lattice behaves as a grid, since it converts the problem of planning in continuous space into one of generating a sequence of decisions chosen from distinct alternatives, but it does not encode default assumptions of how the states are connected. Thus, in order to encode nonholonomic constraints, it is important to capture the local connectivity of the state lattice inside a certain radius from the origin.

To this regard, a control set is defined as a finite subset of the state lattice that only includes paths from the origin out to some radius; it is desired to be a minimal set of primitive paths, that when concatenated can generate any other path in the lattice. Figure 3.4, represents the typical car-like motions, this set of motion primitives conforms the Control Set.

The state lattice is constructed by using an inverse path generator to find paths between any node in the grid and the origin. It has two important properties, that are used to formulate nonholonomic path planning as graph search:

- **Discretization**, converts problem of motion planning into a process of sequential decisions. Assuming that the decisions are made only at discrete states, then the states are the nodes in the lattice and the motions are the edges that connect the states. The state vector can have arbitrary dimensions, for instance if 3-dimensional then each node of the lattice represents a 3-dimensional posture that includes the 2D position and the heading (x, y, θ). The position can be discretized into a rectangular grid of a certain

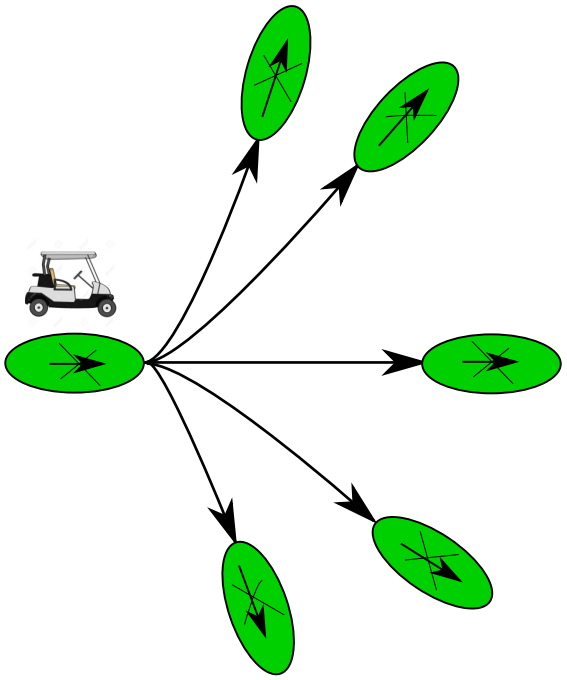


Figure 3.4: Precomputed motion primitives - Control set

separation Δ_l, any heading discretization is admissible, we can assume for the time being an uniform division of the interval $[0, 2\pi]$.

- **Regularity**, if the discretization has any degree of regularity then the state lattice will have the translational invariant property, meaning that it is possible to repeat the same motion in many nodes of the lattice. Considering that the lattice is regular in the translational coordinates (x,y), then the same motion that leads from state A to B can lead also from state C to D. Formally if the path between two postures is feasible (Equation 3.1)

$$\begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 \\ y_2 \\ \theta_2 \end{pmatrix} \tag{3.1}$$

Then it is also feasible the path of Equation 3.2 for any value of the integer k.

$$\begin{pmatrix} x_1 + k\Delta_l \\ y_1 + k\Delta_l \\ \theta_1 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 + k\Delta_l \\ y_2 + k\Delta_l \\ \theta_2 \end{pmatrix} \tag{3.2}$$

Figure 3.5 shows how the graph can be constructed using motion primitives. By super-

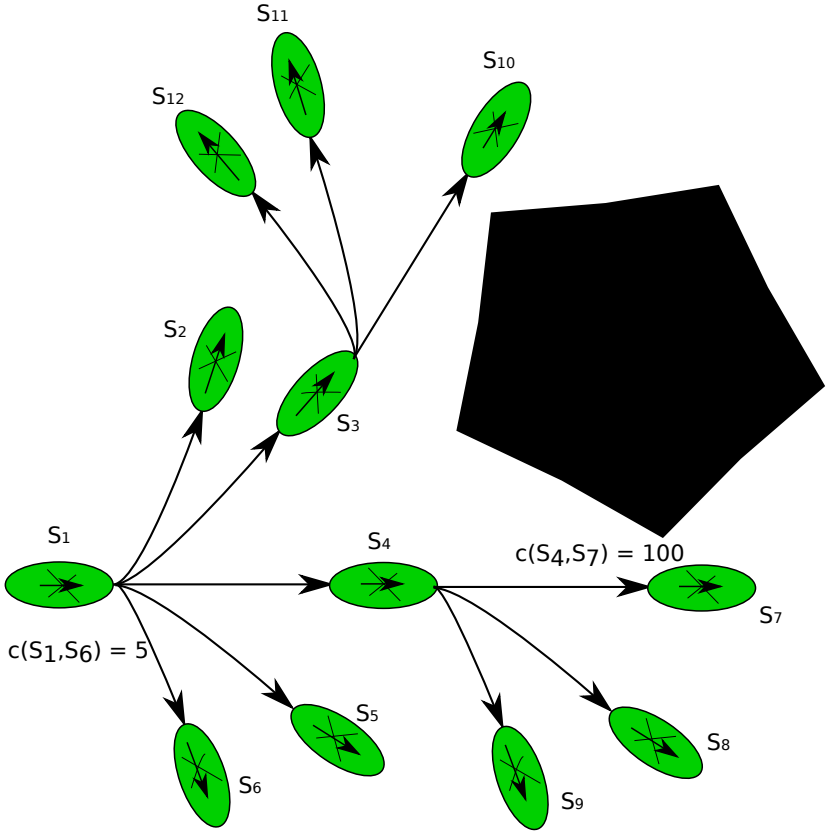


Figure 3.5: Online replication of the motion primitives

imposing the control set on the graph, the successor states represent what the robot can smoothly transition to. Valid successors of a given state are states that can be reached with these motions while not colliding with anything else.

Motion primitives allow to encode the kinematic constraints of the robot, and also allow to set different cost to each motion, meaning that if reversing for the robot should be avoided then a high cost can be assigned to that particular motion primitive. To deal with the creation of motion primitives, SBPL provides a Matlab tool to build motion primitives for different types of vehicles. For this thesis a modified version of the primitives generator file was used to account for the kinematic constraints of an Ackerman vehicle.

To summarize what has been explained on this section, Table 3.1 contains the pros and cons of working with a State-Lattice graph.

Table 3.1: State-Lattice pros and cons

Pros	Cons
Resolution complete	"Dimensionality curse". The number of states grows exponentially with the dimensionality of the state-space.
Optimal	State-lattice construction requires solving non trivial two-state boundary value problem.
Offline computations due to regular structure are possible	Regular discretization might cause problems in narrow passages not aligned with the hypergrid.
	Discretization causes discontinuities in state variables nor considered in the hypergrid, so motion plans are not inherently executable.

3.2 Open Motion Planning Library

OMPL is an open source C++ implementation of different sampling-based algorithms, it is aimed for users in the research of motion planning, as well as for robotics educators and end users in robotics industry [1]. As mentioned earlier in this thesis, the idea behind the sampling-based motion planning is to approximate the connectivity of the search space with a graph. The samples of the search space form the nodes of the graph and the edges are valid paths between the nodes. Then, in order to build the graph, two things must be taken into consideration, the probability distribution used to sample states and the strategy for generating the edges.

In order to deal with the different planning issues, OMPL was designed as a set of components shown in Figure 3.6, this high-level overview of OMPL structure shows the main classes and their relationships. The following classes are analogous to known concepts in sampling-based path planning, as explained in the OMPL Primer.

- **State Sampler**: The sampler is needed to generate different states from the state space. This class implementation provides methods for uniform and Gaussian sampling, in common state-space configurations. The library includes methods for constructing Euclidean spaces, 2D and 3D space rotations SO(2) and SO(3), or the combination of SO(3) and $\mathbb{R}^3$, which is SE(3), and its lower-dimensional analogue SE(2). For complex systems, enumerating all possible state spaces can be unfeasible, if they can be decomposed into smaller spaces, then OMPL allows to construct these spaces using the CompoundStateSpace class. This class requires to know if a sampled state is valid (ValidStateSampler), which uses the StateValidityChecker.

- **StateValidityChecker**: Aims to determine if a state collides with an obstacle (or does not comply the constraints of the robot). Since it is an integral part of the sampling-based algorithms, a default checker is not provided by OMPL, the user must define it, and provide a callback to the isValid method.
- **NearestNeighbors**: Used to provide an interface for the planner to perform a nearest neighbor search. The core library includes Geometric Near-neighbor Access Trees (GNATs) and linear searches. It can also use an external data structure.
- **MotionValidator**: Analogous to the local planner, it aims to check if the motion between two states (tree nodes) is valid, meaning that the path is obstacle free, and it respects the motion constraints. The default class in OMPL is the DiscreteMotionValidator, which uses interpolated states between the two states and using the StateValidityChecker verifies that the motion is valid. This operation can be modified by the user.
- **OptimizationObjective**: Provides the interface for the cost operations required by the planner. In general the used objective is the PathLength, which aims to minimize the length of the overall path.
- **ProblemDefinition**: This class defines the start state, the goal state, and the optimization objective. To retrieve the solution, this class is used.

OMPL Planners are divided into two categories, Geometric planners and Control-based Planners. The first category only accounts for geometric and kinematic constraints of the system, on the other hand the second category is used when the system is subject to differential constrains, therefore these planners will rely on a StatePropagator that is in charge of computing how a system state changes when applying valid controls, instead of just a simple interpolation.

The geometric planners are divided in three categories, the first two already mentioned in the second chapter; i.e., Multi-query planners and Single-query planners, and the Optimizing planners category, whose planners can overlap with the other two categories.

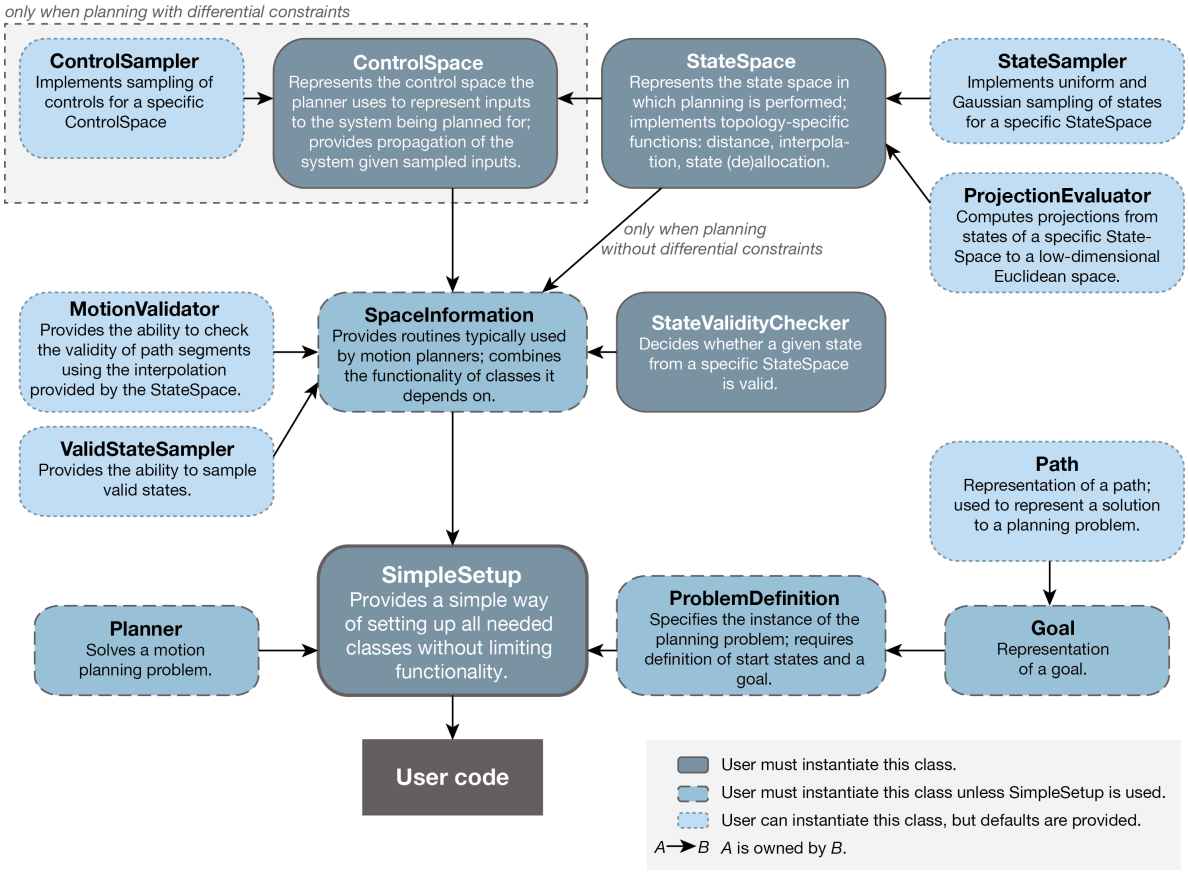


Figure 3.6: Overview of OMPL structure [1]

3.2.1 Setting Up a Planning Context

To set up a planning context in OMPL four main objects are required:

- **Space**: For geometric planning only the StateSpace is required, for this the bounds of the space must be specified. If planning with differential constraints, also a representation of the space of controls is needed.
- **Space Information**: The SpaceInformation is conformed by the StateSpace and ControlSpace if needed, plus the classes needed to check validity of the state and motion. By default OMPL takes all states as valid. Therefore this must be defined by the user. To describe motions the DiscreteMotionValidator is used as mentioned before, but the user must define the maximum distance between states to be checked for validity along a path segment (using the method setStateValidityCheckingResolution).
- **ProblemDefinition**: Setting the start and goal states can be done by calling the set-

StartAndGoalStates method, by passing as arguments the start and goal state defined.

- **Planner**: The planner requires to have available an instance of the SpaceInformation, which is passed to the constructor. Once created the setup method is called. For the planning operation the solve method must be called.

OMPL provides the possibility to encapsulate various of the objects previously mentioned by using the SimpleSetup Class, without inhibiting the native functionalities of OMPL. This class takes care of instantiating objects like SpaceInformation and the ProblemDefinition. It makes sure that all the objects are properly created before the planning operation begins.

Furthermore, there is also the possibility to use OMPL.app, which is a graphical front-end the OMPL library. This GUI is written using Python and PyQt. It comes packaged with some mesh models for 2D and 3D environments and robots, for the user to start experimenting with solving a motion planning query. Examples of how to use it can be found in the OMPL Primer [30].

3.3 ROS

ROS has been developed by Willow Garage and Stanford University as a part of STAIR project, as a free and open-source robotic middleware for the large-scale development of complex robotic systems. It is an open-source, meta-operating system for robots. It provides the services like an operating system, including hardware abstraction, low-level device control, message-passing between processes, and package management [31]. Nowadays it is widely accepted and used in the robotics community. Its main goal is to make the multiple components of a robotics system easy to develop and share so they can work on other robots with minimal changes.

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. For our purposes we require to understand the Computation Graph Level, which is the peer-to-peer network of ROS processes. Its basic elements are shown on Figure 3.7 and explained below.

- **Nodes:** Nodes are processes that perform computation; ROS is designed to be modular, therefore a robot system is conformed by many nodes, which separates the code and functionalities, making the system simpler. They are written with the use of a ROS client library as roscpp or rospy.

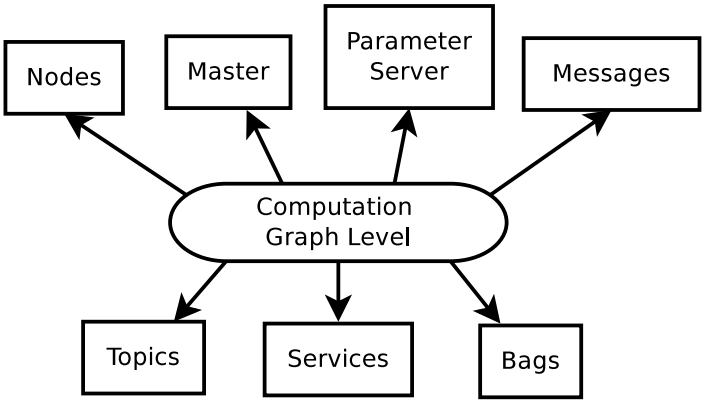


Figure 3.7: ROS Computation Graph Level

- **Master:** The ROS master provides naming registration and lookup to the rest of the Computation Graph services. The role of the master is to enable individual ROS nodes to locate each other. Once these nodes have found each other, they communicate in a peer-to-peer fashion.

- **Parameter Server:** The parameter server allows data to be stored in a central location. With the parameter server it is possible to configure the nodes while running or to change the working parameters of a node.

- **Messages:** Nodes communicate with each other by passing messages. A message contains data that provides information to other nodes, it is a data structure comprising typed fields.

- **Topics:** Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message, therefore a node that is interested in a certain kind of data will subscribe to the appropriate topic. When a message is received on a topic, a user callback function is called, this function is in charge of managing the obtained data; the performed action can even be the simple storage of the data.

- **Services:** When a topic is published,the data is sent in a many-to-many fashion, this model is not appropriate for request/reply interactions, instead it is done via a Service, which is defined by a request message and a reply message. A node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

- **Bags:** Bags are the primary mechanism for data logging in ROS, they allow to record datasets, visualize it, label it, and store for future use. The store data can be played

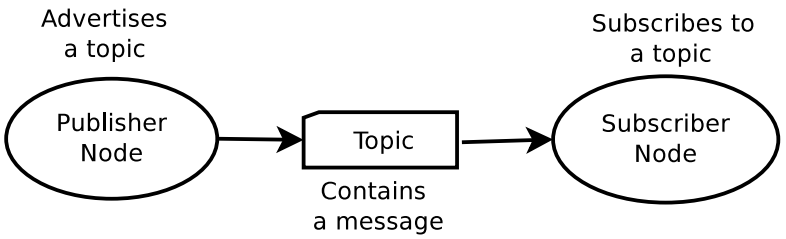


Figure 3.8: ROS Communication

back and use as if a ROS node was sending it.

Figure 3.8 shows how the communication happens with a publisher and a subscriber node, the publisher advertises a message in a topic and the other node will subscribe to the topic to obtain the message.

In ROS, there is also the possibility to use Timers, they let you schedule a callback to happen at a specific rate; timers are not to be intended as real-time threads. To use a timer the user has to start it (can be done during a callback of a topic for instance); once the timer has been started a callback function is executed, when the operation is finished the timer must be stopped so that it can be started again later on.

Another important aspect about ROS is that it comes with a 3D visualizer called RViz (ROS visualization), the idea behind RViz is to provide visualization displays for different ROS messages, it is easy to use, the user only needs to select a display either by the message type, or search also by the available topic names. For navigation several displays can be used as Map, Path, Pose, Odometry, TF, etc. Each display has some options that can be setup like the color, and the topic to subscribe. The visualizer also allows to save a configuration to be able to use it for a specific application, so there is no need to setup the visualizer every time. Figure 3.9 shows an example for setting up RViz displays.

3.4 Navigation Stack

For mobile robot navigation several tasks are required to solve three main problems: mapping, localization and path planning. ROS has a set of resources that are useful so a robot is able to navigate through a known, partially known, or unknown environment; using these a the robot is capable of planning and tracking a path while it deviates from obstacles that appear on its path throughout the course. These resources are found on the Navigation Stack [32].

The Navigation Stack Setup can be seen on Figure 3.10, the move_base node subscribes to odometry, sensor data, and goal position messages and produces velocity commands to be send

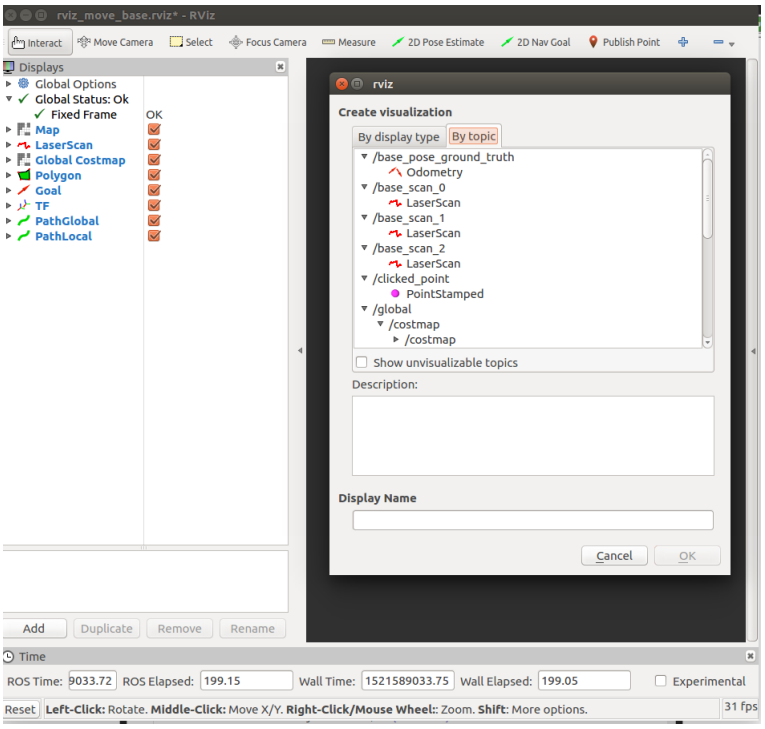


Figure 3.9: ROS visualization

to the mobile base. It can also subscribe to a map, so it can generate a better path with known information of the environment; the localization part is done by the Adaptive Monte Carlo Localization Algorithm. The Navigation Stack main component is the move_base package, it is divided into global and local path planning modules, and it maintains a global and local costmap, used by the respective planners, which keep the information of the obstacles in the environment in the form of an occupancy grid. The global costmap is initialized with a static map, if one is available, and then it can be updated with data coming from the sensors.

For any global or local planner to be used by the move_base package, they have to adhere to some interfaces defined in the nav_core package. The global planner must adhere to the nav_core::BaseGlobalPlanner interface and the local planner to the nav_core::BaseLocalPlanner interface; also they must be added as plugins to ROS in order to work with the navigation stack.

