POLITECNICO DI MILANO
Master of Science in Automation and Control Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

BENCHMARKING OF ROBOT OPERATING SYSTEM (ROS) GLOBAL PLANNERS

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

Supervisor: Prof. Matteo Matteucci
Co-Supervisor: Enrico Piazza

Thesis by:
Furkan Çağrı Elitok
10654840

Academic Year 2019-2020

To my family...

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The motion planning problem is an important problem in robotics. Motion Planning can be defined as a computational search problem to find a sequence of valid configurations for a collision-free path from an initial state to a final state. Moreover, benchmarking is the practice of comparing.

In this thesis, the global planning problem is handled for unicycle and Ackerman steering vehicles for different environments in a (x, y, θ) spaces. Different types of metrics are defined for comparing global planning problems for various planning algorithms. Environments with diverse features are used to see the performance of planning algorithms. Path plans are made according to many starting points and results are obtained for defined metrics. Finally, results are published and commented on clearly.

Sommario

Il problema della pianificazione del movimento è un problema importante nella robotica. La pianificazione del movimento può essere definita come un problema di ricerca computazionale per trovare una sequenza di configurazioni valide per un percorso privo di collisioni da uno stato iniziale a uno stato finale. Inoltre, il benchmarking è la pratica del confronto.

In questa tesi, il problema della pianificazione globale è gestito per monociclo e veicoli sterzanti Ackerman per ambienti diversi in spazi (x, y, θ). Vengono definiti diversi tipi di metriche per confrontare i problemi di pianificazione globale per vari algoritmi di pianificazione. Ambienti con caratteristiche diverse vengono utilizzati per valutare le prestazioni degli algoritmi di pianificazione. I piani di percorso vengono realizzati in base a molti punti di partenza e si ottengono risultati per metriche definite. Infine, i risultati vengono pubblicati e commentati in modo chiaro.

Acknowledgements

First, I must express my very profound gratitude to my parents, my sister, and my brother. I am grateful for their emotional support during these two and a half years abroad (especially in COVID-19 times). This accomplishment would not have been possible if my family didn't stand behind me.

Last but not least, I would like to thank Prof. Matteo Matteucci, for giving me the opportunity to work on the thesis and opening to doors of AI & R Lab for me, besides for the challenge to keep pushing me to learn more. As well as special thanks to Enrico Piazza for giving me time, paying attention to all my steps, solving my problems, and answering my questions all the time.

Chapter 1

Introduction

Nowadays, robotics is a fundamental topic for academic life and in the field of engineering. Each year many academic papers have been publishing, and robotics enters our lives more deeply. From autonomous cars to industrial robots, robotics has a very wide field of study. In this comprehensive workplace, motion planning has an important role.

Motion Planning can be defined as a computational search problem to find a sequence of valid configurations for a collision-free path from an initial state to a final state. Motion planning includes different types of aspects like a variety of path planning algorithms also diverse environment types, such as narrow environments (e.g., a studio in the apartment), indoor, outdoor environments, wide-open environments, and so on. Moreover, this problem includes various models of robots, for instance, mobile robots and robot arms.

Path planning is essential for moving from one point to another point. In path planning, there are two main parts basically, which work together. One is called global planning, and the other is called local planning. Although, global planning has generally known about the map and according to overall map planner can generate its global plan. Local planning has only known some part of the map which is defined some distances from mobile robot.

For mobile robots, path planning has a significant role. While planning, the robot must care about robot kinematics and environment specifications. For example, making a plan can be easier to make a path plan for open spaces, however parking a car in a busy area can be tricky. Different type of combinations is an open area for research.

Motion planning needs a benchmarking approach to compare which path planning algorithm is feasible or related to decided tasks or desires. This

thesis focuses on how to make some benchmark runs for different types of environments using search-based and sampling-based planning algorithms. In particular, we focus on algorithms with unicycle or Ackerman steering vehicle models and compare its results according to related metrics. For this work, we use Robot Operating System (ROS). We wrote execution and run scripts for dealing with different configurations and different maps. Furthermore, we wrote a supervisor script for initializing environments which helps to send initial and goal points, and activate path planning algorithms. We test a set of libraries that solves the path planning problem: ROS global_planner library [42], Search Based Planning Library (SBPL) [6] and Open Motion Planning Library (OMPL) [4]. Our experiments look into conditions such as Voronoi distance of path, time, Euclidean distance of path by using supervisor as a .csv file.

Within this work, we assume vehicles that have some motion primitives according to differential drive and Ackerman steering vehicle robot kinematic constraints. Moreover, we used various environments from very small environments to large and open environments. In this way, we compared two different vehicles in different environments.

The thesis comprises of 9 chapters, the 9 chapters that follow this introduction are outlined below:

In Chapter 2 introduces related work and historical notes. After, we provide some background on the basics motion planning which describes the cornerstones of the problem.

In Chapter 3, we present the reference vehicle and its specifications.

In Chapter 4, we present the environments and their specifications with the Voronoi description.

In Chapter 5, we give general information about state lattice. We elaborate on differential drive and car-like vehicles motion primitive generation.

In Chapter 6, we give description of the Robot Operating System (ROS) and used libraries.

In Chapter 7, we give a more in-depth description of the script designed.

In Chapter 8, we evaluate the result of the work done with different test cases.

In Chapter 9, we conclude the work.

Chapter 2

The State of the Art

This chapter illustrates the state of the art of the benchmarking for motion planning and the historical background of the path planning algorithms. Starting with the related works about benchmarking for motion planning followed by path planning notes are briefly mentioned. Later, the description of path planning algorithms and their most favored methods are given.