The current interfaces provided by the Navigation Stack are listed below [33]:

- **Global Planners**
  - **global_planner:** Fast interpolated global planner built as a replacement to the older navfn. This one can use the Dijkstra algorithm or the A*, or if required it

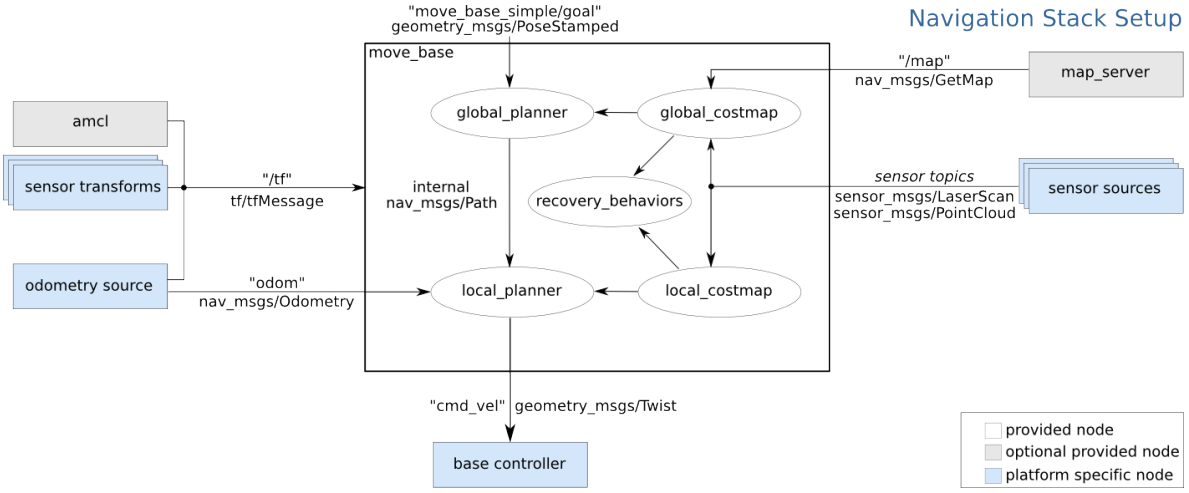


Figure 3.10: Navigation Stack Setup

can also behave as the old navfn. The behavior is set through the parameters.

- **navfn:** Grid-based global planner that uses a navigation function to compute a path for a robot.
- **carrot_planner:** Simple global planner that takes a user-specified goal point and attempts to move the robot as close to it as possible, even when that goal point is in an obstacle.

- **Local Planners**
  - **base_local_planner:** Provides implementations of the Dynamic Window and Trajectory Rollout approaches to local control.
  - **eband_local_planner:** Implements the Elastic Band method on the SE2 manifold.
  - **teb_local_planner:** Implements the Timed-Elastic-Band method for online trajectory optimization.

One of the problems of using the navigation stack is that is not very flexible, or easy to configure, for instance the current implementations of the global planner in ROS assumes a circular-shape robot; this could lead to create unfeasible path for the actual robot footprint. Besides this, the available move_base global planners ignore kinematic and acceleration constraints of the robot, so the generated path could also be dynamically unfeasible.

Chapter 4

Global Planners Design and Advances

In this chapter, first we explain the design of our modular packages, by describing the structure and the used architecture for our two global planner nodes; the Search-Based Global Planner and the Sampling-Based Global Planner.
Then, we go into details for each of the nodes. For the first one we used the SBPL library, thus, we clarify the modifications perform to the library; change of the cost to optimize and use of the manual refinement. As well, the required configurations to initialize the environment and planner are explained. For the second planner, the OMPL library was employed, we divide it in two subsections, the first one explains the configurations required to initialize the planners, as well as the usage of the Dubins and Reeds-Shepp spaces. The second subsection contains the modifications performed to the work by Basak, RRT*_MotionPrimitives, and our three novel proposals for sampling the space with our motion primitives.

4.1 Global Planner Node Structure

Both the SBPL and OMPL global planner nodes created have the same basic structure regarding the topics they subscribe and publish, as shown on Figure 4.1.

The transformation /tf, is a ROS package that lets the user keep track of multiple coordinate frames over time, for the global planner it is needed to obtain the current position of the robot in the environment (map). A simulator node like Stage can provide the different transformation of the robot (/odom, /base_link), and a static transformation can be used to set the relation between the map and the odom frames, so the transformation from /map to

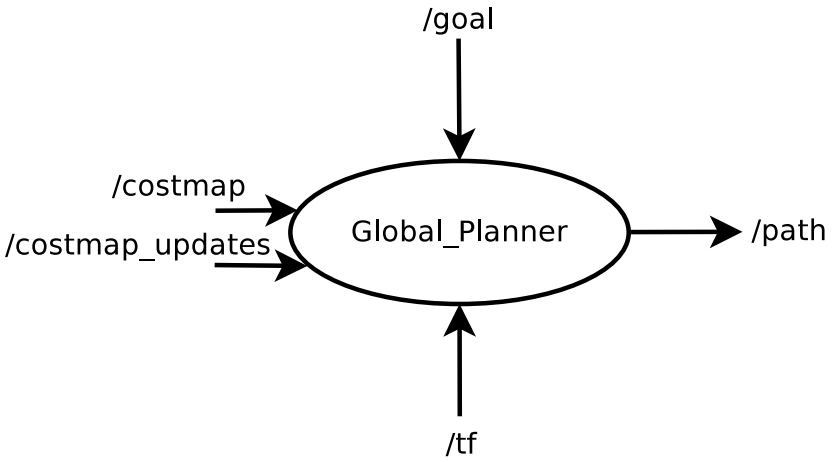


Figure 4.1: Global Planner Structure

/base_link can be obtained.

The /goal topic, carries the message of the 2D Pose (position and orientation) of the state to reach with the planning, this topic can be provided by an external node, for simplicity we use the ROS visualization node Rviz, which has a tool that publishes a 2D Nav Goal topic (/move_base_simple/goal), so that from the visualization window an arrow can be set inside the map, to represent the goal state.

TO obtain the costmap and its updates a map of the environment is required, for which ROS map_server node is used, this node receives a yaml file, its most important information is listed bellow.

- image: Path to the image file that contains the occupancy data. Generally a .pgm file.
- resolution: Defines the resolution of the map, meters/pixel.
- origin: 2D pose of the lower-left pixel in the map (x, y, θ).

The map_server node takes the given information and publishes a map in the form of an Occupancy Grid (nav_msgs) in the topic /map, which uses the values 0 to represent a free cell, 100 for occupied and -1 for unknown.

Then a costmap_2d_node is used, named global_costmap. It is composed of three layers, a static one, which takes the information from the /map topic, an inflation layer which inflates the obstacles from the map, and an obstacle layer which takes the information from a laser scan to account from obstacles not represented in the map. In order to register the obstacles in the map frame and update the costmap, a localization system is required, so the costmap

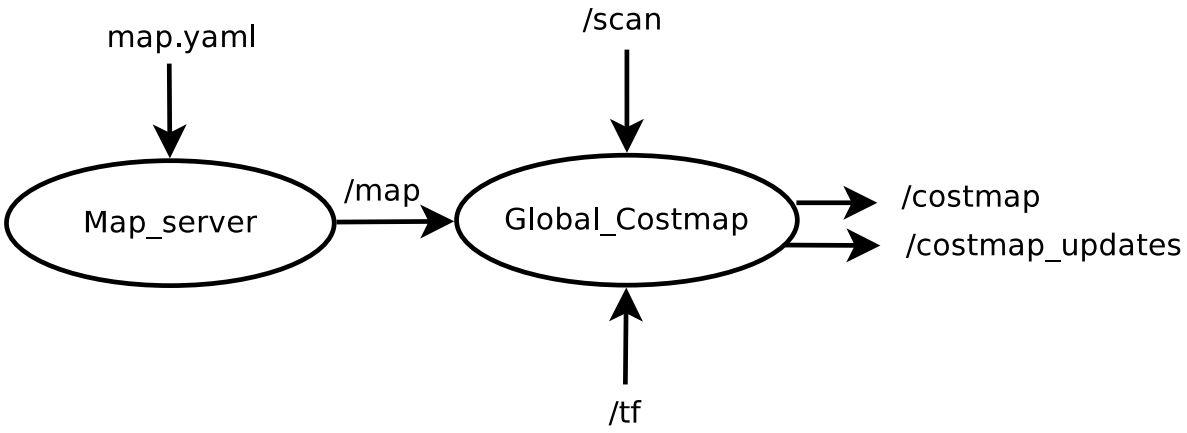


Figure 4.2: Global Costmap Node

node also subscribes to the transformations topic /tf. The costmap is setup using a configuration file.

Next section explains the architecture used for both global planners.

4.2 Global Planner Nodes Architecture

The main structure of our global planners nodes was automatically generated from an architecture model by Gianluca Bardaro, using Architecture Analysis and Design Language (AADL). AADL provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions. Therefore, the idea behind this is to exploit model-based approaches to create description of a robot architecture and use the model for automatic code generator. In [2] Bardaro presented the ros_base::ROSNode class from which our nodes extend.

A graphical representation of the base node is shown in Figure 4.3, it has an internal state machine with a node life cycle containing four different states, *init*, *running*, *closing* and *error*. The transitions between the states are triggered by events. The *main_thread* in the figure shows the main execution flow of the node, and it uses an asynchronous spinner.

The subprograms of the main thread are explained below:

- prepare: Active during the *init* state, it is in charge of initializing the parameters, and we have defined here the subscribers, timers and advertisement of the publishers needed for our planners.

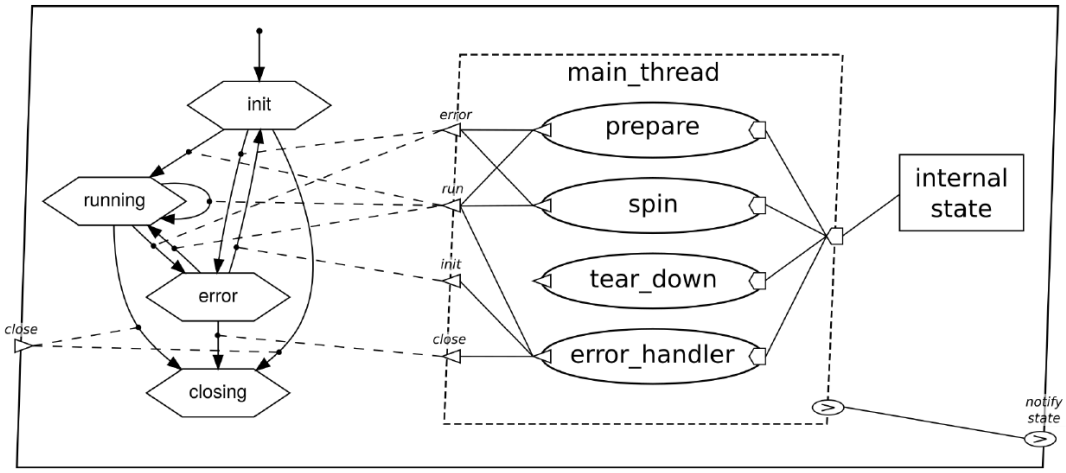


Figure 4.3: Graphical Representation of the base node [2]

- spin: This subprogram manages the callbacks (from the subscribers and timers), it is enabled during the running state, and it is the core part of the developed planners.
- tear_down: This subprogram manages the termination of the node, it is enabled in the closing state, which can be triggered by either an external or internal signal.
- error_handler: If an error or faulty behavior is experienced then the state machine changes to *error* and error_handler is enabled.

Further information of how to automatically create ROS C++ code with this methodology can be seen in [2]. Next the characteristics of the planner callbacks are explained.

4.2.1 Callbacks for the Developed Planners

The callbacks are the functions that are executed once an event occurs, the event can be the arrival of a new message from a topic our node is subscribed to, or the start of a timer. They are managed by the asynchronous spinner.

Costmap Callback

The first message our nodes need to receive to start is the costmap. The costmap node will publish the costmap once a node has subscribe to it. Once the message is received the costmap_cb_callback runs, calling the saveCostmap function.
The saveCostmap function initializes a new costmap_2d object with its values as Obstacles

(254), it then takes the OccupancyGrid message with values from -1 to 100, and translates these values onto the usual costmap values from 0-255, to fill the new costmap, having 0 as free space, 254 as Obstacles and 255 as Unknown, the values in between depend on the distance from an obstacle cell, as can be seen in [34] where the Inflation layer is explained. Finally the initialize function is called. This is explained specifically for each planner later on.

Costmap Updates Callback

This callback is related to the subscription to the map message OccupancyGridUpdate, which provides the changes in the map thanks to the information coming from the laser sensors; it allows to update the map in case of new obstacles or new free space in the environment.

In case of Search-Based planners it checks which cells from the grid have changed values so as to inform the planner that the costs have changed; this is used by the AD* planner, that can leverage on the information about the changed cells and update the predecessor of the changed edges.

Goal Callback

As explained on Section 4.1, our nodes subscribe to a Pose topic. When a pose is sent from Rviz, or some other node, the goal_cb_callback runs the function SaveGoal, which takes the message and saves it in the respective (x, y, θ) variables. Then it starts the timer: timer_path_out_pub, this timer enables the Path Publish Callback.

Path Publish Callback

This callback is enabled when the timer_path_out_pub starts. It runs either the sbplPlan or omplPlan function returning the found path message to be published. Once the path has been published the timer is stopped so it can be restarted when a new goal arrives.

Figure 4.4 shows a flow diagram of the explained elements of the global planner nodes.

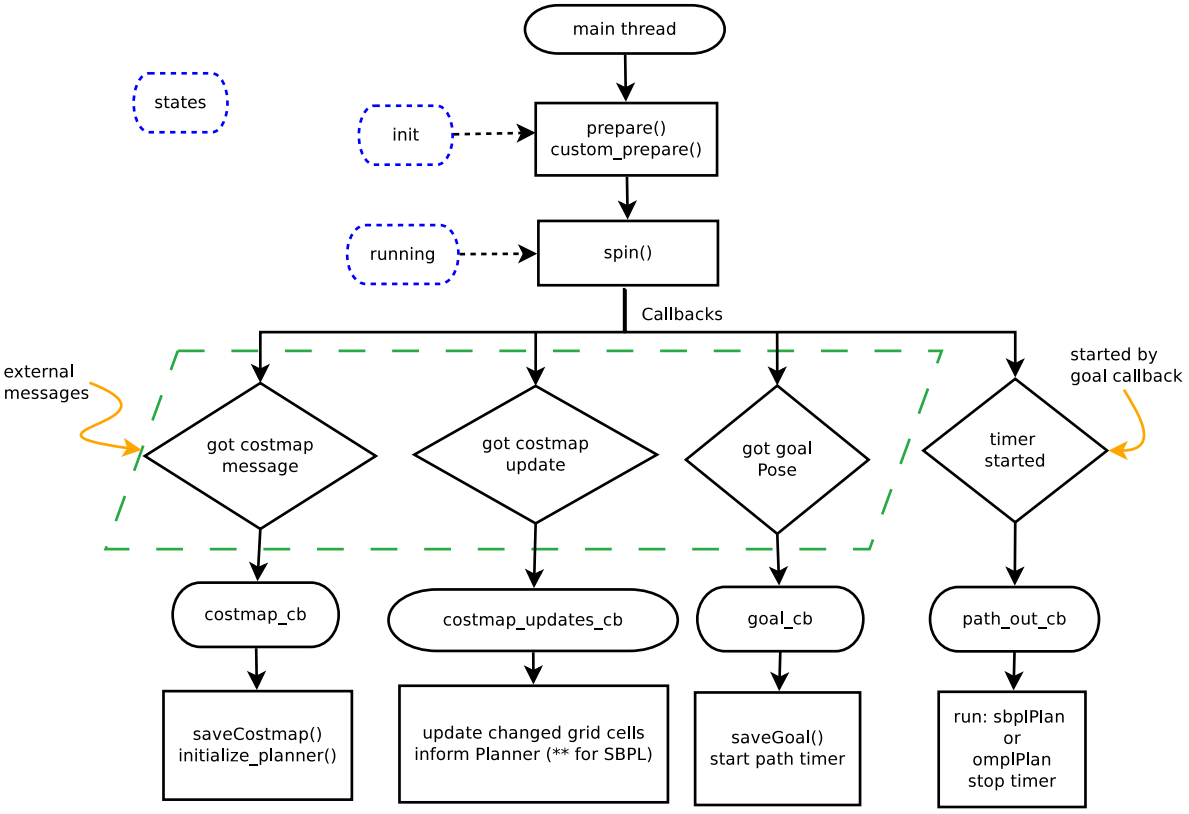


Figure 4.4: Flow diagram of the global planners

4.3 Motion Primitives

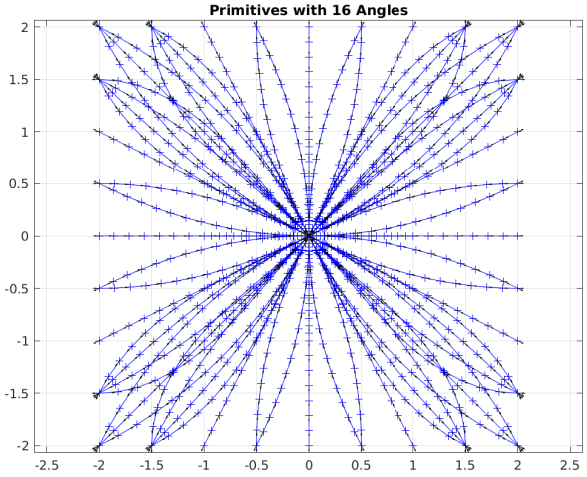
Before describing the details of both global planners the Motion Primitives used are shown in this section.

This thesis was not focused on the creation of motion primitives, since it had already been pursued in thesis like [23] and [35], we have used an existing script to generate a text file containing the information of each primitive. The following information is required by the script:

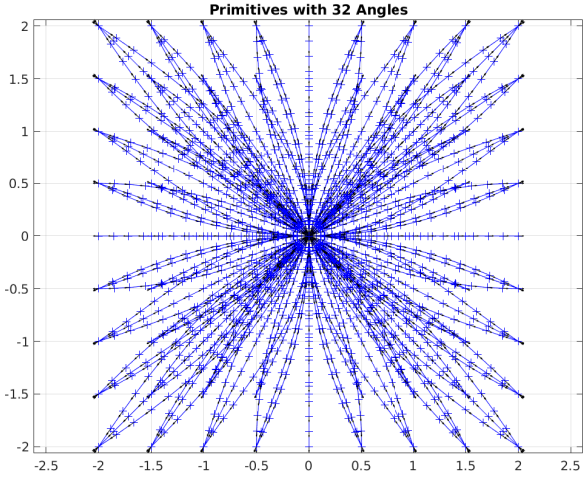
- wheelbase: Distance between the centers of the front and rear wheels, given in meters.
- steering_angle_max: The maximum angle the vehicle can steer.
- number of angles: Number of angles to discretize the angles $(-\pi, \pi)$
- resolution: it must coincide with the discretization of the space in the costmap for better results.

- max_distance: Maximum length of the primitives, i.e., traveled distance.

For our application the used primitives have a resolution of 0.25m and are discretized in 16 and 32 angles, with a maximum length of 2m and a maximum steering angle of 35°. The primitives can be seen in Figure 4.5



(a) Primitives with 16 Angles



(b) Primitives with 32 Angles

Figure 4.5: Motion primitives with 0.25m resolution

4.4 SBPL Global Planner

As explained in Section 3.1, SBPL has two objects the environment and the planner. The first step for the design of a path planner, is to initialize the environment, this is performed as shown in Figure 4.6.

First the perimeter of the robots is computed, then some parameters are taken from the primitives file, as the number of angles, and the resolution (cell size in meters). Next an InitializeEnvironment function is called which takes the char map from the previously saved Costmap, as well as the origin, and size of the environment, to pass it to the InitializaEnv function from SBPL. This last function is in charge of setting the Configuration of the environment, meaning it takes the costmap and initializes the grid, sets the nominal linear and angular velocities provided to internal variables, as well as the perimeter and resolution.

It also computes the cost of every primitive called *actions* in the SBPL library; in the original implementation of SBPL to compute the cost, first the linear distance and angular distance are calculated, then by means of the max angular velocity and linear velocity, the time to traverse the primitive is computed. Once the times are calculated the cost is computed as an additional cost multiplier given by the user in the primitives file, and then multiplied by 1000 to pass from m to mm, as follows:

```
cost = add_cost * COSTMULT_MTOMM * max(linear_time, angular_time);
```

We have modified the cost, since our goal is to find the shortest path, as the linear distance of the primitive:

```
cost = COSTMULT_MTOMM * linear_distance;
```

The reason why the initialization function has been splitted in these three sections, is to facilitate the selection of different kinds of environments (we have anticipated the use of an environment with 5 variables, in case primitives with linear and angular speeds are available).

Once the initialization has been done, the system is ready to receive a goal callback and start planning. The first task to do is get the Robot Position, and set the Start and Goal States to the environment. Next, the PlanWithSBPL function is executed, in this section the initial epsilon is set, as well as the decreasing step and the search mode (backwards or forward). Once this is done the replan function of SBPL is called, it takes the allocated time to run the planner, and returns a vector with the StateIDs of the solution if one exists. The replan function calls the planner Search function that runs until the final epsilon is reached or

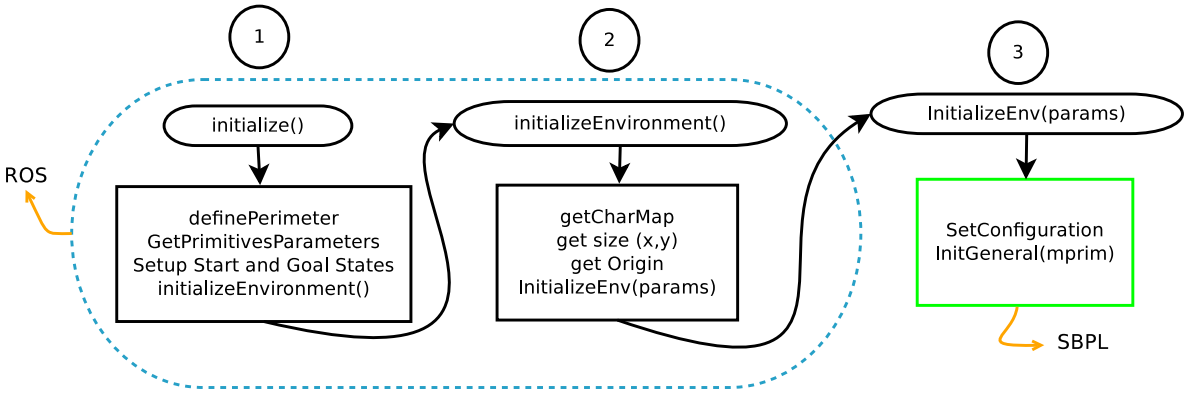


Figure 4.6: SBPL Environment initialization

until the time is up. Once a solution has been found the StateIDs are converted to XYTheta path, so the nav_msgs Path can be filled and published. As seen in Figure 4.7. If a path was not found the message is published empty, and a new goal can be set.

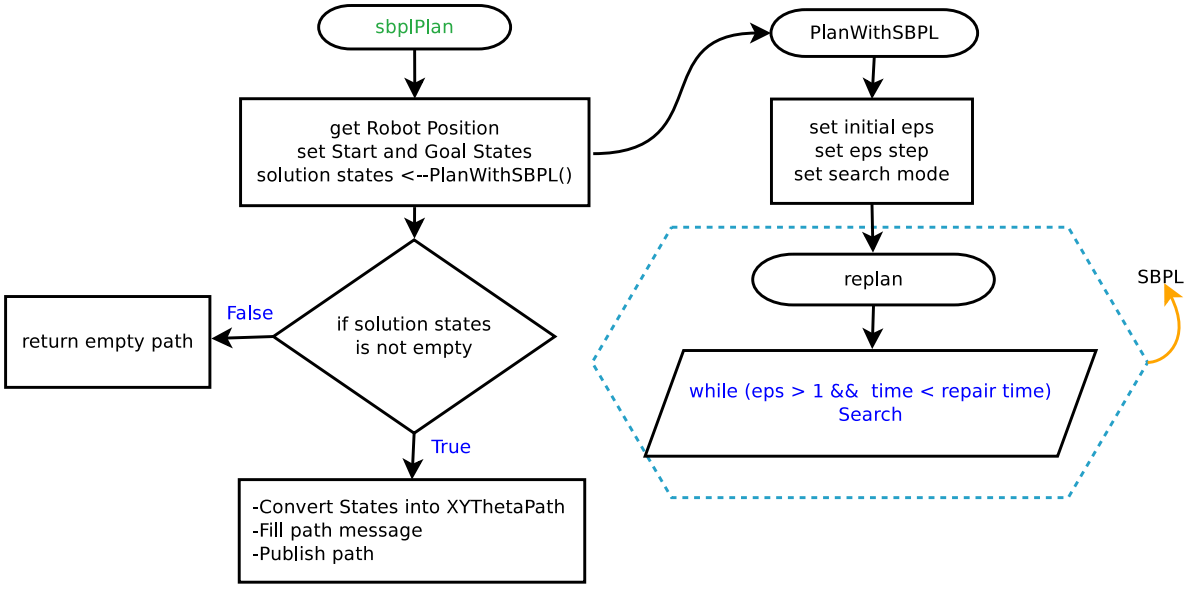


Figure 4.7: SBPL Planning

Regarding the search mode, SBPL gives also the possibility to return the first solution, with the initial epsilon. Therefore by using this, we have defined a Flag to make a manual replanning, we manually decrease epsilon, and run the replan function of SBPL with the option to return the first solution, which returns a suboptimal path. We keep decreasing epsilon until we get $\epsilon = 1$ (until we obtain the optimal path). This allows the user to observe how is the refinement of the path done with SBPL, and to decide on the initial epsilon value

as well as the decrease step of epsilon.

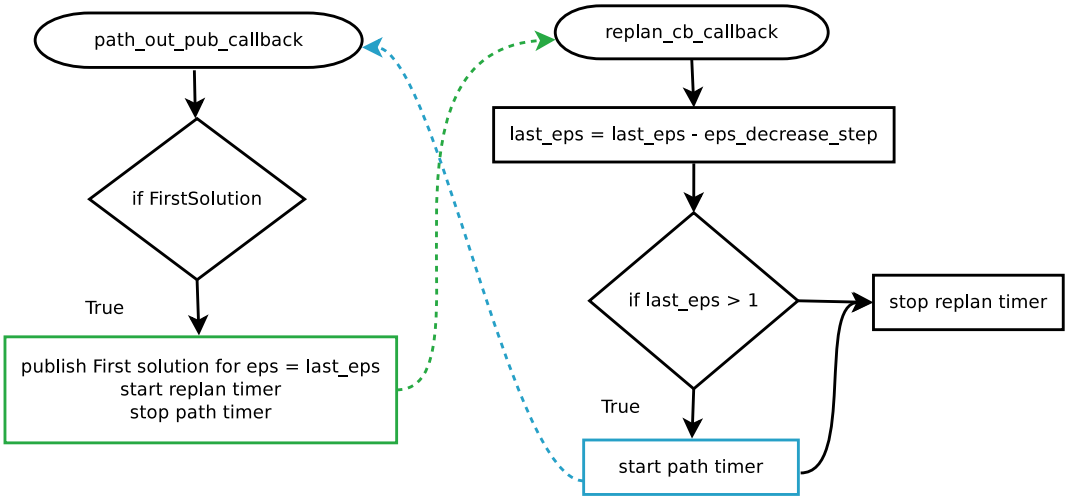


Figure 4.8: SBPL Manual Refinement

The rest of the customizable parameters of the planner are explain in the User Manual in Section A.3.1

4.5 OMPL Global Planner

To explain the OMPL global planner design first the normal implementation of the existing OMPL planners is shown, then the OMPL+MP implementation is explained; they have been separated because both the initialization and the planning are different, even though both use the SimpleSetup implementation mention in Section 3.2.

As with SBPL, first the specifications of the environment must be set before a path request is performed. For this task an initialization function takes place (Figure 4.9).

In the usual OMPL implementation an SE2 Space is set, for setting its boundaries the information from the map is taken into consideration for the (x,y) coordinates, while for θ the bounds are set from $(-\pi, \pi)$. To deal with kinematic constraints OMPL offers implementations of the Dubins path (shortest curve that connects two points in the xy plane with a constraint on the curvature set by the wheel base and the maximum steer angle of the vehicle) and the Reeds-Shepp curves (the difference with Dubins is that it accepts reverse direction). The Dubins and Reeds-Shepp space extend from SE2, therefore the bounds do not change, the only requirement is to set the maximum turning radius of the vehicle.

Once the space has been configured SimpleSetup is instantiated. Important elements that must be setup are:

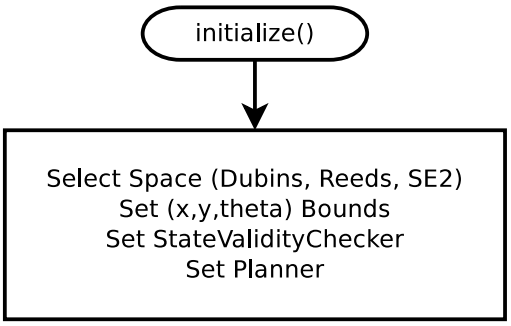


Figure 4.9: OMPL Initialization

- The State Validity Checker Function: a function is set to check if a state is valid. This same function is used to try connect two states with a trajectory.
- The Longest Valid Segment Fraction: This is important so the state space can compute how many segments the motion should be split, so that the number of states checked for validity are satisfactory (a Discrete Motion Validator is used).
- The planner: Different planners have been tested, in general RRT and PRM variations.

For both implementations i.e., with and without motion primitives, the same isStateValid Function was used. As shown in Figure 4.10, it first checks if the State is within the bounds of the Space, if so, then an oriented version of the Footprint is computed and by using the footprintCost function from the CostmapModel in base_local_planner it is determined if the Footprint is completely in Free Space.

Figure 4.11 depicts the possible outcomes the footprintCost function could return for different situations.

- If center cell is inside an obstacle then the cost $= -1$.
- If some points in the perimeter are inside an obstacle then the cost $= -1$.
- Else if all points are in free space a value > 0 is returned

After the Space and the StateValidityChecker have been set, the planner can solve a query.

The planning starts by getting the current robot position, then the previous planner data is cleared to start the new plan, in the case of PRM planners we do not want to clear the Roadmap, instead only the information of the query is cleared. To obtain a plan the solve function is called, it receives as parameter the planningTime, once a solution is found we check if the solution is Exact meaning that the path has exactly arrived to the goal pose, otherwise

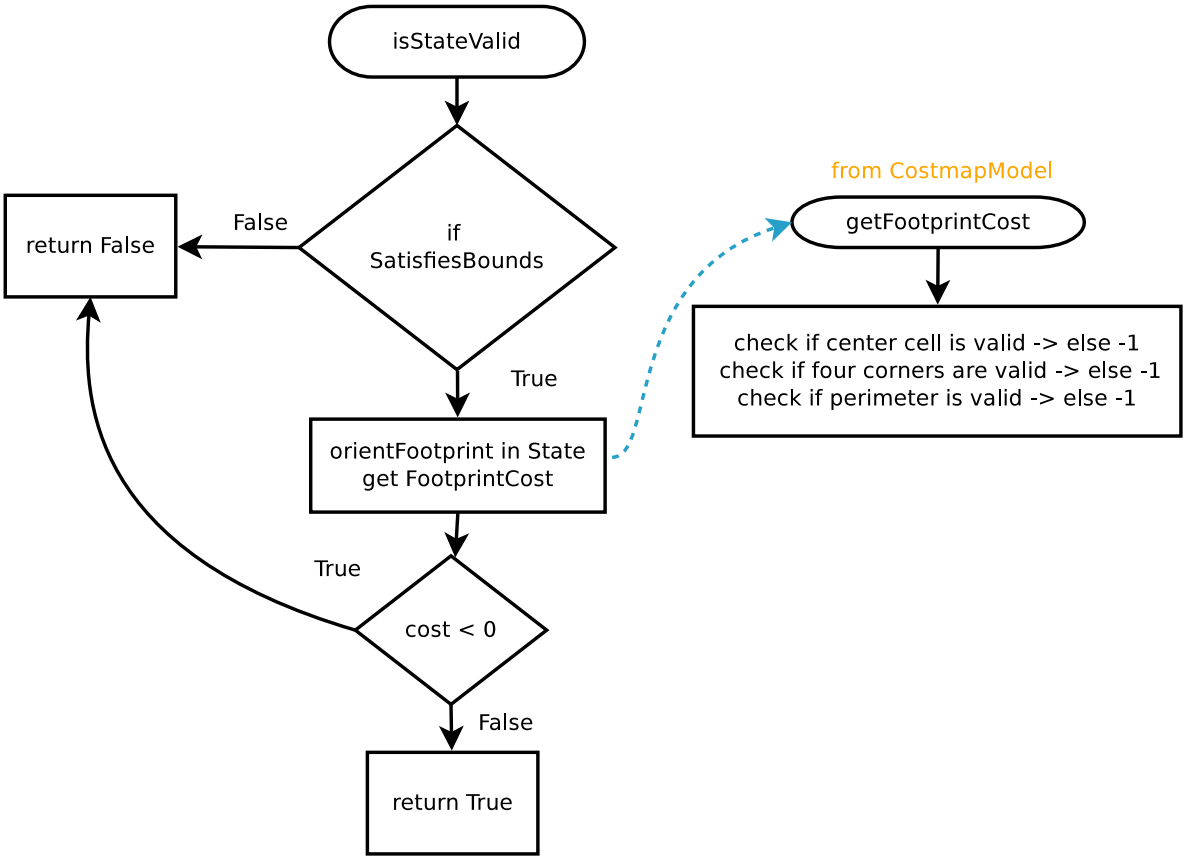


Figure 4.10: isStateValid Function

the solution is inside the goalRegion (given by a threshold value from the goal), if not, we try to solve again with more time, for a number of trials set by the user. If a solution is found then the Solution Path and states are stored, and the path message is filled and published. The flow diagram is shown in Figure 4.12.

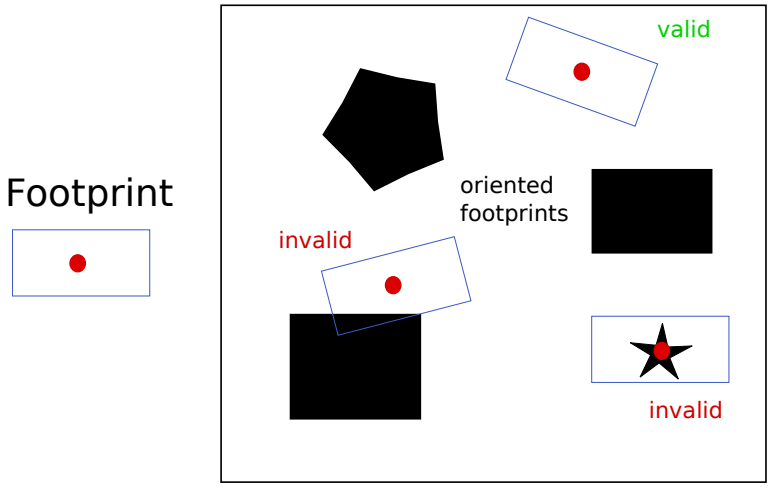


Figure 4.11: Footprint Cost

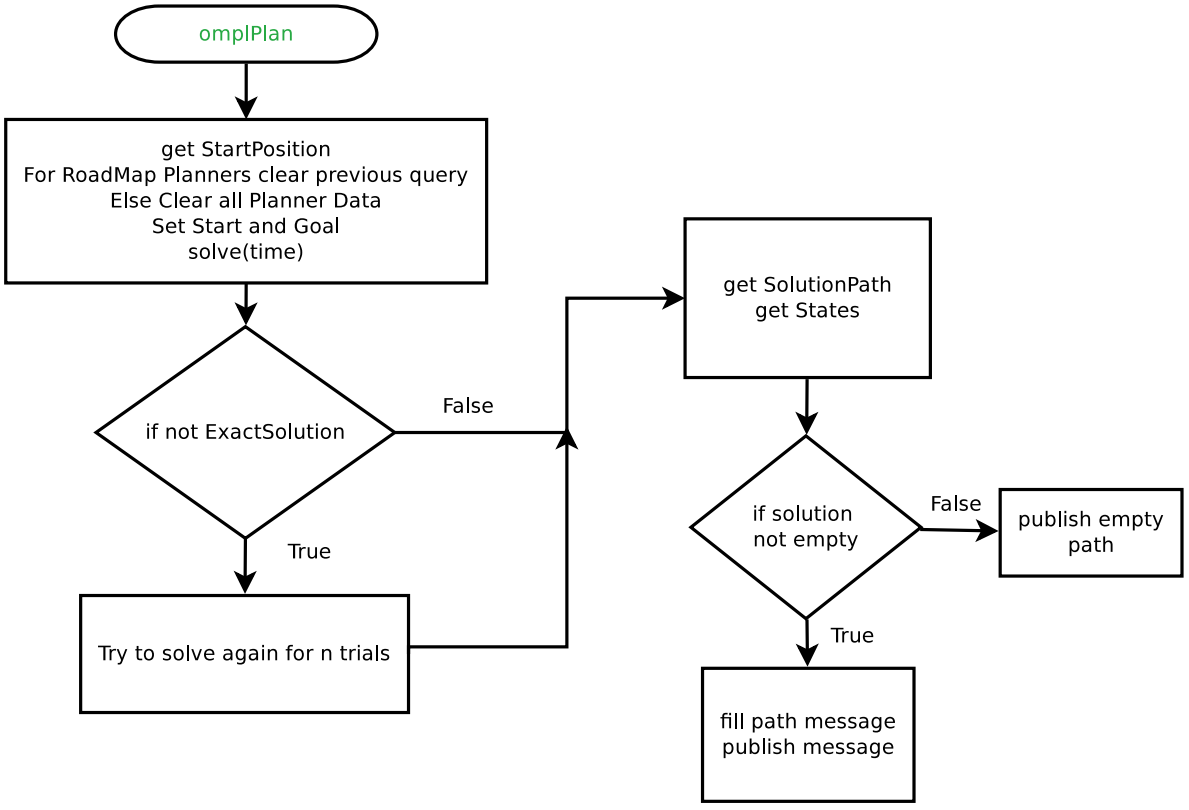


Figure 4.12: Ompl Planning

4.5.1 OMPL+MP

As mentioned before the Dubins or Reeds-Shepp paths do not give enough constraints as to construct good paths, then we decided to also include motion primitives as in SBPL to perform the path planning.

To do so the implementation of RRT* with Motion Primitives done by Basak Sakcak [4] was modified. Let's first mention the original implementation to later explain the different modifications and the parts that were aggregated. Basak's implementation can be divided into 5 elements:

- SampleSet
- StateSamplerFromSet
- MotionPrimitivesStateSpace
- AsymmetricNearNeighbours
- RRTStarMotionPrimitives

Starting from the Space, it is defined with four variables (x, y, θ, v), as an area of 10x10 meters discretized in steps of 0.5m, for theta it was defined $(0, 2\pi)$ in steps of $\pi/4$, while for speed from 0m/s to 4m/s, in steps of 2m/s. Then the SampleSet is basically a matrix containing all the combinations of the 4 state variables.

Basak defined a StateSamplerFromSet class, which contained the sample function, this function is in charge of randomly taking one state from the previously mentioned SampleSet, check if it is valid, and return it, otherwise it is removed from the SampleSet, and a new random sample is taken until a valid one is found.

The MotionPrimitivesStateSpace, was defined as a 4 dimension space, from this class the most important methods related with the primitives to mention are distance() and getTrajectory(), since they are used by the planner RRTStarMotionPrimitives to solve the planning query.

- distance(): This function takes two states, and verifies if there exists a motion primitive that connects the states, if it exist it returns the cost of the primitive, otherwise it returns a $cost = 10000$.
- getTrajectory(): this function receives two states, checks if there is a primitive to connect them, and returns a vector with the intermediate states that compose the motion.

The AsymmetricNearNeighbours class is a datastructure for asymmetric motion, it contains the methods for adding and removing nodes from the tree, as well as the nearestR function, which returns a sorted vector with the nearest neighbors of a state, within a certain radius. This function is used both when Extending the tree and when Rewiring it.

Finally the planner RRTStarMotionPrimitives, its most important method is solve, it starts by adding the start state to the tree, and then contains a while loop, whose ending condition is the maxIterations to perform (this value can be set by the user). The while loop starts by sampling a state with the previous sample function, then it tries to find the nearest node in the tree by using the distance function, if there exists a motion primitive to connect the nodes then it finds which of the nodes around it has least cost and connects that motion by Extending the tree. To connect the tree it first checks if the motion is valid, meaning if it isCollisionFree, this function uses the getTrajectory function, to retrieve the points in the motion, and by using the isValidState function verifies that each state of the motion is obstacle free.

It then checks if the node is inside the goalRegion, if it is then a flag to check for a solution is set. Before entering the check solution part we get the near neighbors of the new random node and try to rewire the tree by comparing the combined cost of the existing nodes to the new random node, if there is a better cost then the tree is rewired. Then the function checks if a solution exists, here we compare the cost of the previous goalMotion, to see if it is better than the previous one or not. Once all the iterations have been done, if a solution was found it is returned.

Modifications

The first change was the way to access the primitives. Basak had used the matio library to access MATLAB matrices that contained the primitives information. Instead we added a function to Read the primitives information from a text file and store them in a vector. Each element of the vector contains the following information:

id: in form of a string, composed by $\theta_i, x_f, y_f, \theta_f$.

starttheta_c : discretized representation of the angle.

endcell: the end cell corresponds to the discretized value of final (x_f, y_f, θ_f).

intermptV: vector with the intermediate positions of the primitive, the values are not in discrete mode, but continuous.

cost: the cost value is calculated after.

Regarding the cost computation there are two possibilities, time and length:

1. length: If length is chosen, the linear distance is computed by taking the intermediate points and computing the sum of $\sqrt{dx^2 + dy^2}$ for all the points. For the simulations we have used the length cost, as the aim is to find the shortest path.

2. time: If time is selected, then we compute both the linear and angular distance, and by dividing by the nominal linear or angular velocity the time is obtained. The maximum between both times is selected as the cost.

Furthermore we have modified the distance and getTrajectory functions. Regarding distance it checks if the states are in a distance smaller of 2m, because our primitives, as shown in Section 4.3, have a maximum distance of 2m. Then it must be checked if there exists a motion primitive to connect the states, to do so we look in the vector that contains the primitives by means of the id, if there exists, the distance function returns the cost of the primitive, otherwise a cost of 10000 is returned, as shown in Figure 4.13.

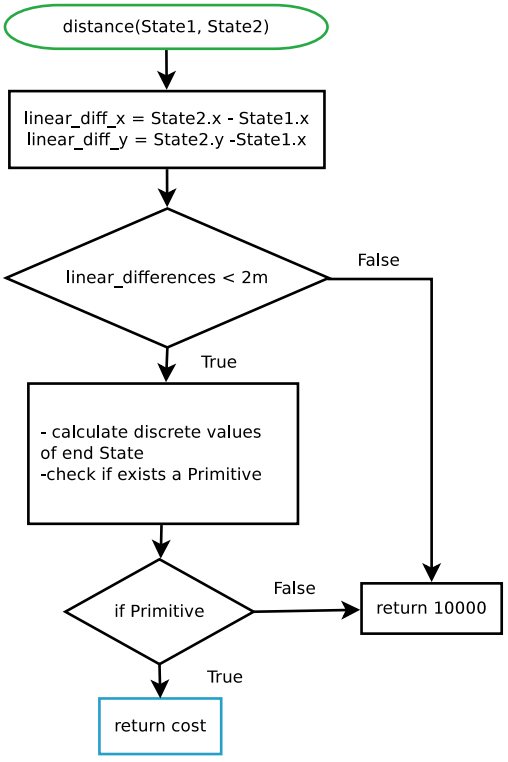


Figure 4.13: Distance Flow Diagram

The getTrajectory function also receives two states, and as the distance function it checks if a motion primitive exists to connect them, but instead of returning the cost it returns a

vector with the intermediate points of the primitive.

With these changes we were already able to use Basak's implementation, but we realized that the tree expansion was too slow. There are two reasons why the tree expansion is slow in our implementatio , the first is related to the size of the environment, unlike the original implementation, ours is a space of around 200x200 meters, and 16 possible angles. The wasted time is taken by the sampling function, the sampler takes a sample from the defined SampleSet, checks if is valid, then gets the nearest node in the tree and checks that the sample is at most 2m away from the nearest node (maximum size of our primitives), and then verifies if a primitive exists, if there exists, then the sample is accepted and the motion is connected.Therefore the size of the space makes it more probable to get a far away sample, which leads to a lot of samples are discarded. The second reason corresponds to the limited amount of motion primitives we have, Basak had a bigger amount of primitives which increased the probability to find a valid motion to connect the nodes.

Figure 4.14 shows our map, and all yellow points are samples taken by the sampler, in this case 4500 iterations were taken, it took more than 200s and not even one primitive was found to connect a new node. The original algorithm from [4] was unfortunately not able to work on realistic examples other than the small one provided by the author.

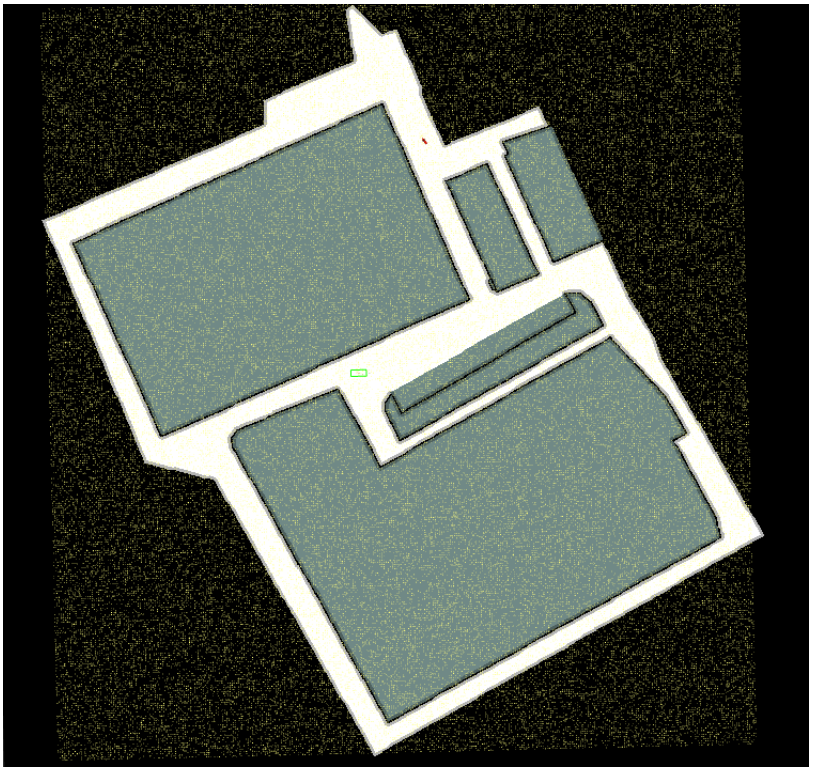


Figure 4.14: Sampling in all the space

This being said we came up with three novel ideas for changing the planner based on the sampling, by taking a node from the list of nodes in the tree, and then resample around it.