2.1 Related Work

Motion planning is the problem that requires finding a feasible path from any start point to goal point in a space dealing with different constraints. From puzzle games to robotics, motion planning has a broad usage area. Under this broad motion planning problem definition and usage area, we need some benchmarking method to compare and put some restrictions for this wide research area so people can select better configurations according to their research or works. For example, one factory owner can check the results of motion planning benchmarking, and can choose mobile robot is the best according to particular needs.

When we checked the literature, we found research about motion planning benchmarking. According to Cohen et al. the alpha puzzle, which has been used to test the performance of motion planners for narrow passage issues, is a widely cited benchmark [12]. Baltes shortly define benchmark as a measurement of performance for a specific task and touched on its background, and metrics including a brief introduction with popular robotic games, commonly use mobile robot comparison (e.g., wheelchair robot or car-like robot) and with static and dynamic path planning differences [8]. Similarly, Calisi et al. work on an objective that is to provide

a common testing system for quantitative assessment and comparison for autonomous navigation and mobile robots of various motion strategies [11]. Mobile manipulators' path planning and obstacle avoidance are examined and presented in [36] with state-of-the-art benchmarking in a static environment. Another different work is including the experimental procedure for real indoor navigation in robotic to show advantages of the benchmark [44]. Besides these, Moll et al. create a software that can make benchmarking with some metrics using OMPL for general purposes like toy problem or more complex real-world scenarios (e.g., moving a table from one room to another room) [35]. A series of benchmark for testing sampling base planning algorithms according to robot including different tasks and its metric results are given in [12]. On the other hand, there are some works about benchmarking local planning. In [47], Wen et al. made a benchmarking for local planning algorithms according to various environments.

Our work is different from previous work. We create an open-source benchmarking and metric calculation software. This software can work for the differential drive and car-like model vehicles that include motion primitives. For car-like vehicles, we also create an open-source MatLab software package that can create new primitives according to car length and maximum turning radius. Moreover, we compare different planning algorithms (e.g., search-based planning and sampling-based planning). In this work, we used Robot Operation System (ROS), which allows us accessibility and flexibility. Thanks to ROS, we tried various environments for mobile robots with different start and goal positions.

2.2 Historical Notes About Motion Planning

In literature, the motion planning problem is knowns as a path planning problem (also known Piano Mover's Problem). According to Schwartz and Sharir, Piano Mover's Problem is defined as how to move a rigid body from one point to another point without hitting anything [41]. Intuitively, this can be imagined by assuming that there is a piano in the home, and we are trying to move it from one room to another room without touching any other furniture. Here the point is finding a feasible path for this rigid body. Moreover, we can increase path planning examples like Alpha Puzzle, an automotive assembly to solve problems, navigating mobile robots, and so on. These examples show several problems that are waiting to solve using the motion planning approach.

As LaValle mentioned "Historically, planning has been considered different from problem solving; however, the distinction seems to have faded

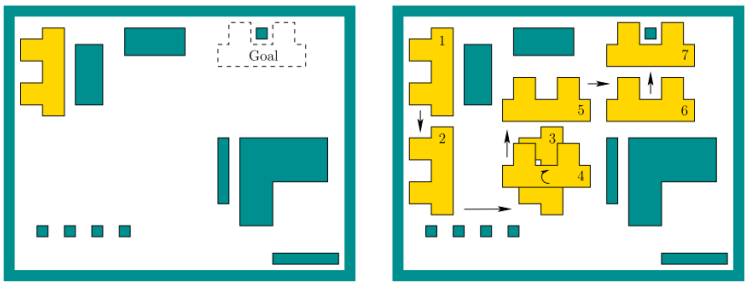


Figure 2.1: Piano Mover's Problem

away in recent years." [26]. Thanks to the development of calculus and the increasing computational power of computers, many planning algorithms have been developed for from robotic applications to protein folding [26].

2.3 Motion Planning Algorithms

In this section, we briefly talked about some main path planning algorithms. We mentioned path planning into two categories. One is Search-Based Planning, and the other one is Sampling-Based Planning. Before talking about algorithms also, we touched on important contents definitions that related to path planning.

Basic contents of planning include [26]:

Environment: The environment is also known as a map. In this area, according to given tasks, the robot can move from one point to another point. This environment can include some obstacles like a wall or table that are stable obstacles. On the other hand, it can include some dynamic obstacles like moving robots or humans.

State: Inside the environment, all the possible robot situations compose the state space of these environments. Every element of this state space is called a state. In Cartesian coordinate environment it can composed by the position of the robot (x, y) and its orientation (θ).

Initial and Goal States: For creating a plan, the robot needs one start and one end point to follow. These points are represented by initial and goal states.

Action: Switching from one state to another is done via actions. Moreover, actions are the control elements of the plan.

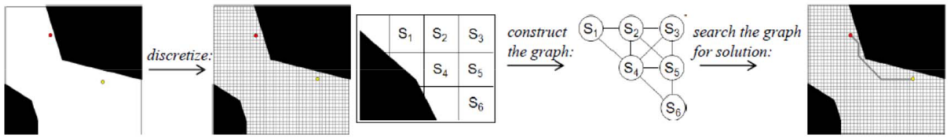


Figure 2.2: 2D grid-based graph representation for 2D search-based planning

Plan: Between given initial and goal states, a feasible sequence of action creations are called a plan.

Time: Passing duration while creating a feasible plan can be defined time.