1. **The cube**: Given the fact that our primitives have a maximum size of 2m, we have decided to take the sample in a smaller area, that we have called cube, due to the fact of having three variables (x, y, θ).

   As in the original sampling mode in this section, we only check if the state is valid, the part to check if there is a valid primitive is performed later, but we have the advantage that it is less probable to discard a state given the fact that we are in an enclosed area. The idea behind the cube is the following:

   - Take a random node in the tree, and compute the boundaries of the cube as shown in Equation 1, having x and y as the position of the random node in the map and max_distance as the maximum distance that a primitive can reach.

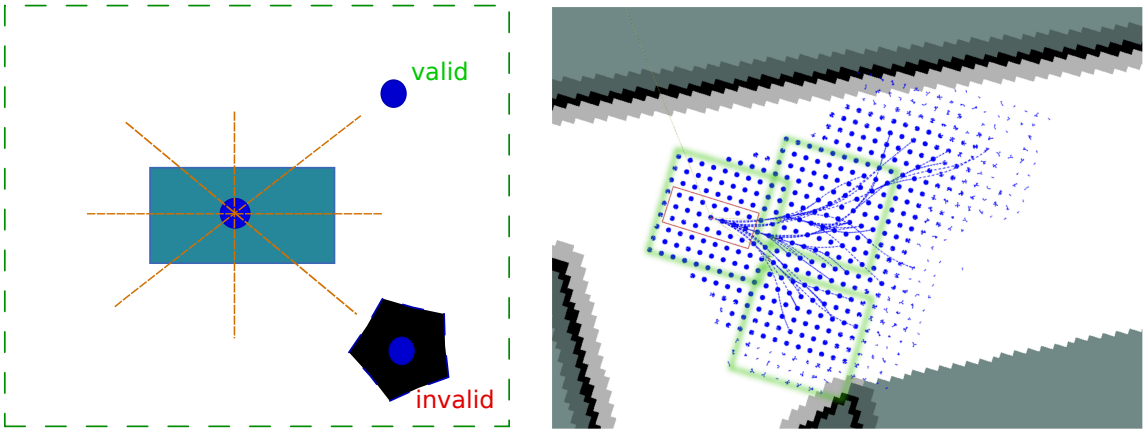
$$\begin{aligned} x_{min} &= x - max\_distance \\ x_{max} &= x + max\_distance \\ y_{min} &= y - max\_distance \\ y_{max} &= y + max\_distance \end{aligned} \tag{4.1}$$

   - Select a new random sample as seen in Equation 1, regarding the angle, we will take a value from the Theta Sample Set, this set contains the discretized angles, for 16 or 32 angles (depending on the used primitives), for x and y we also take only values respect to the resolution of the primitives.

$$\begin{aligned} rstate.x &= x_{min} + rand(0, \frac{x_{max} - x_{min}}{resolution}) * resolution \\ rstate.y &= y_{min} + rand(0, \frac{y_{max} - y_{min}}{resolution}) * resolution \\ rstate.theta &= ThSampSet[rand(0, num\_thetas - 1)] \end{aligned} \tag{4.2}$$

   - Check if the state isValid, otherwise sample again.

   Figure 4.15a exemplifies the mentioned behavior, while in Figure 4.15b, we have highlighted some of the samples within different cubes around the tree.



(a) Sampling inside a cube (b) Example of sampling with the cube

Figure 4.15: Cube Sampling Mode

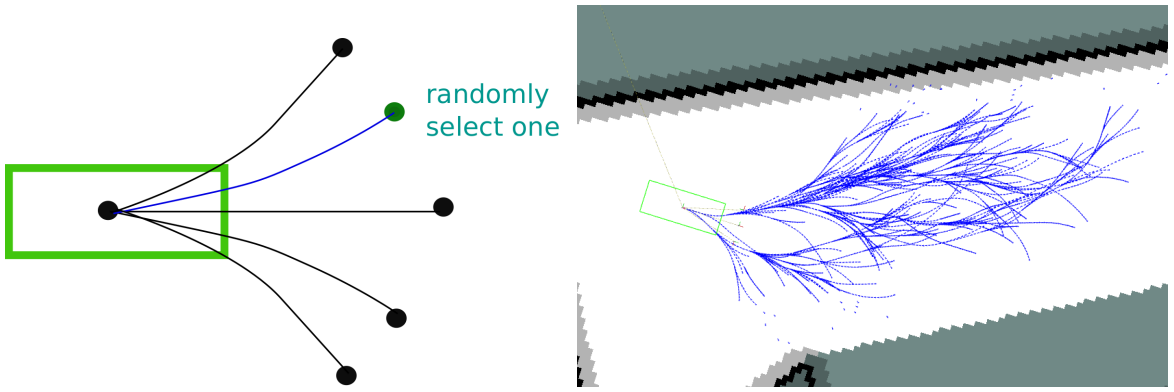
With this approach, we have a faster behavior than the original one; even though we were using a lot of iterations the tree makes connections faster. We observed two things, we were not removing the already used samples, and still in the cube there are many samples for which a primitive does not exist.

2. **Random Primitives**: Our second method aimed at solving the issue of wasted samples, i.e., those without primitive inside the cube. We generated a function to order the primitives according to the starting angle, so as to have separated all the primitives. Then, with a function getRandomPrimitive, instead of getting a Random state in the space, a random primitive is selected from the possible primitives for that particular node in the tree. This way we skip the samples discarded for not having a primitive able to connect them to the tree.

   The function returns the sampled state, which we then check for validity (not in obstacle), otherwise again we get a random node from the tree, and get a Random Primitive for this node.

   Figure 4.16a exemplifies the idea behind this method, we randomly select one of the available primitives and use the last intermediate point of the primitive as the random state to return. While on Figure 4.16b an example of this method being used is shown; unlike the previous method, we can see that every sampled state has a primitive, and only those that are obstacle free are connected to the tree.

   With this second method we still have the issue that we could be sampling again the same primitive for a certain node.



(a) Randomly selection of the primitives *(b) Application example*

Figure 4.16: Random Primitive Sampling Mode

3. **Random Primitive and Store**: This last method is very similar to the previous one in the fact that we only want to sample an available primitive, but now, when adding a new node in the tree we store its possible primitives, and every time one is used, or it is not valid we remove it from the storage, so that it is no longer accessed. If all the primitives of a node have been checked, then we set that node as Not Expandable.

   In Figure 4.17, it is shown how the first node of the tree (start position) is stored with its available primitives, once a primitive has been used it will be removed from the storage, and the new node will store its possible primitives. In red we show a primitive that had a not valid state, this primitive is also removed from the storage.

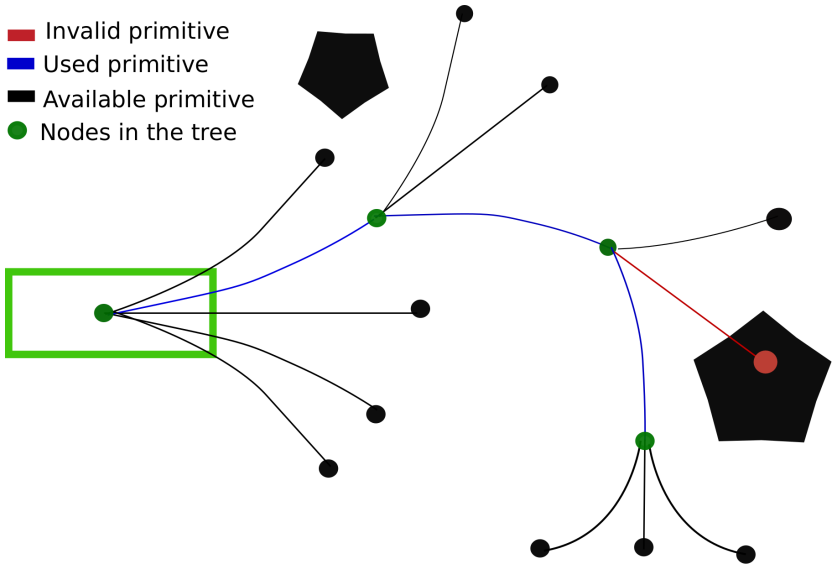


Figure 4.17: Sampling in the Random Primitive and Store Method

Once we finished with the three new methods for sampling, we realized that we still had some issues expanding the tree towards the goal, the reason for this is that in the usual OMPL implementations, when a sample is taken from a distance longer than a defined maxDistance, an interpolation function is used in order to find a new node in that direction, but in accordance to the maxDistance parameter. In our case, since we need to have a motion primitive to connect the nodes, this approach is not used. Instead what we implemented is a way to sample with respect to the distance to the goal as follows:

- When finding a new node save also the Euclidean distance to the goal in a vector, and in every iteration sort the vector, in order to have the ones closer to the goal in the upper positions.
- Start the tree expansion by taking random nodes from the whole tree.
- Once we have a certain amount of nodes in the tree (N), using a mixing parameter (or threshold) we select between choosing a node to sample from the whole tree, or take one that is closer to the goal, by randomly selecting from the N first nodes in the distance vector.
- If a Solution has been found, in order to improve the path, we sample from the whole tree, and not from the ones closer to the goal, to keep refining the solution. This is shown in the diagram of Figure 4.18.

The example shown in Figure 4.19 shows how the tree expansion behaves in both scenarios. If the distance to the goal is not taken into account, most of the nodes concentrate in a smaller area. Instead, when we take into consideration the mixing parameter to iterate sampling from the whole tree and the nodes closer to the goal, the expansion towards the goal is easier.

The last element modified in order to speed up the search was the function to get the nearest neighbors. The original implementation checked the whole tree in order to find those that had a primitive to connect to the node. Instead we checked in a gradual form first the current parent of the node, and then progressively checking the children of that parent and the parent of the parent, and its children. By doing this, we reduce the amount of verifications we must do in order to get the neighbors. Comparisons of the two methods are shown in the next chapter.

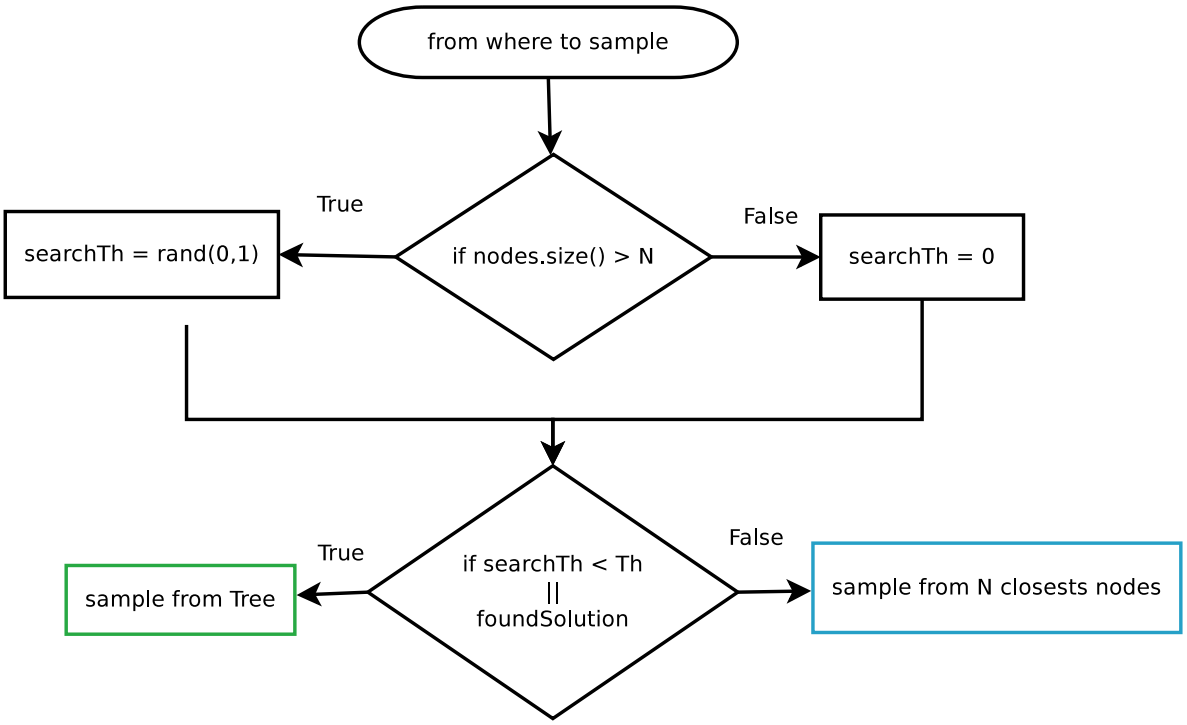
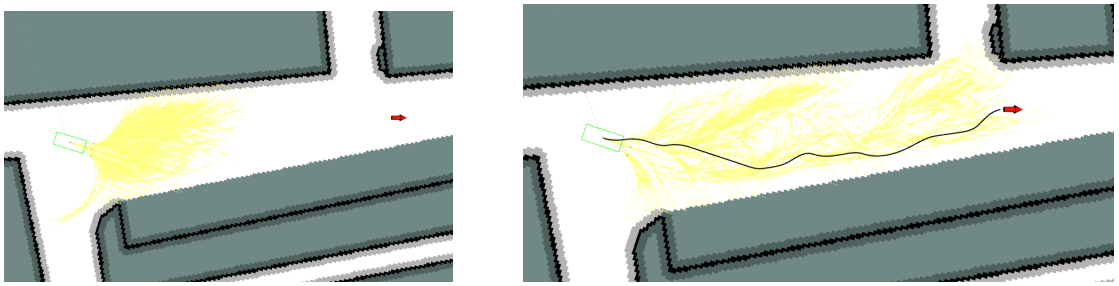


Figure 4.18: Sample mixing parameter



(a) Without sampling towards the goal *(b) Sampling with mixing Parameter*

Figure 4.19: Comparison between sampling with and without mixing parameter

Chapter 5

Simulations and Results

This chapter contains the tests we have done to evaluate the performance of the proposed solutions, we first explain the environment we used, as well as the Simulator, then we continue with different simulations between the planners.

The idea of the simulations is to compare three global planners implementations. The first is the move_base global planner A*, following by the developed SBPL Global Planner and the OMPL Global Planner. We first compare our global planners with A*, to show their different properties and advantages. And then we compare the SBPL planner and the OMPL planner (the standard planners, and the RRT*_MotionPrimitives planner) both in terms of planning time as in the optimality of the path. The simulations are enumerated as follows:

1. Move Base A* vs SBPL AD* and ARA* without motion primitives.
    - SBPL with MP, replanning due to costs changes.
    - SBPL with MP, manual refinement.
2. Move Base A* vs OMPL
    - For OMPL: RRT , PRM, RRT* and PRM*..
    - For OMPL: RRT* Dubins
3. SBPL AD* with 16 and 32 MP and OMPL RRT* with Dubins and Reeds-Shepp.
4. SBPL AD* MP vs OMPL+MP, with the different sampling modes.

5.1 Environment

For our simulations our main environment is a map of $203 \times 217\, m^2$, with a resolution of 0.25m, meaning that the grid is conformed by 812×868 cells, to which we refer as Aster map. We have selected 3 start positions for the vehicle, and for each start several goal positions. As shown in Figure 5.1 we have used different colors for the (start,goal) pairs for clarity, the goals have been named by the number of the start position and a letter from a-d.

Figure 5.1: Aster environment, with different (start,goal) pairs

We also made use of a different map, to which we will refer as U5, just to show the portability to different environments. The map has a resolution of 0.1m for an amount of cells of 1059x999. This map is from a parking lot in University of Bicocca.

Since this map has a smaller resolution, then different motion primitives were used as SBPL requires the same resolution of the map as shown in Figure 5.3.

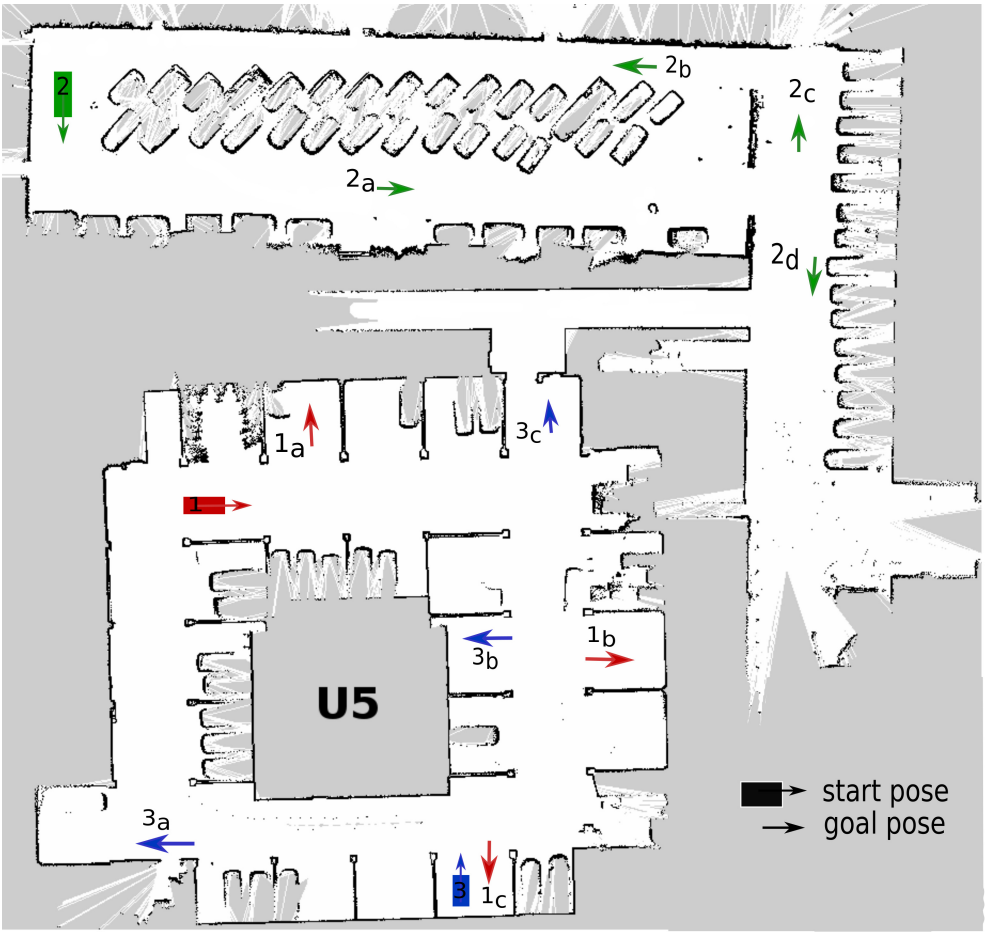
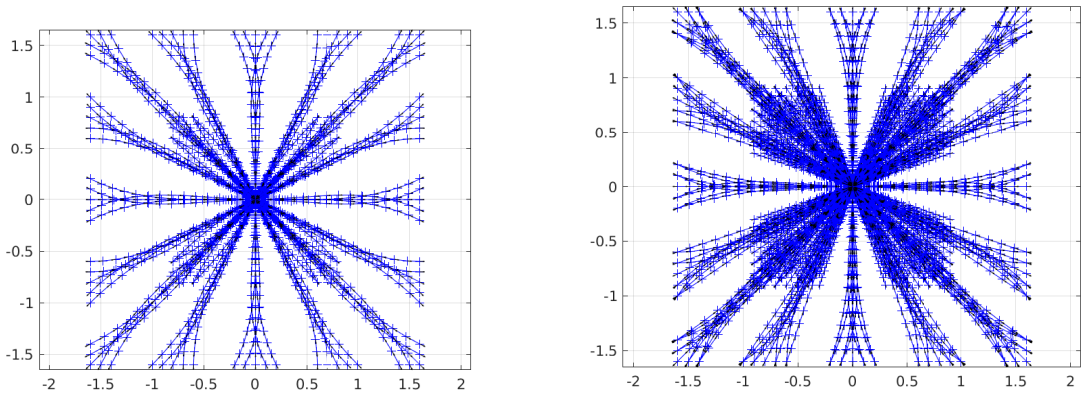


Figure 5.2: Park Lot environment, with different (start,goal) pairs



(a) Primitives with 16 Angles

(b) Primitives with 32 Angles

Figure 5.3: Motion primitives with 0.1m resolution

5.2 Simulator

For the simulations we used Stage in ROS, which is a robot simulator that provides a virtual world populated with mobile robots and their sensors [36].

For its usage first the world must be defined, it contains:

- window: the size, center, rotation and scale of the space must be defined.
- floorplan: For the floorplan a path to the bitmap to be used is specified, as well as its size, and position in the window.
- robot: the model of the robot to be used.

In Stage the mobile robot base models are called position model, our simulated robot was created as a position model and named carlike_robot model. The carlike_robot contains three laser models situated in front of the vehicle, on the left and on the right corners. For localization we used a *gps*, since the simulator returns the true global position. To define the kinematics, Stage has three option: *diff*, *omni*, and *car*, the last one was selected as it accounts for the Ackerman vehicle kinematics, it has velocity and steering angle, plus the wheelbase value is set.

Figure 5.4 shows the view of the simulator using the Aster map. The red box represents the robot mobile base, while the green parts represent the data from the laser sensors.

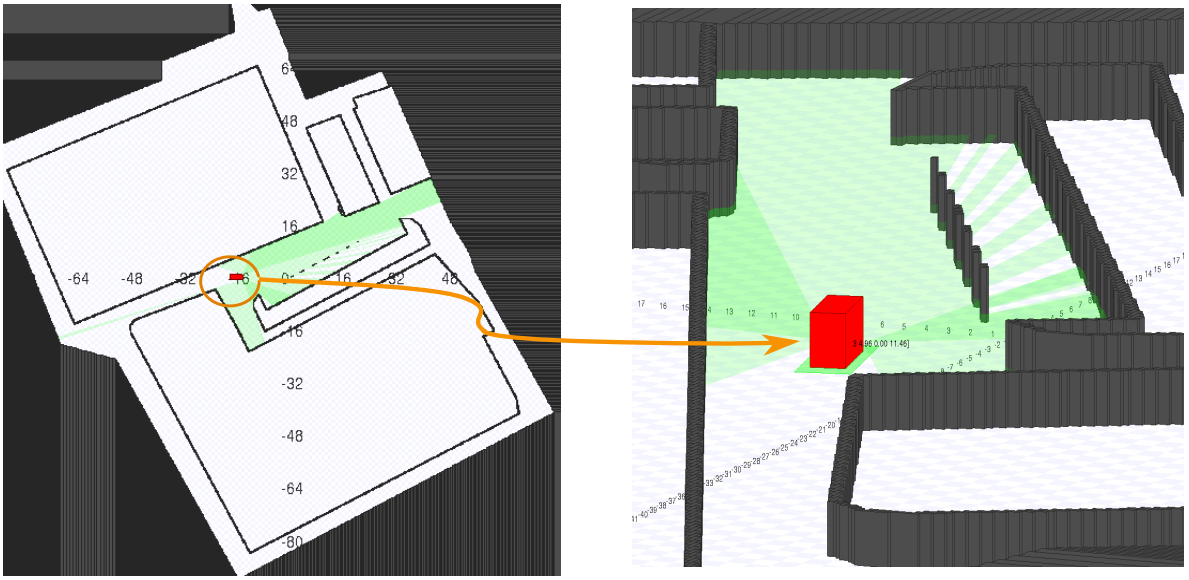


Figure 5.4: Stage Simulator

5.3 Tests Setup

For setting up the different tests we have written two nodes, test_planners_pub and test_planner_sub, the first one publishes the start and goal pose messages, while the second node subscribes to the results data, to be stored and analyzed later on. The flow diagrams of these two nodes are shown in Figure 5.5. For the publisher a timer has been defined and two publishers, one for the start and the other for the goal. The start positions and the goals, shown in Figure 5.1 and Figure 5.2, are taken from some predefined lists. The timer activates every certain amount of seconds (depending on the simulation parameters), and when it does, it activates a flag *publish*. Every time the flag is set, two PoseStamped messages are filled with the goal and start states from the lists. When the end of the lists is reached the node stops.