2.3.1 Search-Based Planning

Search-Based Planning (Figure 2.2) is one of the most used algorithms in the motion planning field of the robotics. Search-based planning can be defined as a type of motion planning that uses methods of graph search to compute paths or trajectories over a discrete problem representation [6]. According to this definition, we can understand that there are objectives. First, we need to the discretized environment. Then, given a discrete graph, we need to find the solution from one point to another. These are shown in two main problems of Search-based motion planning algorithms.

These sorts of algorithms are also called heuristic graph search algorithms because they include heuristic functions, and search inside the graph. The heuristic function is shown as h(s), and it gives the estimated cost value from any state to the goal state. It is considerably significant while the planner is trying to find an optimal plan. If the heuristic function is admissible, the planner returns an optimal path. Otherwise, it will return a sub-optimal path. Thus, when one is deciding about heuristic function, one must be careful. One of the most basic cost values can be defined as Euclidean distance to the goal state.

Before giving basic search-based algorithms, we can define some important concepts that are used in search-based algorithm pseudo-code so the explanation can be clear.

s: State from creating a discrete environment (graph). This state must be reachable and feasible according to the robot dynamic and environment specifications.

s_{start}: Start state.

s_{goal}: Goal state

S: All states are the collection in the graph.

$succ(s)$: Mentioned the set of all succeeds of all s states inside all state collections of S.

$c(s, s')$: Denotes cost of the transition from state s to further state s′. s and s′ must be inside S and s′ must be inside of the succeed cluster.

Cost of path: Collecting costs s_{start} to s_{goal} gives cost of path.

$C^*(s, s')$: Optimal cost of path.

$g(s)$: Real cost of the path from s_{start} to s state.

$h(s)$: Heuristic cost of the path from s state to s_{goal} state.

$f(s)$: $f(s) = g(s)+h(s)$ that is node evaluation function. This function gives cost of s_{start} to s_{goal} with s that is the next node in the path and s node's heuristic cost value to s_{goal}. If heuristic value admissible, meaning that search-based algorithm can find least cost path from stat to goal that also means it is optimal.

According to Ferguson et al. for the computation of optimum paths on a weighted graph, several classical graph search algorithms have been developed; Dijkstra's algorithm and A* are two common ones [14].

Dijkstra Algorithms

Dijkstra (Algorithm[1] 1) one of the most popular and easiest path finding algorithm. Furthermore, Paden et al. said that the Dijkstra algorithm is currently the most known algorithm for finding the shortest paths in a graph [37]. Dijkstra algorithm finds shortest path from one point (or can be called node in graph) to another point (in Figure[1] 2.3). This algorithm is conceived by E. W. Dijkstra, who gave his name in 1956. According to Dijkstra, algorithm can find the optimal or shortest path between given two nodes inside a network or graph [13]. Detailed explanation about Dijkstra algorithm is given by Mehlhorn et al. and it can be found in [33]. The algorithm starts with the given discrete environment that is called a graph. The graph is composed of edges and nodes. In this graph, all nodes are marked as unvisited nodes. While search starts initial node takes zero cost,

[1] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

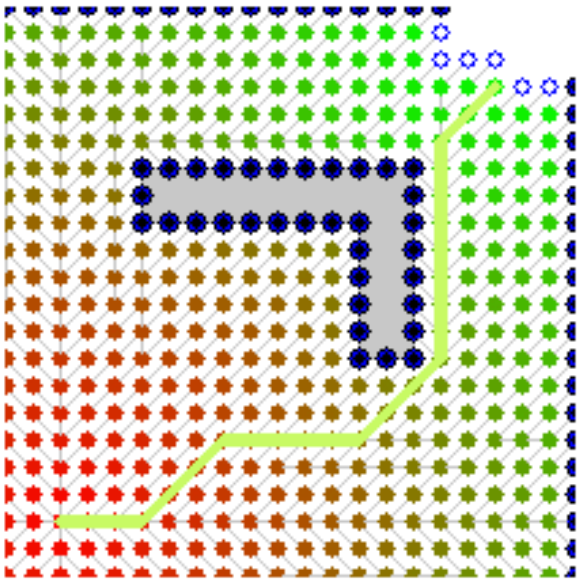


Figure 2.3: Dijkstra Process

and other nodes take infinite cost, which is the tentative cost value. For every step from the previous node and further node, cost calculation can be done, and unvisited nodes become visited with new cost value. According to the minimum cost value of nodes, the path makes relaxation and chooses the optimal one. The graph continuously expands for all the directions uniformly and searches for minimum cost. Because of it, the Dijkstra algorithm can be very slow and memory expensive in complex or relatively large environments [6]. On the other hand, Dijkstra does not use the heuristic function. Because of it, always returns an optimal path.

A* Algorithm and Weighted A* Algorithm

Unlike Dijkstra, A* uses heuristic function. Furthermore, Paden et al. mentioned that A* is the most popular heuristic search algorithm [37]. A* developed by Hart, Nilsson, and Raphael in 1968 as a heuristic optimal (minimum cost) solution for path finding problems with improved computational efficiency [19].

The working logic of A* is quite similar to Dijkstra. The only difference is A* has a heuristic function. Hart et al. defined the algorithm in [19].