The most important results for us are the planning time and the path length, the planner nodes are publishing these values inside a std_msgs::Float64MultiArray message in a topic called /results.

The test_planner_sub node subscribes to the goal, the path, and to the results topics. Every time a query is sent to the planner it will then publish the path, and the planning results. When this happens our subscriber node takes the information and stores in different vectors, when the results topic is received a counter is increased, if the counter reaches 10 (number of queries for the Aster and the U5 maps), then the results, the goals, and the paths that were stored in the vectors are written to a rosbag file.

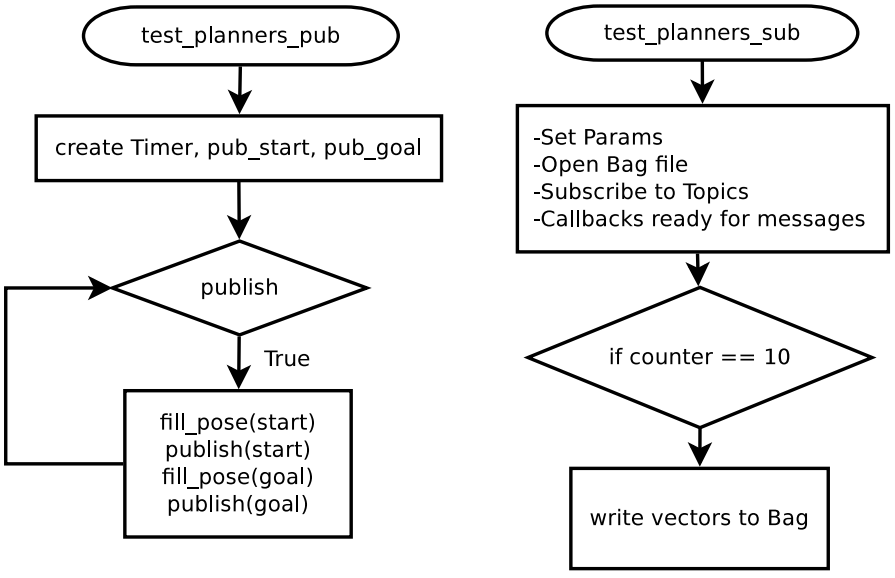


Figure 5.5: Test nodes flow diagram

To plot the information we have used MATLAB, since it has the possibility to easily open

and parse rosbag files.

5.4 Simulation # 1

Our first simulation is a comparison between SBPL planners without motion primitives and ROS move_base planner. For this test we have set an initial $\epsilon = 3$, and a decrease epsilon step of 0.2, while the allocated planning time was set to 10s.

The reason why we decided to compare planners without motion primitives was to have a fair comparison with move_base which does uses primitives. Given that with move_base there is no access to planning time, this was computed as the time difference between the moment our test_planner_sub receives a goal and the moment it receives the resulting path. The length was calculated as seen in Equation 5.1 for all the planners, i.e., being n the number of points in the path

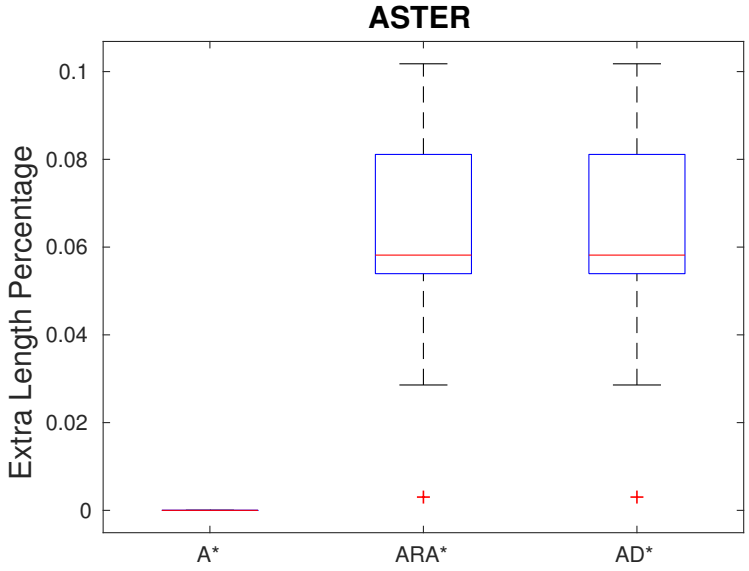
$$length = \sum_{i=0}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{5.1}$$

To show the results we decided to use box_plots; since they show the data highlighting the 25, 50 and 75 percentile values, as well as the maximum and minimum value. In the case of time, box plots show the required planning times in seconds. For length we decided not to show the length for each path, but to take the minimum path of each query and use that value as reference; box plot show how much the planners differ from the minimum path in percentage (the deviation from the optimal path), as shown in Equation 5.2.

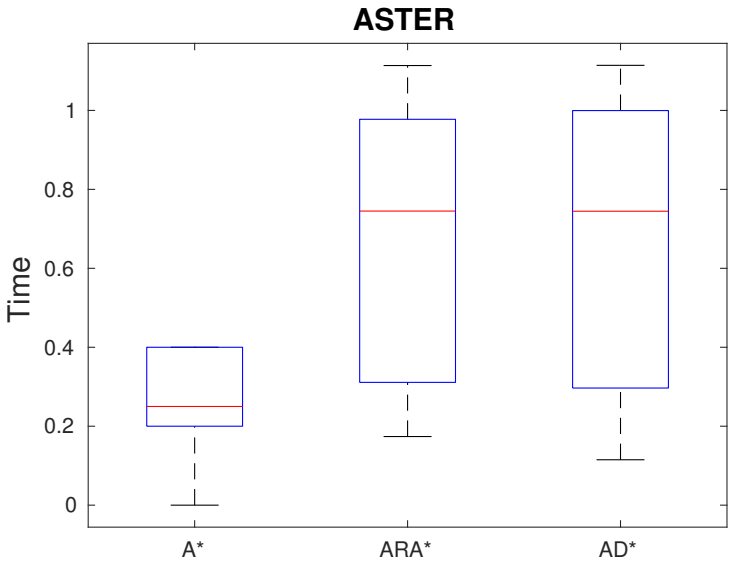
$$modifiedLength = (currentLength - bestLength)/bestLength \tag{5.2}$$

For this simulation, the results in Figure 5.6 show that in general A* finds smaller paths and the time response is smaller. AD* and ARA* have an identical response since, unless there are some dynamical changes in the environment, both algorithms have the same behavior. Because of this reason we continue only with AD* comparisons, as in general it is the most complete of the SBPL planners.
It is important to mention that all the planners in this simulation were able to find a solution, with $\epsilon = 1$.

Given that the length and time plots do not show any attractiveness for using AD* other than A* on Figure 5.7 we show two of the 10 planning queries, and highlight two interesting features of AD*.



(a) Extra Length



(b) Time

Figure 5.6: A vs AD* and ARA**

The first feature is circled in red, it shows that AD*, or SBPL in general, cares about the path planning in (x, y, θ), therefore it considers the orientation of the goal to be part of the solution, reason why the path length is not the same as A*. While the part circled in blue shows the fact that for AD* it is possible to select if the solution should be searched backward

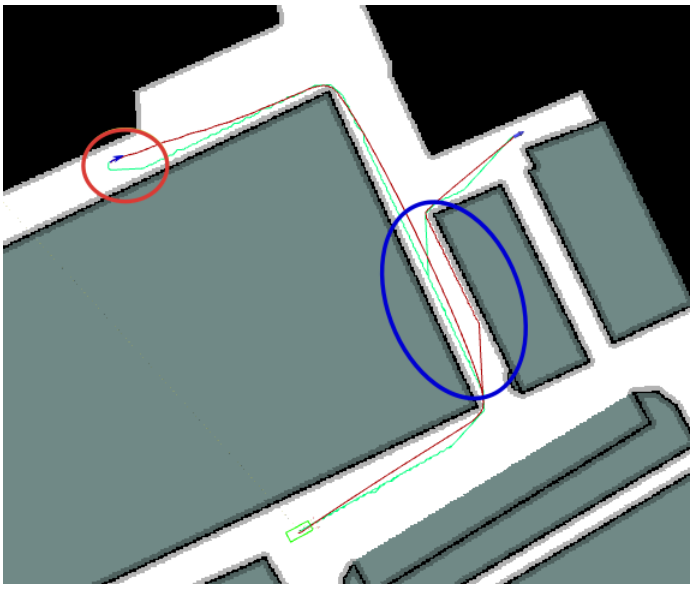


Figure 5.7: AD vs A**

or forward from the goal, unlike A*; therefore the path in that area is thus reversed.

Besides the possibility for backward search, another useful tool, is the possibility to plan in dynamic environments, unlike ROS move_base global planners, the SBPL planners we have developed, when having information from sensors to update the costs of the grid, can replan and modify the solution as to avoid obstacles. This is shown in the example of Figure 5.8.

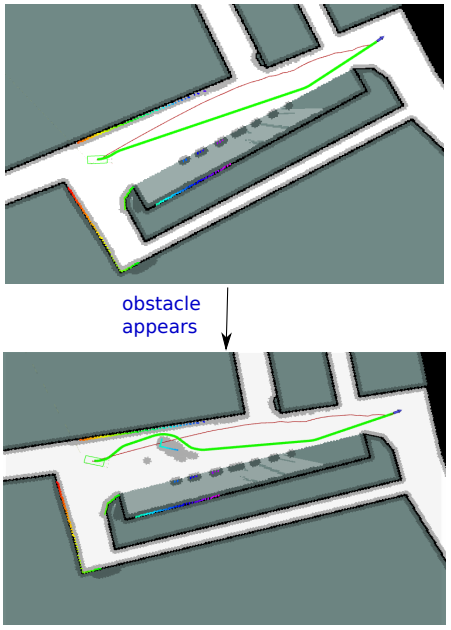


Figure 5.8: Dynamic Planning Example

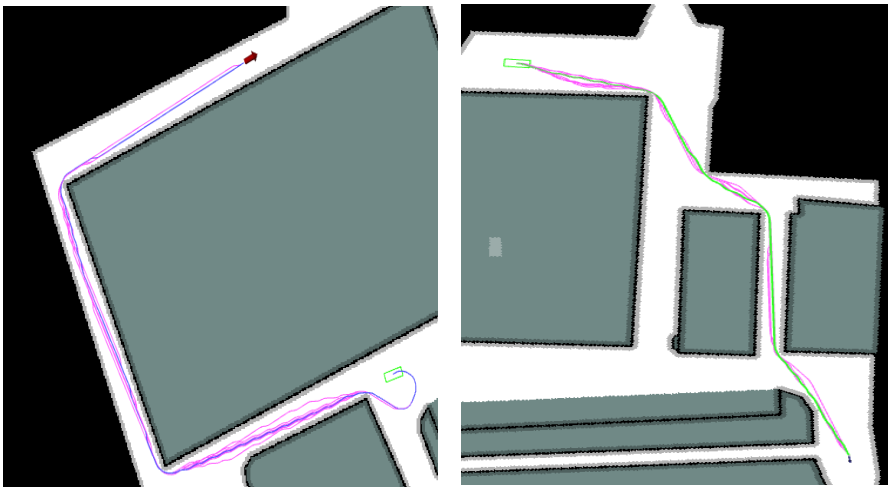


Figure 5.9: Manual Refinement Examples

Another advantage of using anytime algorithms is that they can quickly find an initial, possibly suboptimal, solution, and then concentrate on improving the solution while time allows. To have the best performance however an adequate epsilon should be selected, as well as the decrease step. In this case we can make use of the manual refinement option we have set in our implementation, and check the behavior of the expansions at each ϵ decrease.

We started our test with a $\epsilon = 5$, and decrease step of 0.1, and realized that usually most of the expansions are done after the replanning function arrives to $\epsilon = 3$. We also realized that there is a tradeoff between the selection of this two parameters and the planning time. If we care to obtain a very fast initial solution, then a big epsilon is required, if we want to obtain rapid solutions between the refinements then the epsilon decrease step must be small. On the other hand, if we care to find only the optimal solution, then we can set the initial epsilon equal to one, and do not perform any refinement, which means there will not be a fast probably suboptimal initial solution, we would obtain only the optimal solution. We have decided to select an $\epsilon = 3$, and step of 0.2 for all the cases, in this way we can have fast refinements between solutions, and a fast first path. Two examples of the manual refinement are shown in the Figure 5.9, the green and blue paths corresponds to the normal path planning, while the pink lines show the paths obtained by manual refinement.

For a better understanding of how the process of refinement works, Table 5.1 shows the data recovered from the refinement of the left image in Figure 5.9, the data contains the epsilon, the cost, the extra time in each refinement, and the number of cells expanded. This query was solved in 16.987s, with a final cost of 161460, which corresponds to the 161.5m of the path. As it can be seen all the expansions until $\epsilon = 3$, had the same cost, after this point more expansions are performed on every epsilon decrease until the optimal solution is found.

Table 5.1: Refinement information Example 1

Solution	eps	cost	δtime	expands
0	5	165532	0.271438	20196
1	4.9	165532	0.146475	9684
2	4.8	165532	0.120149	8800
3	4.7	165532	0.098965	7727
4	4.6	165532	0.080198	6676
5	4.5	165532	0.046682	3615
6	4.4	165532	0.023954	1515
7	4.3	165532	0.007278	564
8	4.2	165532	0.009417	351
9	4.1	165532	0.003308	228
10	4	165532	0.00382	327
11	3.9	165532	0.011265	800
12	3.8	165532	0.139828	9843
13	3.7	165532	0.124243	8887
14	3.6	165532	0.236199	10527
15	3.5	165532	0.277376	10406
16	3.4	165532	0.226016	8955
17	3.3	165532	0.347754	13603
18	3.2	165532	0.369411	13064
19	3.1	165532	0.365777	13678
20	3	165532	0.399822	15141
21	2.9	163670	0.375819	17916
22	2.8	163670	0.235848	11668
23	2.7	163670	0.3156	14503
24	2.6	163670	0.407761	17020
25	2.5	163670	0.789697	33129
26	2.4	163670	1.07677	43438
27	2.3	163055	1.34661	54709
28	2.2	163055	1.97196	77476
29	2.1	162799	2.07898	83391
30	2	161731	1.99298	81530
31	1.9	161460	1.75625	67540
32	1.8	161460	0.085348	3700
33	1.7	161460	0.145198	4642
34	1.6	161460	0.114813	3838
35	1.5	161460	0.072926	2926
36	1.4	161460	0.072076	2514
37	1.3	161460	0.063012	2473
38	1.2	161460	0.076278	2548
39	1.1	161460	0.081132	2660
40	1	161460	0.053437	2765
41	1	161460	0.013739	0

5.5 Simulation # 2

The second test performed is a comparison between ROS global planner A* and the OMPL global planners RRT and PRM and their asymptotically optimal versions RRT* and PRM*. As with the first test, we used the Aster map with the same 10 queries. The allowed time set for the OMPL planners was 1s, with the possibility to make 3 extra trials.

All the planners found a solution inside the goal region. The obtained results are shown in Figure 5.10, as it can be seen regarding the time, the RRT and PRM planners found the solutions in the shortest time, while the RRT* and PRM* took all the given time to improve the solution (in this case the imposed time was 1s). Regarding the length, in all the occasions A* presented the smallest path, while RRT had the biggest deviations from the best paths in all the queries.

The most important aspects regarding the solutions found is related to the feasibility of the paths. In the examples shown in the Figure 5.11 it can be seen that even though the fastest planners are the RRT and the PRM the found solution is not optimal, nor feasible by the vehicle. A* presents shorter paths than the RRT* and PRM* planners.

Another aspect to note regarding the solutions with OMPL is that the path is always further away from the walls in the map, the reason for this is that the samples in the circumscribed area to the obstacles are not considered valid, therefore the path is more distanced form the walls unlike A*. This can be an advantage for the local planner in the sense that it gets more space for maneuvering the vehicle without the possibility to collide.

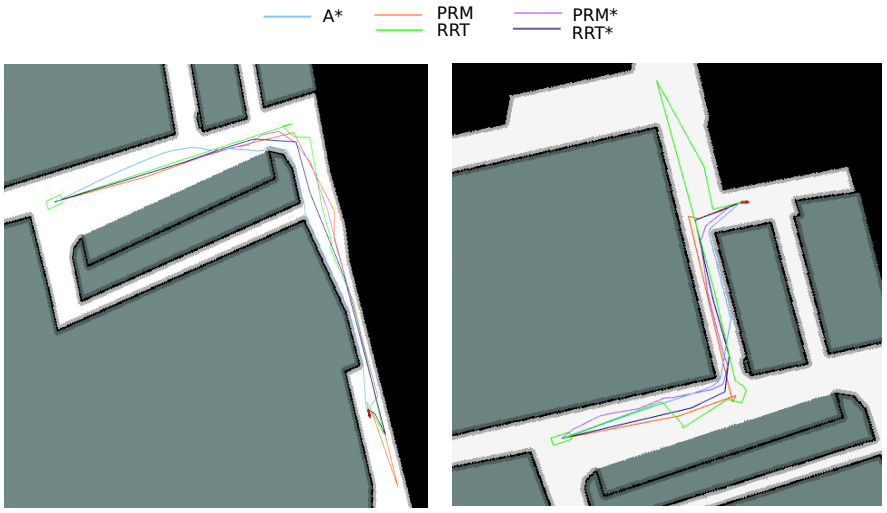
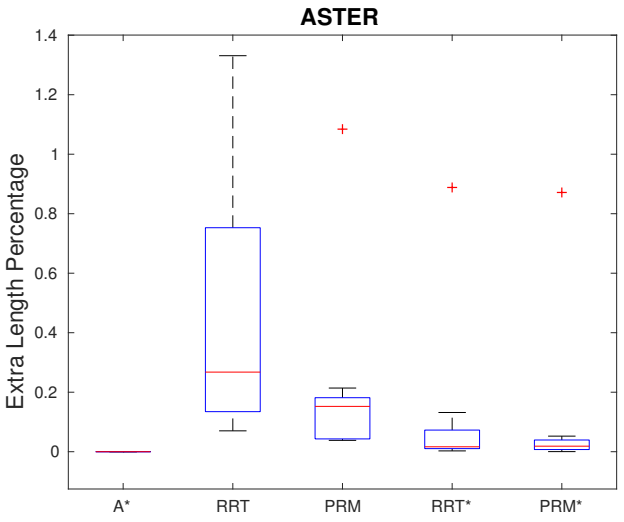
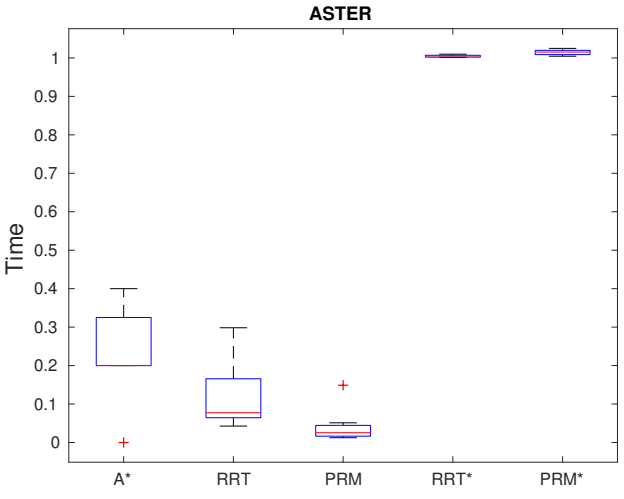


Figure 5.11: Planning examples of A vs OMPL planners*



(a) Extra Length

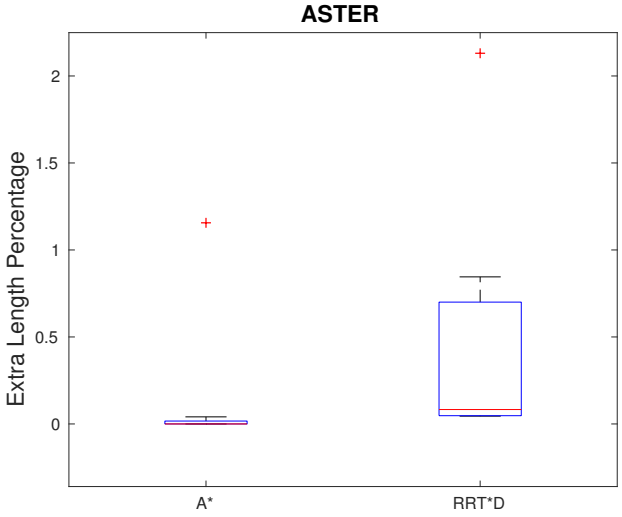


(b) Time

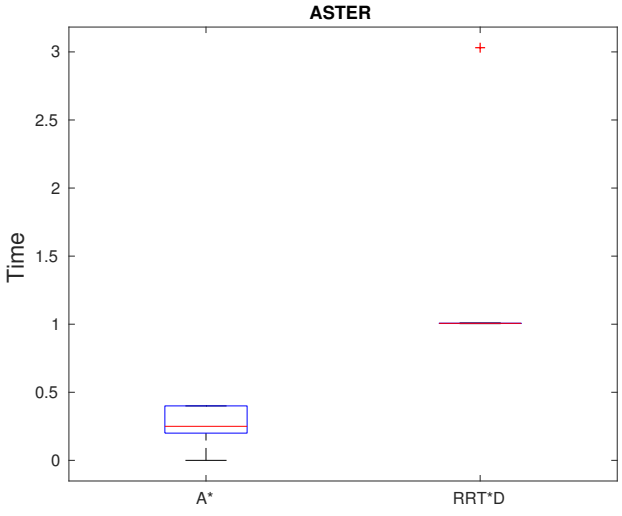
Figure 5.10: A vs OMPL planners*

Given that the solutions did not give a smooth path for our vehicle, we tried with the Dubins Space, which, given a turning radius parameter (3.0705m in our case) provides three motion primitives, turn left, turn right and go straight. To form the path OMPL with Dubins Space uses a combination of these primitives, instead of just simple interpolation to join the points in the tree (which is the reason why the others are not smooth). The results are found in Figure 5.12. As a fact the solutions found by the RRT* with Dubins are not faster, nor they have smaller path. It can also be noted that one of the solutions was not exact, it used 3

seconds to try to find the path, and result in a not exact solution giving a smaller path length than A*, even though is just an approximation to the goal (reason why the A* path length deviation shows the red cross).



(a) Extra Length



(b) Time

Figure 5.12: A vs RRT* Dubins*

By looking at the query examples in Figure 5.13 the solution is sure more smooth than the ones shown in the previous example, also compared to A*, the Dubins path takes better care of the turns, allowing a vehicle to transverse it. As mentioned before OMPL also contains the Reeds-Shepp space, which we consider in the simulation of the next section.

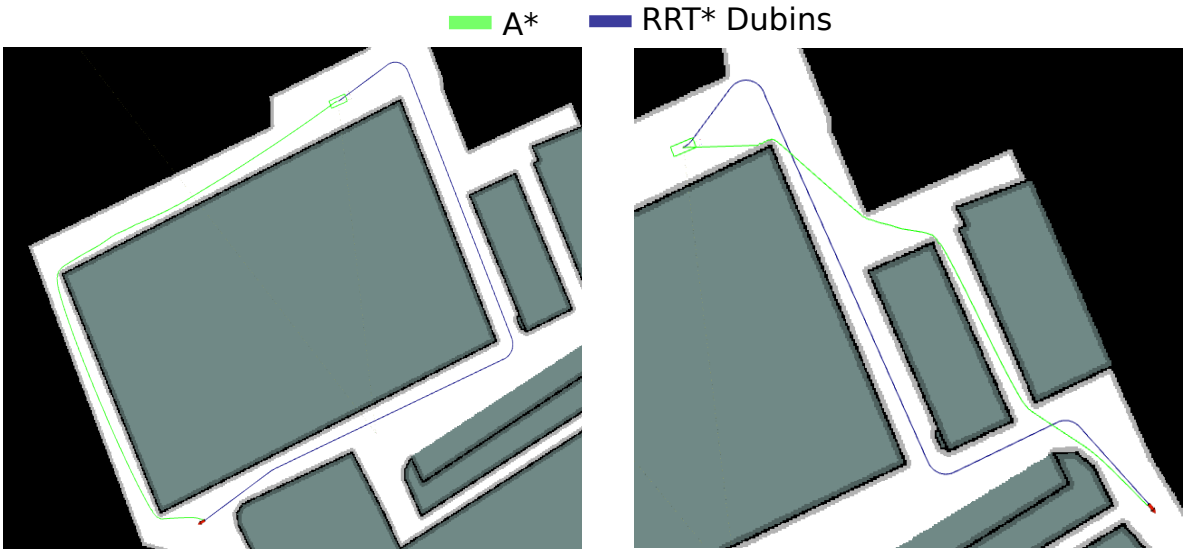


Figure 5.13: Planning examples of A vs RRT* Dubins*

5.6 Simulation# 3

The third simulation is a comparison of the AD* algorithm with motion primitives with 16 and 32 angles, and of the RRT* with Dubins and Reeds-Shepp spaces. The previously used parameters remained the same with some additions; in the case of the Search Based planers the motion primitive path was set, and for the Sampling Based Planners the flag for either Dubins or Reeds-Shepp. For this test we used both the Aster Map and the U5 map.

Starting first with the Aster map, some of the paths founds can be seen in the Figure 5.14.

Figure 5.14: Planning Examples Simulation # 3

In the case of the AD*, the found solutions returned an $\epsilon = 1$ meaning that they are optimal. The planner with the motion primitives of 16 angles found the solutions around 3 or 4 times faster than with 32, but with respect to length the later usually found a smaller path, with maximum differences of 1m as seen in Figure 5.15a.

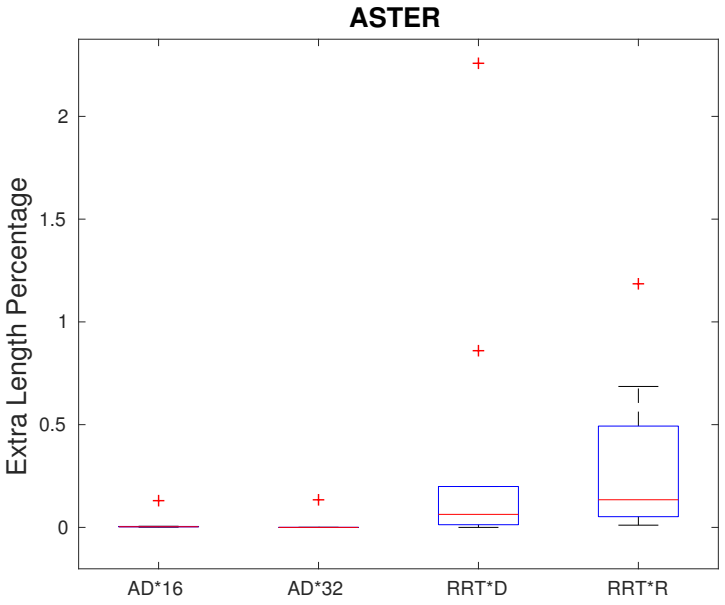
In sampling based planners all the solutions returned an exact solution, except for one case in the Dubins solution, which corresponds to the 1c query (see Figure 5.1). The solution found is an approximation, which is why the length of that solution is smaller than the one from the other three planners (red crosses of AD*16 and AD*32 in Figure 5.15a). As a clarification we refer to a solution to be exact if the path arrives to the goal region, this does not mean that the solution is optimal, as a matter of fact it can be seen in the box_plots that RRT*D and RRT*R differ up to 1% of the path length respect to AD*32. When a solution found is not exact in OMPL, we have set the possibility to plan again but with more time available, to try to solve the query, given that a solution could have not been found because of the limited time in the tree expansion. An amount of trials can also be set, which determines the extra time to plan each time.

In fact for that specific query RRT*Dubins used 3 attempts to find this approximate solution, which lead to the 3s point in the Figure 5.15b. As well AD* with primitives of 32 angles struggles more to find a solution taking around 13s to find it, the reason of this are the position and orientation of the goal, given that it requires the robot to make a complete change in original orientation to arrive to the goal. Figure 5.16 shows the resulting paths from these two planners.

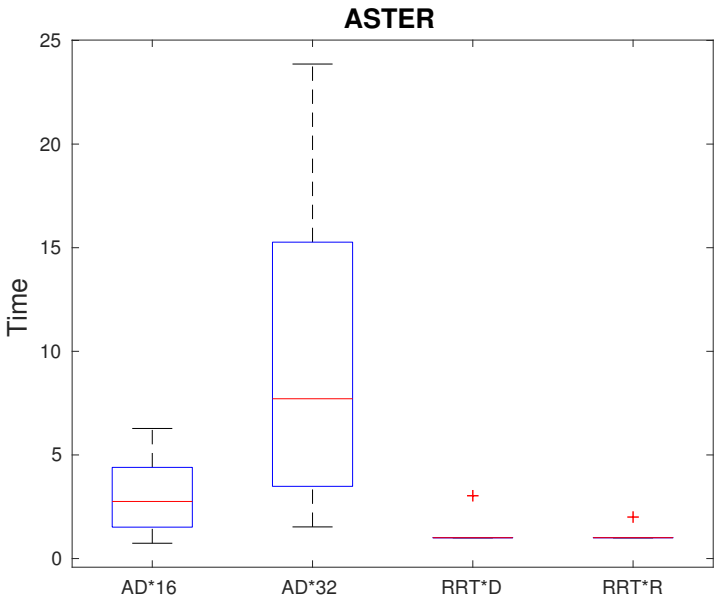
After testing with Aster map we continued with the U5 environment which corresponds to a parking lot as mentioned before, therefore there is not much space for maneuvering a vehicle. This can be a difficulty for Dubins and Reeds-Shepp since they have very few motion primitives.

An example of the solutions found in this case are shown in Figure 5.17; it can be seen that the RRT* solutions present sharper turns which might be complex, or even not feasible. This is a problem of dealing with so little amount of primitives, while for the AD* solutions the trajectory becomes smoother. On the other hand for RRT* with Reeds-Shepp, it can be seen that just before performing the curve it makes a small reverse motion.This should not be accepted general, as this will become an issue for the local planner, indeed there is the possibility that the robot does not have sensors in the back, which can cause collisions in dynamic environments.

As before the planners were benchmarked by time and length, in the case of AD*32 and AD*16, the first showed a slower time response, while the path length of both was very similar,



(a) Extra Length



(b) Time

*Figure 5.15: AD*16, AD*32 , RRT*D and RRT*R in Aster map*

even though the motions differ. In this occasion the results were not good for the Dubins and Reeds-Shepp paths. In fact for RRT*R and RRT*D the queries 1b, 1c, 2b, 2c, 2d , 3a and 3b

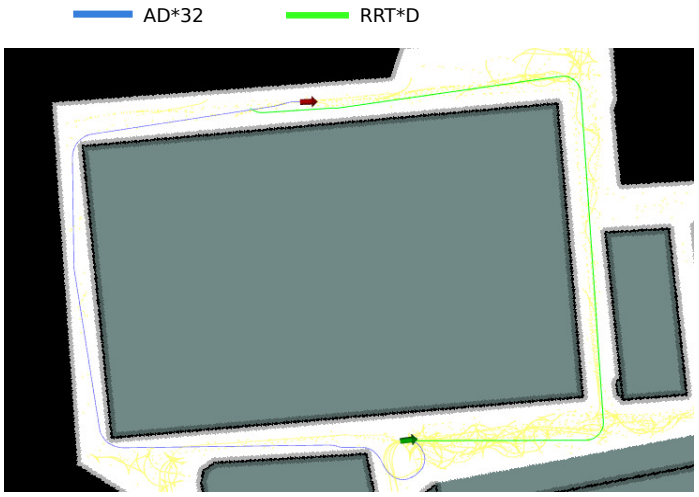


*Figure 5.16: Query 1c, AD*32 vs RRT*D*

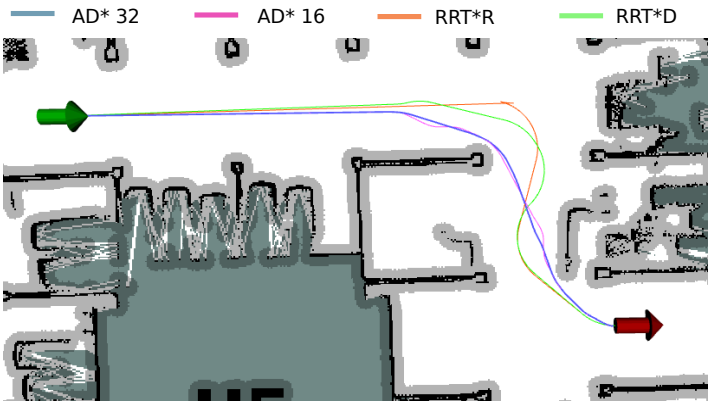
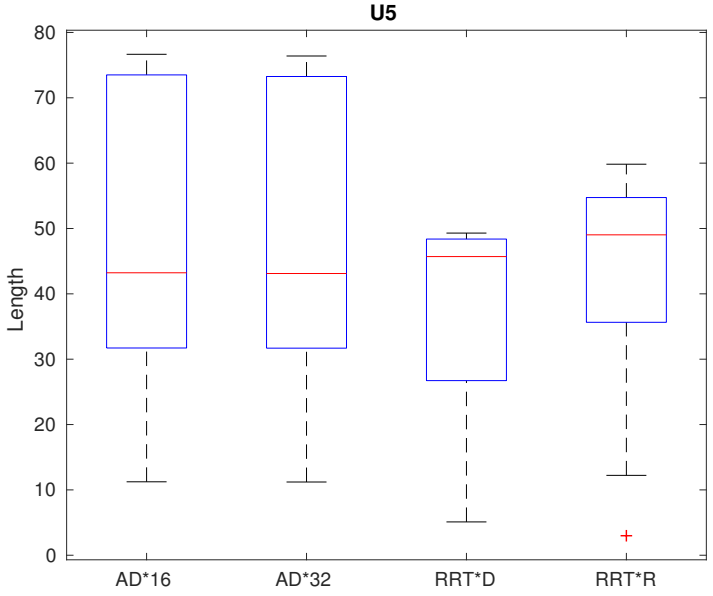


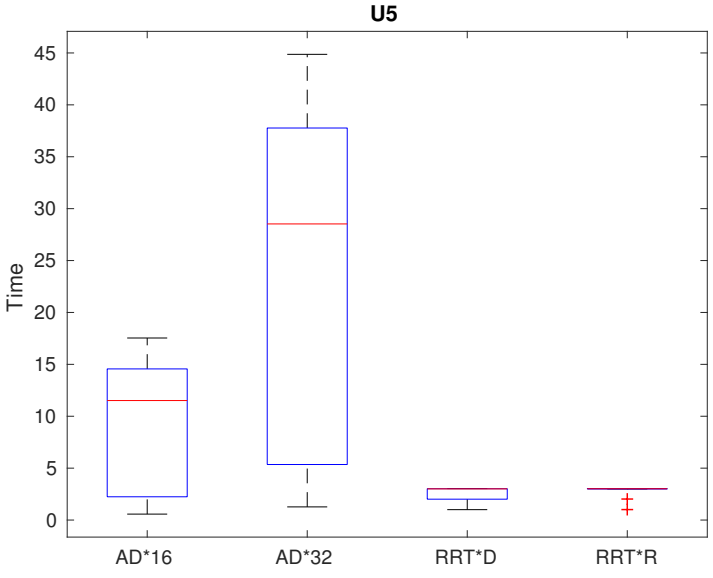
*Figure 5.17: Planning examples of AD*16, AD*32 , RRT*D and RRT*R in U5 map*

could not find an exact path, which is why the length shown in 5.18a shows as if the found paths with the sampling-based planners were shorter, for this comparison the length of the paths was plotted as a whole without the subtraction of the best path, as many of the queries showed shorter paths as approximations to the goal.

The reason of why the planners were not able to find an exact solution, is that the tree expansion with these two spaces is more constrained, as the sampling is done according to the predefined Dubins and Reeds-Shepp paths. A prove of this is shown in Figure 5.19. The first image shows the four planners, having the Search-Based planners correctly finding the optimal path, while the RRT* planners not. The second one shows with some red dots the sampling performed by the Dubins state space, which shows that in the given time, in that area it could not find states to add to the tree. Given this problem we tried with the RRT* regular implementation, to see if under the usual SE2 space the node expansion would be



(a) Length



(b) Time

*Figure 5.18: AD*16, AD*32 , RRT*D and RRT*R in U5 map*

handled properly. This is reported in the third image the correct path is found by RRT*.

In Figure 5.21 the plots of time and extra length percentage of the planners that found an exact solution are presented. The AD*32 paths had the smallest length, with very close

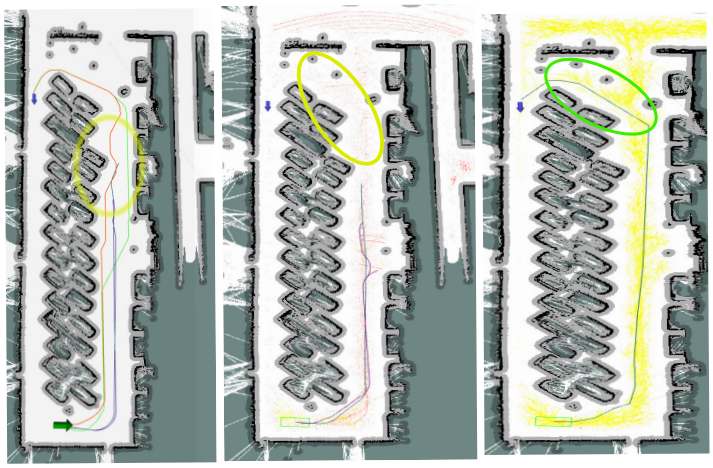


Figure 5.19: Dubins and Reeds-Shepp issues sampling

results by the AD*16, but the time required for planning was much higher as seen in Figure 5.21b.

Initially we only tested the RRT* with Dubins and Reeds-Shepp spaces, but given the obtained results we decided to try for the ten queries also with RRT* to compare the solutions with AD*. Figure 5.22a, shows the paths found by RRT* are very close to the ones found by AD* planners, which were optimal, except for one of the queries, i.e., the 3a. RRT*, instead of taking the shortest path as AD*, returned a path that traverses the whole perimeter around the U5 building. As seen in Figure 5.20, in the area where the AD* path is found, there are no yellow arrows (represent valid states), because of close obstacles, which is why the RRT* took a different route.

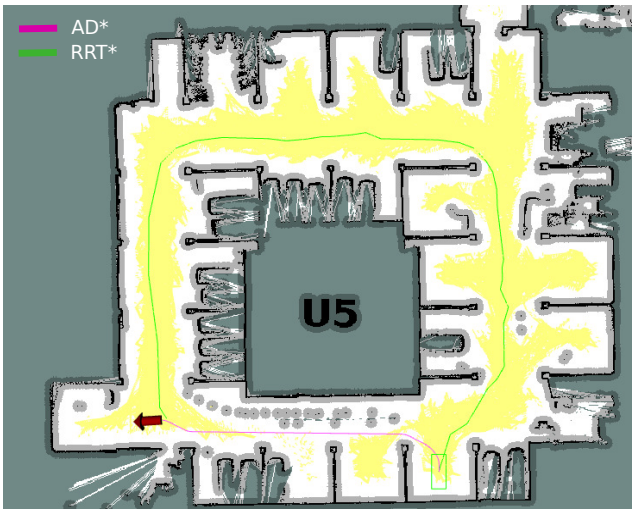
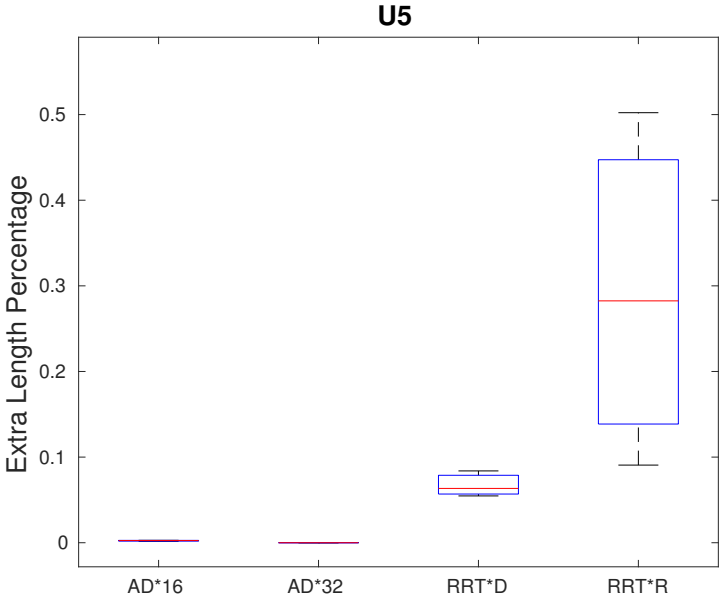
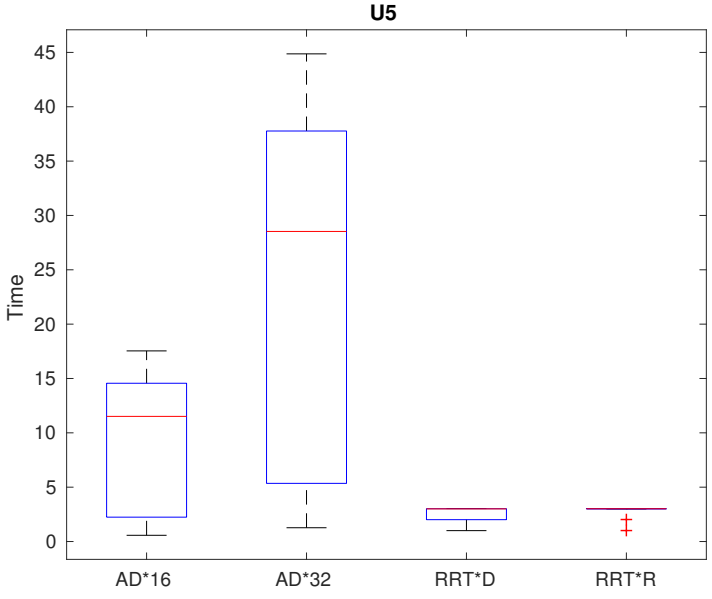


Figure 5.20: Query 3a in U5, RRT vs AD**

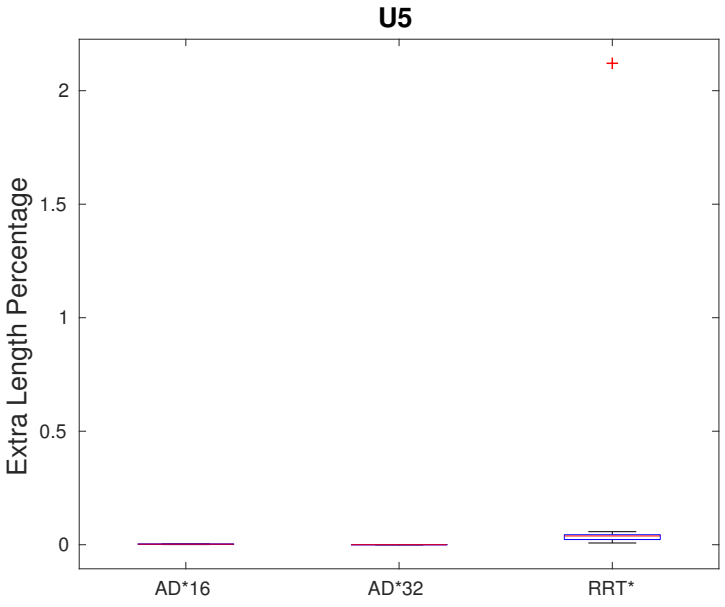


(a) Extra Length

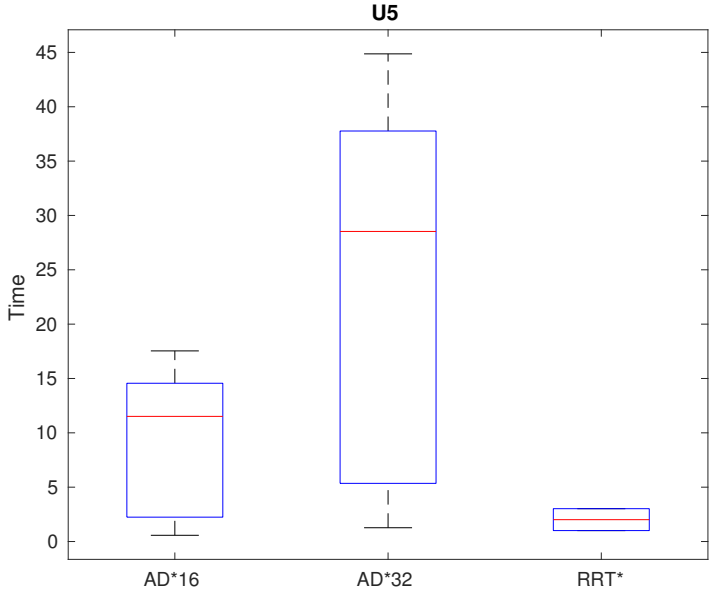


(b) Time

*Figure 5.21: Exact solutions of AD*16, AD*32 , RRT*D and RRT*R in U5 map*



(a) Extra Length



(b) Time

Figure 5.22: AD*16, AD*32 , RRT*D , RRT*R and RRT* in U5 map

5.7 Simulation # 4

This last simulation is regarding the different RRT* with Motion Primitives implementations, considering the different sampling modes, as well as the original getNeighbors function (we will call it *old*), and the new shorter version. The analysis is extended to sampling without rewiring.