Algorithm 1: Dijkstra algorithm

```
function Dijkstra(Graph, source):
create vertex set Q ;
for each vertex v in Graph do
    dist[v] ← INFINITY ;
    prev[v] ← UNDEFINED ;
    add v to Q ;
end
dist[source] ← 0 ;
while Q is not empty do
    u ← vertex in Q with min dist[u] ;
    remove u from Q ;
    for each each neighbor v of u do
        alt ← dist[v] + length(u, v) ;
        if alt < dist[v] then
            dist[v] ← alt ;
            prev[v] ← u ;
        end
    end
end
return dist[], prev[];
```

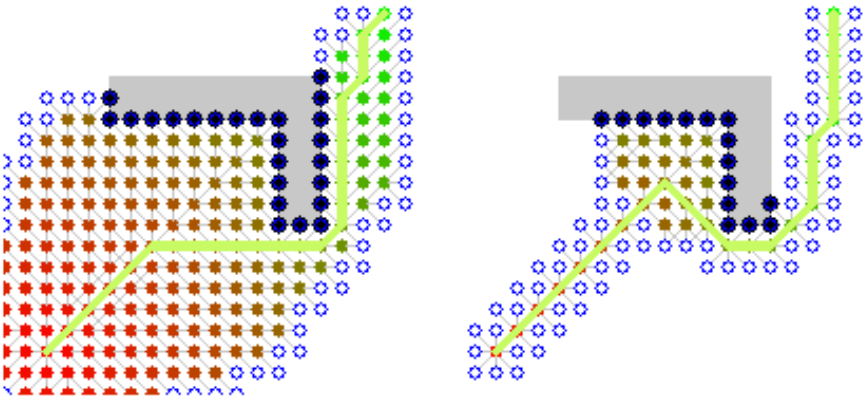


Figure 2.4: A and Weighted A* Process*

http://sbpl.net/node/50

First, sign state s (initial state) as open state and calculate the evaluation function f(s). Then the open node n with the smallest f function is selected. After selecting this node, it will check for other nodes that can have the smallest f function. If it finds another smallest node, relaxation can resolve ties and complete the path with the smallest nodes. When it reaches the goal point with the least cost function, the algorithm stops and gives the final path. At this point, thanks to the heuristic function, A* finds a path faster than Dijkstra, but we cannot directly say A* always finds optimal paths. If the heuristic function admissible, then A* finds the optimal path. Otherwise, it can find fast but sub-optimal paths.

Algorithm 2: A* algorithm [14]

```
ComputeShortestPath() {
while argmin_{s∈OPEN}(g(s) + h(s)) ≠ 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') = g(s) + h(s)
        end
    end
end
}
Main() {
for all s ∈ S do
    g(s) = ∞
end
g(s_start = 0)
OPEN = ∅
insert s_start into OPEN with value g(s_start) + h(s_start, s_goal)
ComputeShortestPath()
}
```

Although A* faster than Dijkstra and optimal search algorithm, some applications do not need optimality. Moreover, A* can take some time to solve and find an optimal solution for difficult search problems. Rather than optimality, one can look for faster but sub-optimal solutions in some search problems [18]. At this point, we can use the Weighted A* algorithm. The only difference from A* is in the evaluation function includes a weight for the heuristic function. This weight can be defined by the designer, and it

can change according to applications. When one gives to weight value as one for Weighted A* algorithm, it returns Weighted A* algorithm to A* algorithm. This weight makes path finding solutions more greedy and faster but sub-optimal.

$$f = g(n) + \epsilon h(n) \tag{2.1}$$

$$\epsilon >= 1 \tag{2.2}$$

ARA* Algorithm

In the real world, it is not always desirable to find optimal routes in too much time. On the other hand, finding an optimal path cannot be feasible for every search according to the desired time. Because of it sometimes, for some applications, people can look at suboptimal but faster algorithms. Then, it can be expected these algorithms to improve themselves for optimality.

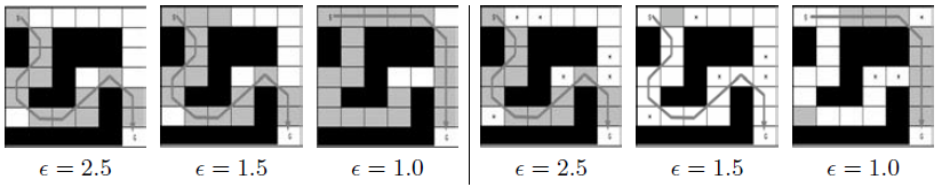


Figure 2.5: Left three columns: A searches with decreasing ϵ. Right three columns: the corresponding ARA* search iterations. [30]*

Weighted A* is fast but, it is not optimal. Moreover, its paths can be bad if one gave high weight for heuristic value. Because of this problem, Likhachev et al. proposed Anytime Repairing A* (ARA*) algorithm [30]. ARA* algorithm uses the basics of the weighted A* algorithm. The user is defining a weight for heuristic value, and the algorithm is starting with this value. If this value high, the algorithm finds a very fast but suboptimal path. Then, according to the given time, it decreases weight until it will be equal to 1. That meant to algorithm updates paths and finds an optimal path. Also, Ferguson et al. say, anytime algorithms generate suboptimal paths very fast, then they improve the solution quality given time [14].

Algorithm 3: ARA* algorithm [14]

```
key(s) {
return g(s) + ϵh(s_start, s);
}
ImprovePath() {
while min_{s∈OPEN}(key(s)) < key(s_start) do
    remove s with the smallest key(s) from OPEN;
     CLOSED = CLOSED ∪ s;
    for for all s'Pred(s) do
        if s' was not visited before then
            g(s') = ∞
        if g(s') > c(s', s) + g(s) then
            g(s') = c(s', s) + g(s)
            if s' ∉ CLOSED then
                insert s' into OPEN with key
            else
                insert s' into INCONS
            end
        end
    end
end
}
Main() {
g(s_start) = ∞ ; g(s_goal) = 0 ;
ϵ = ϵ_0;
OPEN = CLOSED = INCONS = ∅ ;
insert s_goal into OPEN with key(s_goal);
ImprovePath();
publish current ϵ − suboptimal solution;
while ϵ > 1 do
    decrease ϵ;
    Move states from INCONS into OPEN;
    Update the priorities for all sϵOPEN according to key(s);
    CLOSED = ∅ ;
    ImprovePath();
    publish current ϵ − suboptimal solution;
end
}
```

AD* Algorithm

Previously we talked about static algorithms. These algorithms chiefly work in a static environment. They ignore changes inside the environment, and they make a plan according to the global map. However, in real life cannot be seen static environments. There is various type of moving objects that can be human, cars, animals and so on. Moreover, Ferguson mentioned dynamic and complex problems are some of the most fascinating real-world problems [14]. In this situation, static planners will not work with dynamic environments.

Before explaining AD* in this part, we will briefly touch D* [45] and D* Lite [23]. D* was developed by Stentz to produce optimal paths through a graph with evolving or modified arc costs in real-time [45]. After developing D*, Koenig and Likhachev improved it and published D* Lite. This new algorithm similar to a perspective of navigation strategy but, it has some differences in the algorithm structure part. Reduced complexity and making the D* Lite algorithm shorter than D*, creates an easier and more efficient algorithm for complex or unknown environments.

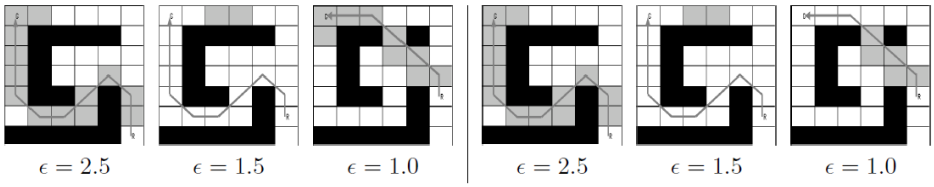


Figure 2.6: Left three columns: ARA searches with decreasing ϵ. Right three columns: the corresponding AD* search iterations. [14]*

For one step ahead from D* and D* Lite, we can see AD*. For real life, dynamic environments Likhachev, et al. developed the Anytime Dynamic A* (AD*) algorithm that a combination of ARA* and D* Lite [28] [29]. This combination causes fast solutions for complex planning problems from ARA* also, thanks to D* Lite, not affected by changes from dynamical environments. Furthermore, Likhachev et al. explain the algorithm "It performs a series of searches using decreasing inflation factors to generate a series of solutions with improved bounds, as with ARA*. When there are changes in the environment affecting the cost of edges in the graph, 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 with D* Lite. States on the queue are then processed until the current solution is guaranteed to be e-suboptimal." [29].

In Algorithm 4 and 5 represent a pseudo-code of AD*. In the main loop, it starts by giving initial values. Here the important part is defining the ϵ_0 value. If the initial weight is chosen adequately high value, the algorithm can find a suboptimal path very fast. After that, the algorithm checks for changes inside the environment. If it realizes some small changes in the environment, it can act like D* and updates the edges. On the other hand, if it realizes some major changes, it can replan from the beginning or just increase the weight and look for another solution. Otherwise, there are no changes in the environment, it behaves like the ARA* algorithm, and it searches for an optimal path decreasing the weight.

2.3.2 Sampling-Based Planning

Sampling-based motion planning is a powerful idea that uses a sampling of the robot's state space to respond to planning queries quickly and efficiently, particularly for systems with differential constraints or multiple degrees of freedom [7]. Because of environment size and motion constraints, solving motion planning problems can take a long time with the traditional approach. Using a proper sampling-based search method can decrease this time.

Sampling-based planning approaches are created to deal with high dimensional state spaces, and they soon proved to be much more practical. These approaches rely on random sampling and checking samples and their connections using collision detection algorithms. Then, it incrementally searches the configuration space for a solution instead of completely characterizing all of the collision-free space. The main idea of sampling-based planning is to sample states in the state-space and connect these states with trajectories in the collision-free space.

The sampling-based motion planning method has some problems like repeatability. Because of random samples, each run can return various paths. By luck, it can find short paths, sometimes long paths. Although making a plan with sampled base is not optimal for a wide-area, it can find paths rapidly using less computational resources.

Since sampling-based algorithms usually do not sample the whole space, so they use less computational resources. Sometimes less sampling can cause missing the goal. They cannot determine if a solution exists or not for a

Algorithm 4: AD* algorithm ComputeorImprovePath function [14]

key(s) {
if $(g(s) > rhs(s))$ **then**
  return $[\min(g(s), rhs(s)) + \epsilon h(sstart, s); \min(g(s), rhs(s)))]$;
else
  return $[\min(g(s), rhs(s)) + h(sstart, s); \min(g(s), rhs(s)))]$;
end
}
UpdateState(s) {
if *s was not visited before* **then**
  $g(s) = \infty$;
if $s \neq s_{goal}$ **then**
  $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
if $s \in OPEN$ **then**
  remove s from OPEN;
if $g(s) \neq rhs(s)$ **then**
  if $s \notin CLOSED$ **then**
    insert s into OPEN with key(s);
  end
else
  insert s into INCONS;
end
}
ComputeorImprovePath() {
while $min_{s \in OPEN}(key(s)) < key(s_{start})$ *OR* $rhs(s_{start} \neq g(s_{start})$
 do
  remove s with the smallest key(s) from OPEN;
  if $g(s') > rhs(s)$ **then**
    $g(s) = rhs(s)$
    $CLOSED = CLOSED \cup s$;
    for all $s' \in Pred(s)$ UpdateState(s');
  else
    $g(s) = \infty$;
    for all $s' \in Pred(s) \cup s$ UpdateState(s');
  end
end
}