Given the amount of possible combinations above mentioned, instead of using the 10 queries for the test, we have limited the test to 3 representative start-goal queries for both maps. For the Aster map the selected queries are 1b, 2b and 3b (Figure 5.1), while for the U5 map the queries are: 1b, 2a and 3c (Figure 5.2). We used the motions primitives with 0.25m resolution with 16 angles for faster response.

The implementation has been modified to stop the solve function also when a first solution is found, instead of waiting until the maximum amount of iterations is reached. The reason of this is that each sampling method requires very different amount of iterations, therefore with this method it is possible to set a high enough *iterations* parameter for all the tests.

First we show the results of the original sampling mode implementation, then we continue with the results obtained from our novel sampling modes.

5.7.1 Original Sampling Mode

The first test considers the original Sampling Mode, which we call *allSpace*. The sample set is created with the min and max values of x and y of the map, and a step of 0.5m, which for the orientation of the samples we used 16 possibilities. In Figure 5.23, the red points represent the valid states sampled in the Aster map. With 6000 iterations the planner took around 300s in the different tests, but it was able to connect only one node as seen in the image.

The same test was performed with the U5 map, given the that it is smaller, then the sample set contains less states, which allowed to make more iterations in less time . In this case as seen in Figure 5.24, the valid states are shown in yellow. With 35000 iterations the planner was able to connect a total of 15 nodes in around 132s.

As mentioned in the previous chapter with this kind of maps it is not viable to use the allSpace Sampling Mode, as it wastes a lot of time rejecting far away samples in the space, and thus the tree extension is very slow. Which is the reason why the new methods were designed in this thesis.

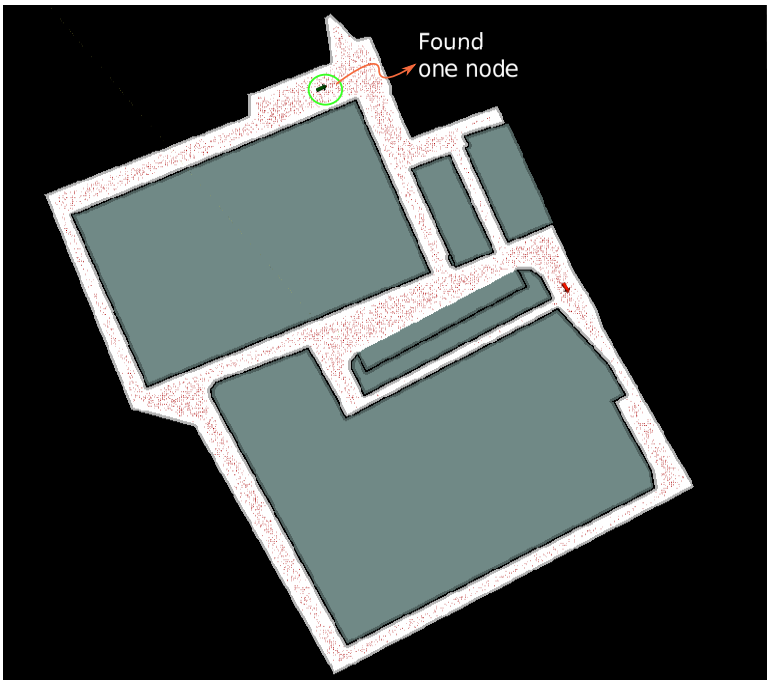


Figure 5.23: AllSpace Sampling Mode in the Aster Map

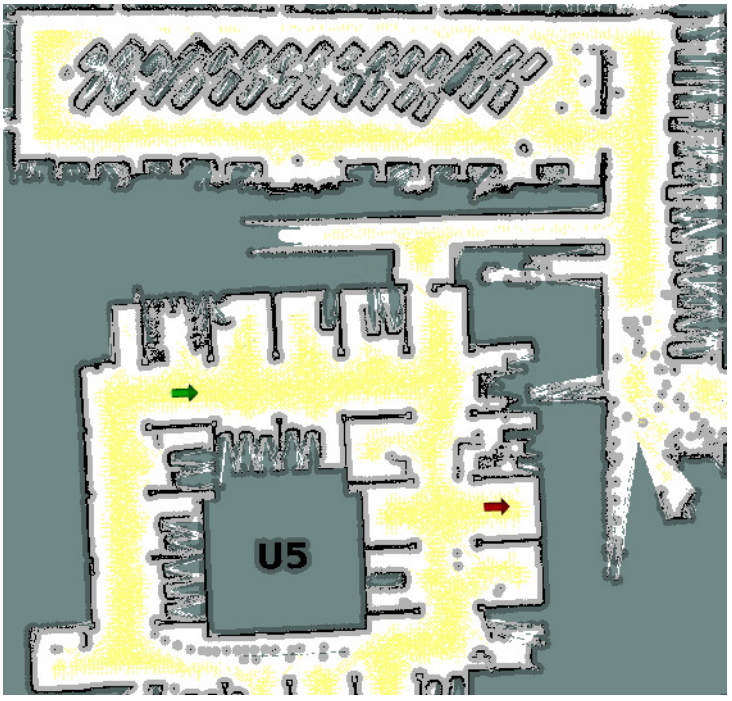


Figure 5.24: AllSpace Sampling Mode in the U5 Map

5.7.2 New Sampling Modes

From the performed tests the planning information has been collected and stored in the tables of Figure 5.25 and Figure 5.26. As it can be seen, for each query the collected data is: time, path length, rewired nodes, amount of iterations and the connected nodes in the tree. In the tables we have highlighted two topics or issues to discuss. In orange the number of iterations for the cube sampling mode, and in yellow some queries that were not solved correctly.

Cube Sampling Mode

With the Cube Sampling Mode, sometimes even when giving a big amount of iterations to use, the planner could not find a solution (as seen in Figure 5.27). There are two reasons why this happens; the first is that a lot of iterations are wasted because a primitive does not exist to connect to the tree the sampled node; this can be seen in the tables by comparing the amount of iterations and the amount of actual connected nodes.

The second reason is that we are not storing which samples have been already used, then there is a probability that the sampler takes and checks again already connected nodes. This leads to behaviors like the one seen in the image, where the tree gets stuck in the same area. Similarly to query 3b from the U5 map (Left image in Figure 5.27), the planner got into a local minima (nodes are surrounded by obstacles); the planner keeps trying to extend the tree, but the nodes closer to the goal ,according to the Euclidean distance, are not able to extend because of the obstacles, then the planner uses all the iterations without success of reaching the goal. The reason is that with this approach we do not set a node surrounded by obstacles as not expandable.

If the planning problem does not contain a path with the possibility to enter in local minima, then the Cube Sampling Mode performs better than the original method, i.e., *allSamples*, and it is able to find a solution in a much shorter time. As seen in Figure 5.28 query Aster 1b and U5 2a are solved correctly.

Random Primitive Sampling Mode

As explained in the previous chapter the Random Primitive Sampling Mode was generated with the idea of reducing both the time and the amount of iterations required for finding a solution in the Cube Mode. As seen from the data in the tables of Figure 5.25 and Figure 5.26 an improvement in the planning time was achieved. Nevertheless we still have the issue with certain queries like the 2b in the Aster map (Figure 5.29), for which the tree expansion was not successful to reach the goal, because as before the planner keeps trying to expand in

		U5 map				
		1b				
		time	length	rewired	iterations	new nodes
Cube	with old	15.15	46.13	34	433794	1345
	with new	11.62	47.17	2	434549	1402
	without	17.18	45.67	0	687496	1799
prim_rand	with old	11.55	46.59	333	2747	2711
	with new	1.76	48.01	39	2030	1992
	without	1.16	48.27	0	2481	2423
prim_store	with old	8.17	45.28	210	2020	2020
	with new	2.21	45.71	52	2383	2383
	without	1.32	47.91	0	2658	2658
		2b				
		time	length	rewired	iterations	new nodes
Cube	with old	12.16	35.13	10	228581	1076
	with new	5.45	34.84	1	174054	826
	without	6.05	35.22	0	202302	944
prim_rand	with old	6.99	35.11	176	1612	1574
	with new	1.39	34.76	20	1869	1780
	without	0.68	35.90	0	1564	1479
prim_store	with old	4.55	34.45	141	1165	1165
	with new	1.21	35.54	20	1398	1398
	without	0.90	37.15	0	1813	1813
		3b				
		time	length	rewired	iterations	new nodes
Cube	with old	167.94	0.00	15	3500000	1330
	with new	136.85	0.00	4	3500000	1135
	without	139.87	0.00	0	3500000	1237
prim_rand	with old	13.12	46.54	504	2610	2557
	with new	7.37	47.15	159	6614	6541
	without	1.38	50.33	0	2966	2886
prim_store	with old	307.31	0.00	927	4500	4500
	with new	7.02	48.86	85	6243	6243
	without	4.32	47.11	0	9545	9545

Figure 5.25: Planning data in U5

		Aster map				
		1b				
		time	length	rewired	iterations	new nodes
Cube	with old	21.00	75.34	94	589544	2260
	with new	21.60	76.59	8	722610	2194
	without	20.12	75.91	0	740013	2336
prim_rand	with old	19.17	73.41	839	4360	4286
	with new	3.22	76.17	66	4196	4112
	without	1.89	76.33	0	4365	4272
prim_store	with old	20.20	75.79	654	4438	4438
	with new	3.65	73.67	121	3828	3828
	without	1.97	75.47	0	4273	4273
		2b				
		time	length	rewired	iterations	new nodes
Cube	with old	27.17	100.28	103	857166	2865
	with new	22.25	102.38	19	832216	2823
	without	22.26	103.80	0	965731	3026
prim_rand	with old	92.64	0.00	3598	10000	8810
	with new	27.14	0.00	382	20000	17598
	without	17.39	0.00	0	40000	34217
prim_store	with old	28.34	100.44	908	5702	5702
	with new	5.52	104.34	132	6137	6137
	without	3.04	105.21	0	6475	6475
		3b				
		time	length	rewired	iterations	new nodes
Cube	with old	24.365546	95.736	78	615064	2664
	with new	19.113318	96.067	18	708981	2864
	without	15.926338	98.9	0	680464	2676
prim_rand	with old	16.47	95.09	847	4228	4206
	with new	3.28	96.41	89	4097	4065
	without	1.81	97.35	0	4063	4024
prim_store	with old	21.25	94.67	1082	4636	4636
	with new	4.17	99.17	130	4590	4590
	without	2.09	96.11	0	4402	4402

Figure 5.26: Planning data in Aster

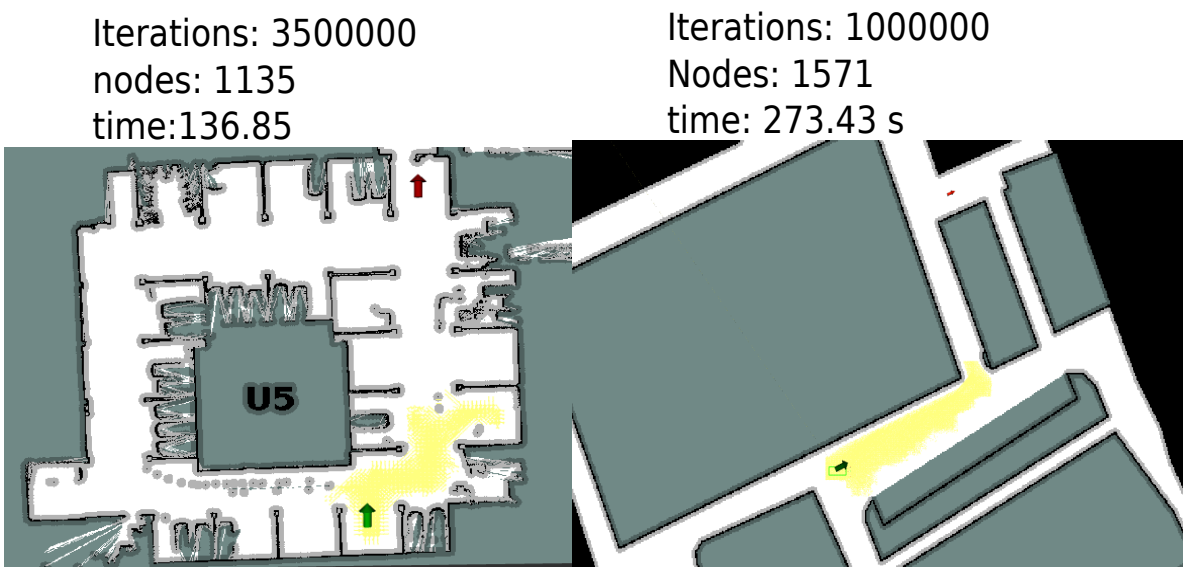


Figure 5.27: Sampling issue with the Cube Mode

areas where obstacles are present, and we have not set these nodes as not expandable.

However, in most of the cases a solution was correctly found with this Sampling Mode as seen by the examples in Figure 5.30.

General Comparison of the New Sampling Modes

After explaining the different behavior of the sampling modes, in this section a comparison of the length deviations from the optimal paths and time of the found solutions are shown in Figure 5.31 and Figure 5.32. We have removed the data from the queries 2b in Aster, and 3b in U5, because not all the planners managed to find a solution, and also the results from AD*16 were included, to use as reference of the optimal paths. The names in the plots are shorten for simplicity, the meanings are as follows:

C: Cube Sampling Mode

RP: Random Primitives Sampling Mode

SP: Random Primitives and Store Sampling Mode

old: Used original getNeighbors function for rewiring

new: Used new implementation of getNeighbors function for rewiring

no: No rewiring was used.

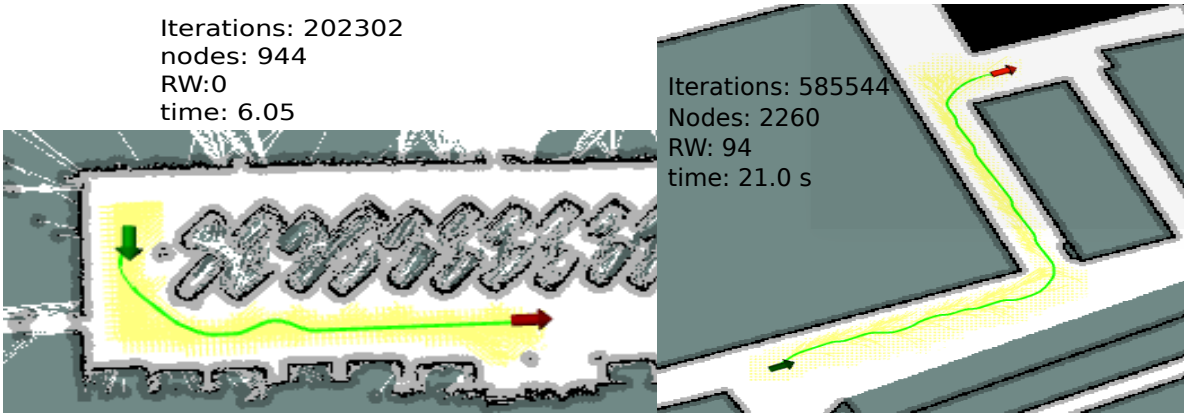


Figure 5.28: Planning Examples Solved with Cube Sampling Mode

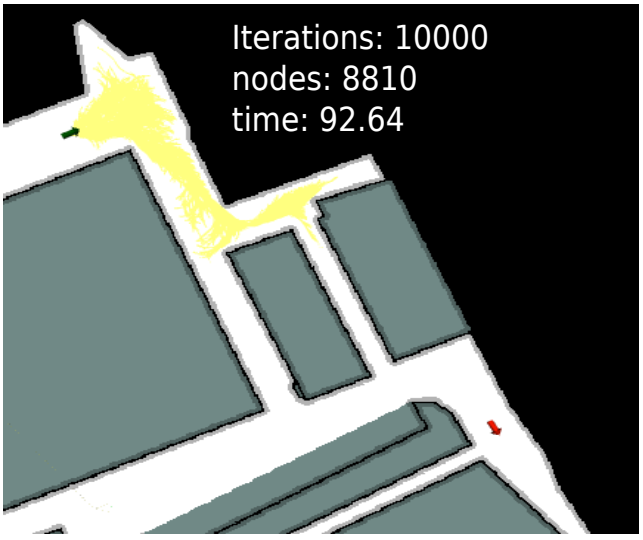


Figure 5.29: Sampling Issue with the Random Primitive Mode

It can be seen that the solutions of Cube Mode required more time than the rest of solutions. The solutions with the original getNeighbors function found more nodes to rewire, as seen from the data in the tables of Figure 5.25 and Figure 5.26, it shows that the original getNeighbors function finds more nodes in the vicinity, which leads to more time spent in the rewiring process, thus all the solutions with the original (old) rewiring method show higher planning times.

The solutions with the original rewiring obtain a smaller deviation of the path length respect to the optimal one, therefore there is a tradeoff between path length and the planning time, a selection must be made between choosing a shorter path, or less planning time. It can also be taken into account that a solution can be found faster with the new implementations,

Figure 5.30: Sampling Issue with the Random Primitive Mode

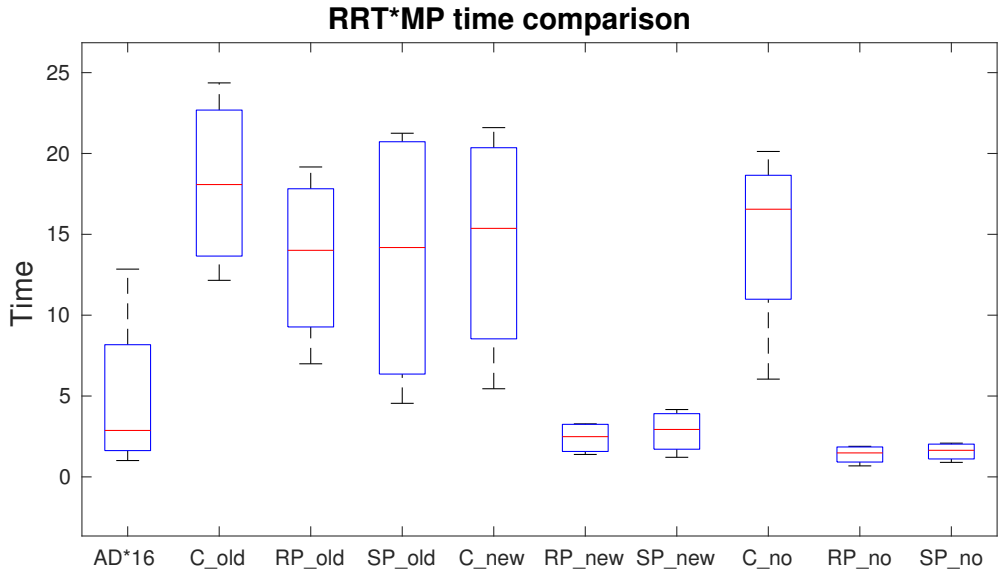


Figure 5.31: Time comparison of the different Sampling Modes

and can then take some more iterations in order to refine the path trajectory, while still taking less time than the original implementation.

Given that the sampling mode called Random Primitive and Store gave the best solutions in terms of planning time and path length, we have separated its solutions in a plot with AD*16 as presented in Figure 5.33. As it can be seen, by using the original getNeighbors function with the Random Primitive and Store Sampling Mode, the obtained paths are shorter than the solutions without rewiring, but the latter were found to be the fastest.

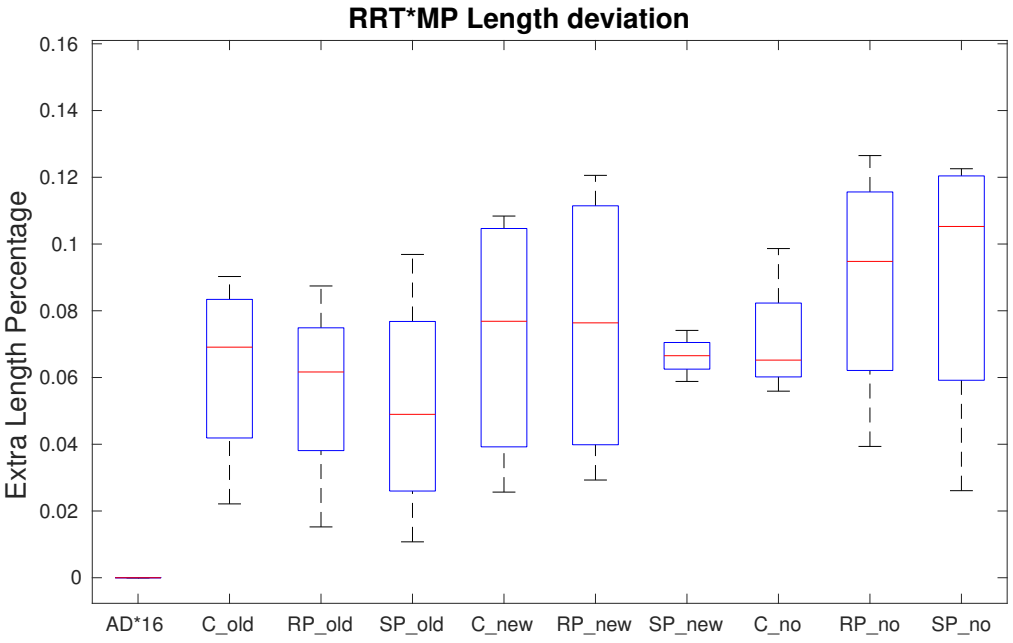
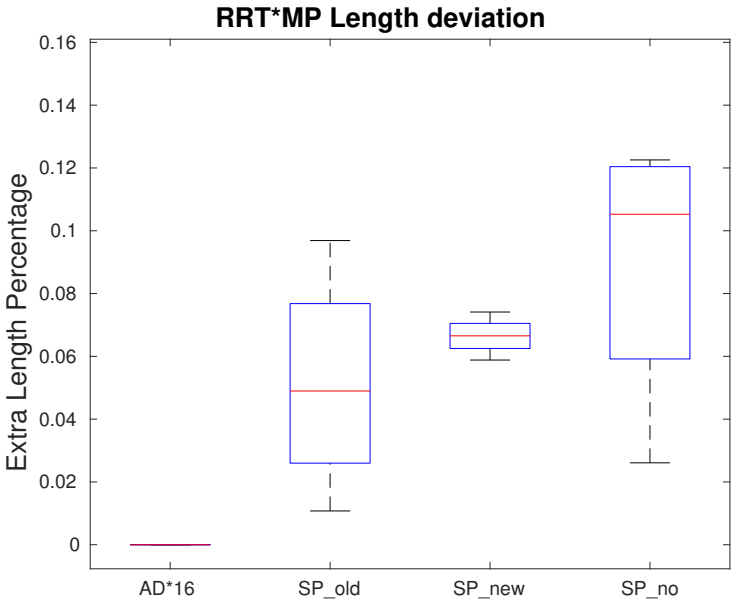


Figure 5.32: Length deviations of the different Sampling Modes

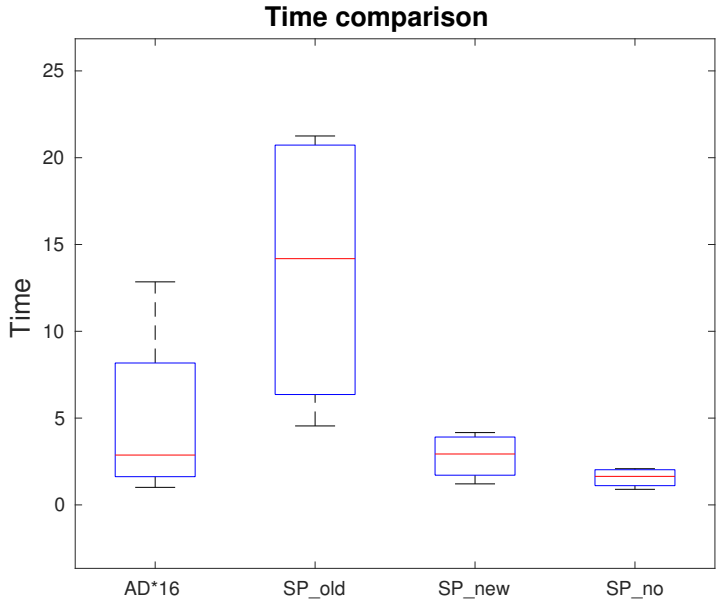
To have a visual overview of how is the path found by the RRT* with Motion Primitives compared with respect to the SBPL result, the Figure 5.34 shows two images; the one on the left shows the path found from the AD*16 and the SP_old as well as the time take for planning. The right image shows instead the path found by SP_new. The path shape with the *old* rewiring is more refined than the *new* one, but the planning time is around 5 times higher.