Algorithm 5: AD* algorithm main function function [14]

```
Main() {
g(s_start) = rhs(s_star) = ∞ ; g(s_goal) = ∞ ;
rhs(s_goal) = 0; ε = ε_0;
OPEN = CLOSED = INCONS = ∅ ;
insert s_goal into OPEN with key(s_goal);
ComputeorImprovePath();
publish current ε − suboptimal solution;
while true do
    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);
        end
    if significant edge cost changes were observed then
        increase ε or replan from scratch;
    if ε > 1 then
        decrease ε;
    end
    Move states from INCONS into OPEN;
    Update the priorities for all s ∈ OPEN according to key(s);
    CLOSED = ∅ ;
    ComputeorImprovePath();
    publish current ε − suboptimal solution;
    if ε = 1 then
        wait for changes in edge costs;
    end
end
}
```

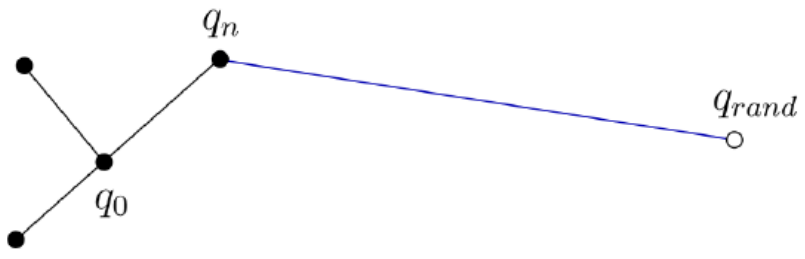


Figure 2.7: Adding a new edge that connects from the random sample q_{rand} to the nearest point in the tree, which is the vertex q_n.

given bounded time. But in an infinitely long time, it can find an optimal path from the initial to the final point. On the other hand, Karaman and Frazzoli proved to asymptotical optimality of sample-based algorithms [20].

Probabilistic RoadMap (PRM)

PRM algorithm is a sample-based multi-query algorithm for static environments. Kavraki et al. presented PRM algorithm in [21].

PRM include two main phases [21]:

- **Learning Phase:** The learning phase is a preprocessing stage in which a probabilistic roadmap is created by attempting connections between N randomly sampled configurations in a collision-free space. Its edges correspond to feasible paths in between randomly sampled configurations. It is time bounded phase, and the designer can define time according to the complexity of the environment. The learning phase consists of two significant steps. One of them is the construction step the other one is the expansion step. For the initially empty graph, the construction step connects random configurations if they are inside the free configuration space. On the other hand, the expansion step improves the connectivity of the graph generated by the construction step.

- **Query Phase:** The query phase is the path connection phase in between initial and final positions, which are selected from the roadmap.

In Algorithm[2] 6, we proposed how PRM work. Moreover, Figure[3] 2.8 represent main steps of algorithm. Very shortly, it randomly samples in the empty space then chooses samples from collusion free space. After that, it starts to connect points with each others making collision check and obtain connected graphs. Inside this graph, it finds a path with query phase from given initial point to final point.

Karaman and Frazzoli showed to asymptotical optimality of The Optimal Probabilistic RoadMaps (PRM*) [20]. In the optimal version, the radius of the connections between the roadmap nodes is chosen as a function of the number of samples n. When the samples increase, the radius of the connections decreases. As a result of these choices, it goes to asymptotic optimality.

[2] http://bascetta.deib.polimi.it/images/b/be/CMR_AUT-Lect6.pdf

[3] http://motion.cs.illinois.edu/RoboticSystems/figures/planning/prm.svg

Algorithm 6: PRM algorithm

$V \leftarrow \emptyset$
$E \leftarrow \emptyset$
for $i = 0, 1, ..., N$ **do**
  $q_{rand} \leftarrow SampleFree_i$
  $U \leftarrow Near(G, q_{rand}, r)$
  $V \leftarrow V \cup (q_{rand})$
  for $u \in U$ *in order of increasing* $\|u - q_{rand}\|$ **do**
    if q_{rand} *and* u *are not in the same connected component of* G **then**
      if $CollusionFree(q_{rand}, u)$ **then**
        $E \leftarrow E \cup (q_{rand}, u)$
      end
    end
  end
end
return $G = (V, E)$

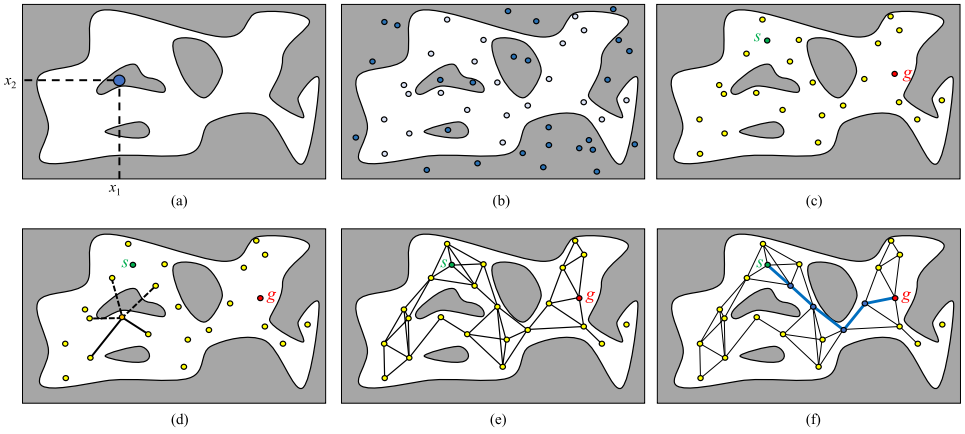
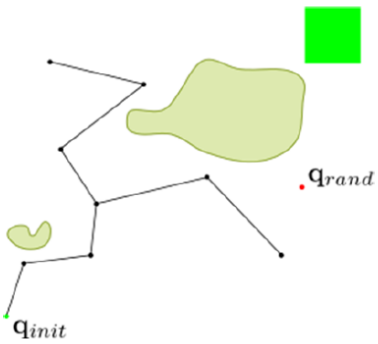
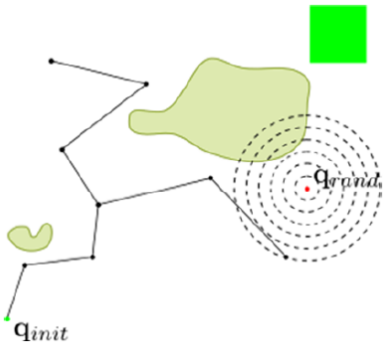