Important to mention that in every case analyzed the path feasibility can be ensured considering each primitive separate, as by definition a single primitive can be followed. While the whole path composed by the primitives is not guaranteed to be feasible, as the links between the primitives can require instantaneous changes in velocity, so the local planner still has to to adjustments from the expected trajectory.

This is a fact also for SBPL implementations. An advantage of working with motion primitives, is that the primitives could be modified as to also contain information regarding the velocity, meaning that the addition of new nodes in the case of OMPL would not also take into account the orientation of the state to check for primitives, but can also add constraints in terms of velocity, the same would happen with the grid search in SBPL. Furthermore, the isValid in OMPL function could not only check if the node is obstacle free, but can check if it is within the limits of velocity.



(a) Length



(b) Time

*Figure 5.33: AD*16 and OMPL+MP (SP)*

From the simulations, we can wrap up that, the SBPL planner is always able to provide a solution if one exists, and with enough time the solution is optimal. The selection of the

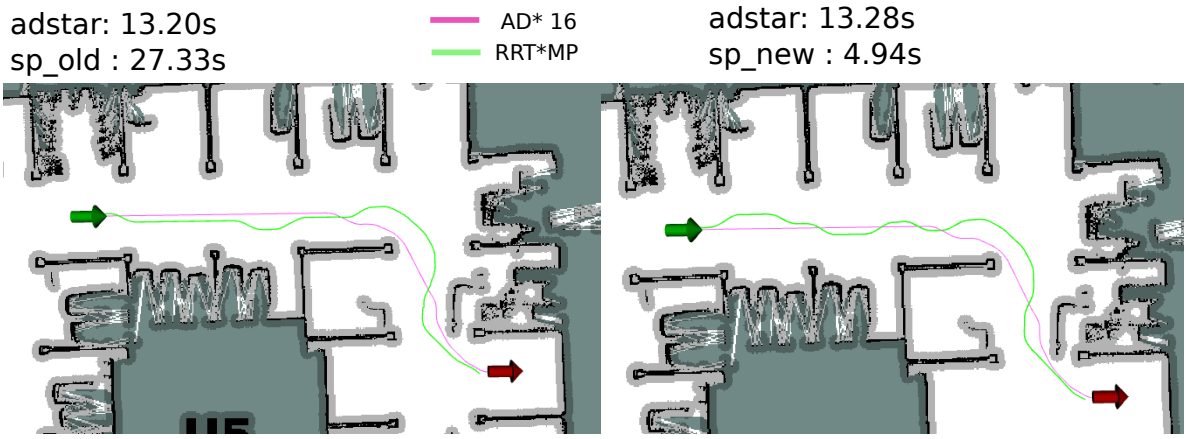


*Figure 5.34: AD*16 and OMPL+MP query 1b U5*

epsilon parameter and its decrease step influence the overall planning time. Regarding the RRT*_MotionPrimitives we found that with our improvements it is feasible to obtain a faster solution in realistic environments, but as the different sampling-based planners the obtained solution will be suboptimal.

Chapter 6

Conclusions and Future Work

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein, (attributed)

In this work we have implemented two global planners for autonomous vehicles, the first with search-based algorithms, while the second uses sampling-based algorithms; the resulting planners are able to take into account the kinematic vehicle constraints by using Ackerman motion primitives. To reach this goal the planners have been developed in the Robot Operating System framework using the SBPL and OMPL libraries.

Our implementation for the first planner uses motion primitives with 16 and 32 angles, and it also gives the possibility to plan without motion primitives (for this the SBPL library was modified) in order to compare AD* and ARA* algorithms with ROS move_base A* and OMPL RRT* and PRM* algorithms.

The second planner works with two different kind of state spaces, the first one is the normal SE2 space, while the second is a Motion Primitives State Space. We have improved from the work by Basak Sakak to include our motion primitives into the original implementation, and we have designed three sampling modes.

To evaluate and compare the performance of the planners a series of planning queries were defined in two different maps and the simulations were divided by categories. The first category was a comparison between A*, AD* and ARA* without motion primitives, which showed that in terms of time and length the A* has a better performance, but AD* and ARA* have the advantage of replanning for dynamic environments.

The second category is a comparison of A* and the Sampling-Based planners including Dubins and Reeds-Shepp spaces. The results showed that even though the solutions given by

RRT* Dubins where not faster nor smaller, they had the advantage of been smoother to be transverse by an Ackerman vehicle.

The third category compared RRT*D, RRT*R and AD* with motion primitives with 16 and 32 angles. Results show that having primitives with more angles has two consequences, it usually takes longer to find a solution, but can find paths with shorter length. On the other side, having too few primitives as Dubins, can lead to obtain longer paths and unnecessary turns, plus it was observed in the results that the OMPL implementation of the Dubins and Reeds-Shepp spaces tend to constrain the tree expansion as it is restricted to predefined movements.

The last category was a comparison between the different sampling methods developed for the RRT*_MotionPrimitives planner using primitives with 16 angles. The original sampling method required a considerable amount of time and had issues growing the tree as many samples were rejected. In practice the implementation was not usable for a real vehicle application.

The Cube Sampling Mode showed improvements from the original, as more nodes are positively connected to the tree, but still requires a lot of iterations and it is computationally intractable. The Random Primitive Sampling Mode improved from the Cube by decreasing the amount of iterations as every sample is linked to a certain motion primitive. The Random Primitive and Store Sampling Mode improved from the other two by keeping track of the used primitives, and also by adding the possibility to escape local minima by setting the trapped nodes as not expandable. The solutions of the last mode were acceptable both in planning time and in path length as compared to the AD*16. solutions

Finally, the different options for rewiring were tested and a tradeoff was found between the path length and the time needed to find a solution. The more nodes are rewired the more time is needed for planning, but the solution might also not change significantly.

Regarding future work for this thesis, the most important one would be the inclusion of velocity constraints in the motion primitives as this would guarantee the feasibility of the whole path, since with only kinematic constraints by definition a single primitive can be followed, but the links between them can require instantaneous changes in velocity. By adding velocity constraints in the primitives the feasibility can be guaranteed.

Another extension can be added to the distance measurement of the nodes in the tree to the goal. In the current implementation the Euclidean distance is used, which does not consider the obstacles in between the tree nodes and the goal, it is only a distance of the states in the space. Instead since our environment is represented as a map, it would be possible to compute the distance from the different states with an algorithm like Dijkstra, by means of

the grid we can account for the obstacles between the tree nodes and the goal state, and have the real distance in terms of cells in the map.

As a final remark is important to notice that even though the implementation was done and tested with Ackerman motion primitives, it can also be extended for different vehicles, with only the requirement of computing the motion primitives for the specific vehicle, e.g., differential drive or skid steering.

Bibliography

[1] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[2] G. Bardaro, A. Semprebon, and M. Matteucci, "Aadl for robotics: a general approach for system architecture modeling and code generation," *Journal of Software Engineering for Robotics*, vol. 1, no. 8, pp. 32–44, 12 2017.

[3] Search-based planning laboratory. [Online]. Available: http://sbpl.net

[4] B. Sakcak, "Optimal kinodynamic planning for autonomous vehicles," Ph.D. dissertation, Politecnico di Milano, 2017.

[5] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Commun. ACM*, vol. 22, no. 10, pp. 560–570, Oct. 1979.

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[7] R. A. Brooks and T. Lozano-Perez, "A subdivision algorithm in configuration space for findpath with rotation," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, ser. IJCAI'83. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1983, pp. 799–806.

[8] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, Apr. 1986.

[9] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau, "Path planning: A 2013 survey," in *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*, Oct 2013, pp. 1–8.

[10] S. M. LaValle, *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006.

[11] D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristic-based path planning," in *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling*, Pittsburgh, PA, June 2005.

[12] E. A. Hansen and R. Zhou, "Anytime heuristic search," *Journal of Artificial Intelligence Research*, vol. 28, pp. 267–297, 2007.

[13] M. Likhachev, G. Gordon, and S. Thrun, "Ara*: Formal analysis," 08 2003.

[14] J. Van Den Berg, R. Shah, A. Huang, and K. Goldberg, "Ana*: Anytime nonparametric a*," in *Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence*, 2011.

[15] A. Stentz, "The d* algorithm for real-time planning of optimal traverses," Carnegie Mellon University, Tech. Rep., 1994.

[16] M. Likhachev, D. Ferguson, G. Gordon, A. T. Stentz, and S. Thrun, "Anytime dynamic a*: An anytime, replanning algorithm," in *Proceedings of the International Conference on Automated Planning and Scheduling*, Pittsburgh, PA, June 2005.

[17] S. R. Lindemann and S. M. LaValle, *Current Issues in Sampling-Based Motion Planning.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 36–54.

[18] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan, "A random sampling scheme for path planning," in *Robotics Research*, G. Giralt and G. Hirzinger, Eds. London: Springer London, 1996, pp. 249–264.

[19] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in *IEEE International Conference on Robotics and Automation*, 1996, pp. 566–580.

[20] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[21] I. Noreen, A. Khan, and Z. Habib, "A comparison of rrt, rrt* and rrt*-smart path planning algorithms," *International Journal of Computer Science and Network Security*, vol. 16, no. 10, p. 20, 2016.

[22] J.-P. Laumond, S. Sekhavat, and F. Lamiraux, *Guidelines in Nonholonomic Motion Planning for Mobile Robots*, 04 2006, vol. 299.

[23] A. Conforto, "On the development of a search-based trajectory planner for an ackermann vehicle in rough terrains," Master's thesis, Politecnico di Milano, 2014.

[24] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957.

[25] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards." *Pacific J. Math.*, vol. 145, no. 2, pp. 367–393, 1990.

[26] M. Moll. Geometric planning for car-like vehicles mark moll. [Online]. Available: http://ompl.kavrakilab.org/2012/03/18/geometric-planning-for-car-like-vehicles.html

[27] M. Likhachev and A. Stentz, "R* search," in *Proceedings of the 23rd National Conference on Artificial Intelligence*, ser. AAAI'08. AAAI Press, 2008, pp. 344–350.

[28] M. Likhachev and A. Stentz, "PPCP: efficient probabilistic planning with clear preferences in partially-known environments," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence*, 2006, pp. 860–867.

[29] M. Pivtoraiko and A. Kelly, "Generating near minimal spanning control sets for constrained motion planning in discrete state spaces," in *International Conference on Intelligent Robots and Systems*, Aug 2005, pp. 3231–3237.

[30] Kavraki Lab, *Open Motion Planning Library:A Primer*. Rice University, 2017.

[31] Robot operating system. Accessed on 05.11.2017. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[32] A. Koubaa, *Robot Operating System (ROS): The Complete Reference (Volume 1)*, ser. Studies in Computational Intelligence. Springer, 2016, vol. 625.

[33] nav_core. Accessed on 17.11.2017. [Online]. Available: http://wiki.ros.org/nav_core

[34] costmap_2d. Accessed on 25.11.2017. [Online]. Available: http://wiki.ros.org/costmap_2d

[35] V. Ressa, "Diffdrive: a global path planner for autonomous navigation of differential drive robots," Master's thesis, Politecnico di Milano, 2015.

[36] R. Vaughan, "Massively multi-robot simulation in stage," *Swarm Intelligence*, vol. 2, no. 2, pp. 189–208, Dec 2008.

Appendix A

User Manual

The software has been stored in the AIRLab Bitbucket repository. It has all been developed under the Linux distribution: Ubuntu Xenial (16.04 LTS). The ROS version used is Kinetic Kame, the tenth ROS distribution released in 2016.

A.1 Dependencies

The first step is to install ROS, to do so it is recommended to follow the instructions on the ROS web page to install the Desktop-Full version, and then create a catkin workspace [31].

Following this, clone the following packages from the Airlab Repository in the workspace:

- planners_ros : this package contains the implementation of the Search-Based planners.
- stage_simulator_config : this package holds the simulation world. It uses the Stageros package.
- ompl_global_planner: this package holds the implementation of the Sampled-Based planners
- global_planner: this package contains elements used by both OMPL and SBPL global planner nodes, like the launch files for the costmap, and the test files for publishing goals and saving the results.

Furthermore, for the usage of the simulation requires to merge the three lasers from the robot model, for this the ira_laser_tools package is used, it requires libopenni and libpcl. This can be obtained as seen here:

```
$ git clone https://github.com/iralabdisco/ira_laser_tools.git
$ sudo apt-get install libopenni2-dev
$ sudo apt install libpcl1.7
```

A.2 Compiling the libraries

Before compiling the ROS nodes, both SBPL and OMPL libraries must be built and installed. The following steps must be followed for each of the libraries.

1. Open a terminal, change directory until the root folder of the node is reached (either the planners_ros or the ompl_global_planner).
2. Move to the library folder (sbpl or ompl-1.3.2-Source).
3. Execute the following commands:

   ```
   $ mkdir build
   $ cd build
   $ cmake ..
   $ make
   $ sudo make install
   ```

   The last command must be executed so the library can be reached by ROS.

Once both libraries have been built, the ROS workspace can be compiled with `catkin_make`.

A.3 Running the code

For running the global planner nodes the costmap and the simulator must be launched first.

```
roslaunch global_planner costmap_sbpl.launch
# or
roslaunch global_planner costmap_ompl.launch
```

The used map is the same for both planners, what changes in the launch file is the configuration file of Rviz (ROS 3D visualizer), to facilitate the use.

The launch file shown in the Listing 1 runs the following nodes:

1. The simulator Stageros, which requires to have a stage world file.
2. Map Server, takes as argument a yaml file with the map information.
3. Static Transform between the map and odom. Set as the robot initial position.
4. Costmap_2D as the global costmap, it requires a yaml file with the parameters for the global costmap configuration. The meaning of these parameters can be found in the costmap_2d ROS page.
5. Rviz, with its configuration file.

Once the simulator has been launch the laser merger can also be launched as seen next, this launch file runs the laserscan_multi_merger node from ira_laser_tools, which takes three laser scan topics and merges them into /scan to be used by the costmap_2d.

```
roslaunch stage_simulator_config laserscan_multi_merger.launch
```

```
<launch>
<node pkg="ira_laser_tools" name="laserscan_multi_merger"
 ↪  type="laserscan_multi_merger" output="screen">
        <param name="destination_frame" value="/base_link"/>
        <param name="cloud_destination_topic" value="/merged_cloud"/>
        <param name="scan_destination_topic" value="/scan"/>
    <param name="laserscan_topics" value ="/base_scan_0 /base_scan_1
     ↪  /base_scan_2" />
</node>
</launch>
```

Listing 1 Costmap launch File

```
<launch>

  <param name="/use_sim_time" value="true"/>

  <!--  ************** Stage Simulator ***************  -->
  <node pkg="stage_ros" type="stageros" name="stageros" args="$(find
   ↪  stage_simulator_config)/stage/aster.world"/>

  <!--  ************** Map Server **************  -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
   ↪  stage_simulator_config)/maps/aster_map.yaml" output="screen">
          <param name="frame_id" value="/map"/>
  </node>

  <!--  Publish a static transform between odom and map  -->
  <node pkg="tf" type="static_transform_publisher"
   ↪  name="static_map_transform" args="87.0 -106.0 0 0 0 0 map odom 100" />

  <!--  ************** Global Costmap ***************  -->
  <!-- Publishes the voxel grid to rviz for display -->
  <node pkg="costmap_2d" type="costmap_2d_markers"
   ↪  name="voxel_visualizer_global">
          <remap from="voxel_grid" to="costmap/voxel_grid"/>
  </node>

  <!-- Run the global costmap node -->
  <node name="global" pkg="costmap_2d" type="costmap_2d_node" >
          <rosparam file="$(find
           ↪  global_planner)/cfg/global_costmap_params.yaml" command="load"
           ↪  ns="costmap" />
  </node>

  <!--  ***************** Visualisation ****************  -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
   ↪  planners)/cfg/rviz_sbpl.rviz" />

</launch>
```

Finally the global planner can be launch as well, each global planner can be launched with a parameters file, to be specified by the user, otherwise default values are used. The next subsections explain the parameters to be set for each global planner.

```
roslaunch planners global_sbpl.launch
#or
roslaunch ompl_global_planner global_ompl.launch
```

```
<launch>
<node name="ompl_global_planner" pkg="ompl_global_planner"
 ↪  type="global_planner.ompl" output="screen">
<rosparam file="$(find ompl_global_planner)/cfg/ompl_params.yaml"
 ↪  command="load" />
</node>
</launch>
```

A.3.1 SBPL Global Planner

For setting up the SBPL planners first a configuration file must be used, the first parameters represent information about the environment, meaning the amount of variables to use, either 3 or 5, only the version for 3 was tested since we did not have motion primitives with 5 variables. Then also the relative path to the motion primitives file to use must be set.

Following some vehicle parameters are required, this concerns the shape, width, length and the maximum speeds of the vehicle.

The planning setup requires setting the possibility to update or not the costmap, if manual refinement is wanted both the bFirstSolution and bRefinement varaibles must be set to true. Then the wanted planner can be set, first setting if the planning is backwards or forward, also the planner type must be set. It is possible to set adstar and arastar both for forward and backward planning, plus they accept manual refinement. But if anastar or anastardouble are wanted only backward planning is allowed and manual refinement was not implemented for them. Afterwards the maximum planning time must be set, as well as the ϵ and the decrease.

In the case of the map parameters the most important correspond to the values in the map that correspond to obstacles, or inscribed obstacles. The costmap node will receive the measurements of the vehicle, as well as how much should the radius be inflated, and it publishes a grid with the values from 0-100, internally we translate the values to 0-255, therefore, having 255 as unknown, 254 as obstacle, 253 as inscribed obstacle, and 128 as possibly inscribed.

The parameters are described in the Table A.1.

Table A.1: Customizable parameters of SBPL planners

Parameter	**Meaning**	**Possible Values**
dimensions	Amount of variables that define the environment	3 or 5
motPrimString	Relative path to the primitives file	string
robotShape	The shape of the robot	rectangular or circular
robot_width	The width of the vehicle in meters	float ≥ 0.01
robot_length	The length of the vehicle in meters	float ≥ 0.01
v_max	The maximum linear velocity feasible by the vehicle	float ≥ 0.01
omega_max	The maximum angular velocity feasible by the vehicle	float ≥ 0.01
bOnlyGlobalCostmap	Flag to use only the static map information	bool
bRefinement	Flag to use manual refinement	bool
bFirstSolutionp	Flag to plan without refinement	bool
bForwardSearch	Flag to set backward or forward search	bool
planner_type	The algorithm to be used	adstar or arastar
allocated_time	Allocated time for the planner to find a solution, in seconds	float ≥ 1.0
initial_epsilon	Allocated time for the planner to find a solution, in seconds	float ≥ 1.0
epsilon_decrease_step	Decrease step for the inflation factor to refine the solution	float ≥ 0.1
resolutionTolerance	Tolerance for the resolution of the motion primitives and the costmap	float ≥ 0
obsthresh	Threshold to identify obstacles, related to the costmap information	int ≥ 1
cost_inscribed_thresh	Refers to the cells that are closer than the robot inscribed radius to obstacles	int $\leq$ *obsthresh*
cost_possibly_inscribed_thresh	Possible also inscribed	int $\leq$ cost_inscribed

A.3.2 OMPL Global Planner

For explaining the OMPL planners we can divide in three sections, first the general parameters, and then one category for the usual OMPL planners, and the last one for the parameters needed for RRTStar+MotionPrimitivies.

The normal parameters correspond to the vehicle parameters and the map parameters, that have already been described in the SBPL parameters. The OMPL Parameters for the planning setup are shown in the Table A.2

Table A.2: Planning parameters of OMPL planners

Parameter	**Meaning**	**Possible Values**
planner_type	The planner algorithm to be used	rrt, prm, rrt-star, prmstar, rrtstarmp
space	Definition of the state space	se2 or mprim
dubins	Flag to set the dubins space	bool
reedsShepp	Flag to set Reeds Shepp space	bool
turningRadius	Definition of the maximum turning radius of the vehicle	float
longest_segment	Longest segment of a motion to be checked	float ≥ 0.01
time	Time available for the planner	float ≥ 0.0
extra_trial	If an exact solution is not found then how many extra trial should be taken	int
only_exact	Flag for only accepting an exact solution	bool

For the implemented OMPL with motion primitives, some primitives parameters must be set, as the location of the file, the number of angles of the primitives and the resolution of them. Then the sampling mode can be selected, as mentioned in section 4.5.1 there are four sampling modes, from which to select, this is done by means of flags. Furthermore there is the possibility to select which rewiring mode to select . The parameters are explained in the Table A.3.

Table A.3: Planning parameters of OMPL+MP

Parameter	**Meaning**	**Possible Values**
motPrimString	The relative path to the motion primitives file to use	string
max_distance	Maximum distance of the motion primitives	float
search_resolution	Resolution of the motion primitives	float
withRW	Flag to set replanning for the planning	bool
costlength	Flag to calculate the cost respect to distance, or time	bool
num_thetas	Number of angles of the motion primitives	16 or 32
all_space	Flag to select the all space sampling mode	bool
cube	Flag to select the cube sampling mode	bool
prim_random	Flag to select the random primitives sampling mode	bool
prim_store	Flag to select the random primitives and store sampling mode	bool
num_distance	Value to select the closest nodes to the goal	int
searchthreshold	Mixing parameter for selecting from sampling from the tree and for the distances	float [0,1]
newRW	Flag to select the compressed rewiring option	bool
max_iterations	Maximum number of iterations for the planner	int