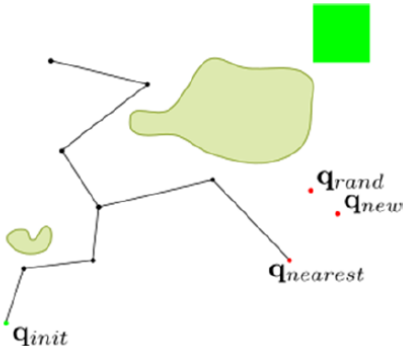
Figure 2.8: The main steps of the PRM algorithm.



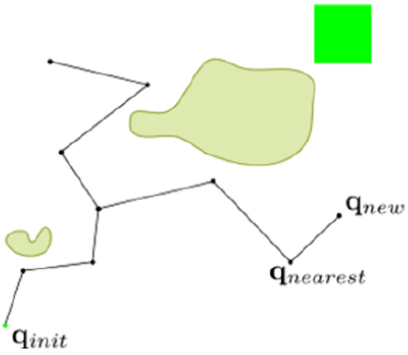
(a) Putting a random node to configuration space



(b) Searching for the nearest node into the tree



(c) Finding new node using steering function



(d) Connecting new node with tree

Figure 2.9: The main steps of the RRT algorithm

Rapidly Exploring Random Trees (RRT)

Rapidly Exploring Random Trees (RRT) is a single query sample-based algorithm. RRT was introduced by LaValle and Kuffner [25] in 2001 and it became one of the first popular sampling-based planners [7].

RRT proceeds in the form of an expanding tree, from the starting point to the end point (in Algorithm[4] 7). RRT incrementally builds a tree starting from an initial node, and inside a given circle, it puts random nodes to enlarge itself. While enlarging, RRT connects nodes and makes a tree structure. In this tree, each edge is a collision-free path between two nodes. Furthermore, in each iteration, the RRT algorithm picks a random node in

[4] `http://bascetta.deib.polimi.it/images/b/be/CMR_AUT-Lect6.pdf`

configuration space and then tries to connect the tree to it by extending the nearest node in the tree. At this point, finding the nearest node function has a significant effect on the performance of the RRT. The tree towards aggressively reaching unexplored parts of configuration space to find the goal point. Figure 2.9 shows the main steps of the PRM algorithm.

Algorithm 7: RRT algorithm

$V \leftarrow \{q_{init}\}$
$E \leftarrow \emptyset$
for $i = 0, 1, ..., N$ **do**
  $q_{rand} \leftarrow SampleFree_i$
  $q_{nearest} \leftarrow Nearest(G, q_{rand})$
  $q_{new} \leftarrow Steer(q_{nearest}, q_{rand})$
  if $CollusionFree(q_{rand}, q_{new})$ **then**
    $V \leftarrow V \cup (q_{new})$
    $E \leftarrow E \cup (q_{nearest}, q_{new})$
  end
end
return $G = (V, E)$

The optimal version of RRT is called RRT* (in Figure 2.11). Karaman and Frazzoli introduced RRT*, making a remarkable improvement over RRT in terms of the quality of the solution [20]. Unlike RRT, RRT* includes two new functions. These are the near neighbor search and the rewiring operation. The near neighbor search tries to find the best parent among the nodes inside a certain search radius. It searches for all the near nodes of the tree and checks the cost for each of them. If it finds a better path that has less cost, then rewire function remove edge between nodes and connect less cost node (in Figure 2.10). These two features make RRT* asymptotically stable. According to Karaman and Frazzoli, if there is a solution, the probability of finding an optimal solution, converges to 1 as the tree grows to infinity [20].

Kuffner and LaValle presented the RRT-Connect algorithm as a brand new method to make collusion free path planning [24]. RRT-Connect is created for path planning problems that do not have differential constraints. RRT-Connect differs from RRT in two ways. The first one is, RRT-Connect includes a Connect heuristic which makes the algorithm greedy and searches with longer distances. The second one is, unlike RRT, the growth of trees

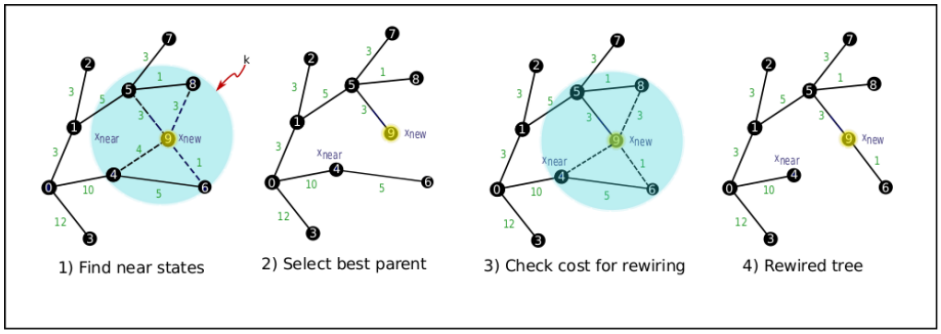


*Figure 2.10: Tree Generation in RRT**

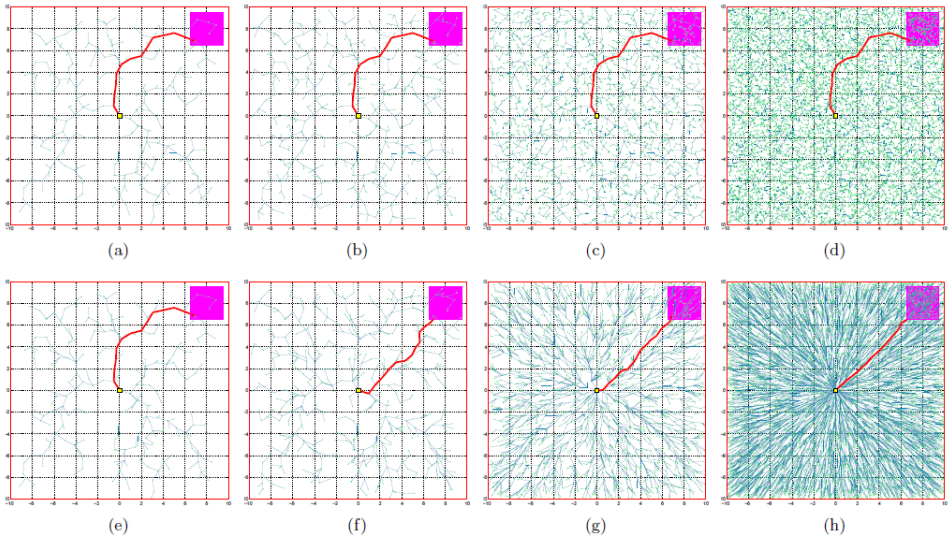


Figure 2.11: The comparison of RRT(first row) vs RRT (second row) algorithms with no obstacles. Remark that only edges differ. [20]*

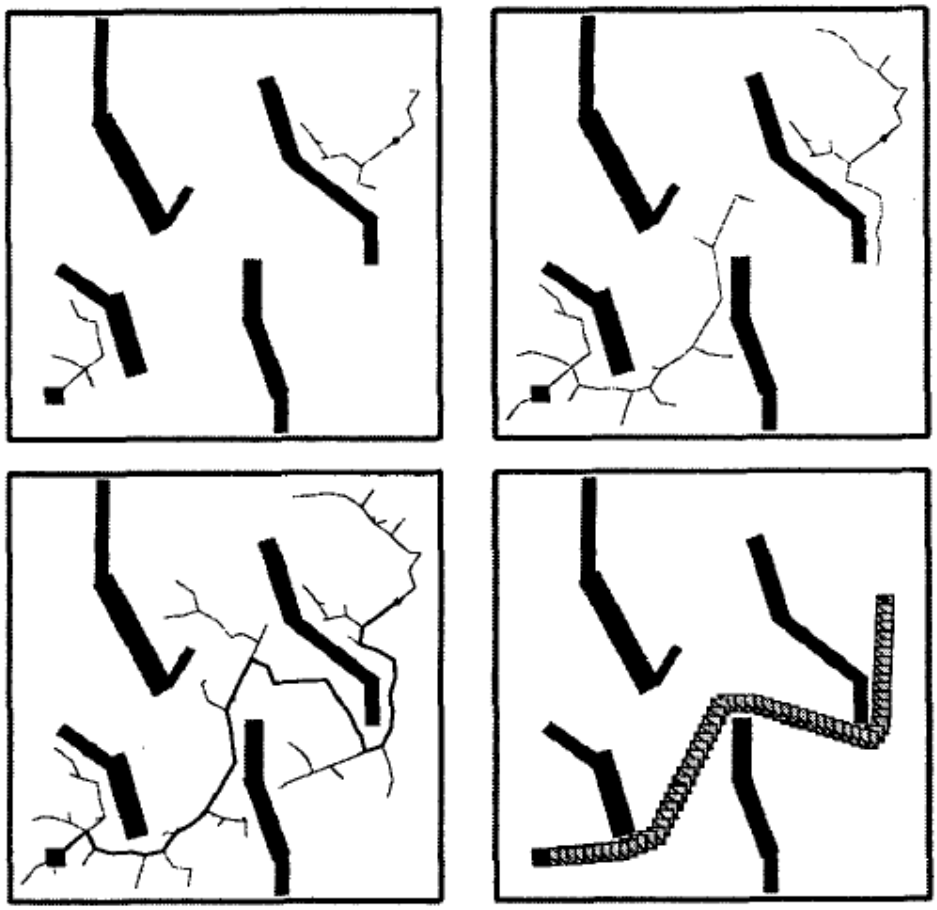


Figure 2.12: Growing two trees towards each other [24].

starts from both ways (from initial and end nodes), and RRT-Connect makes two trees towards each other. When they connect, RRT-Connect finds a solution inside this tree. As noted by Kuffner and LaValle, RRT-Connect has much better performance, and it has been seen as very successful in experiences [24].

Chapter 3

Reference Vehicles

In this work, we used two different vehicles. One of them is called a differential drive vehicle other one is called an Ackerman steering vehicle. For differential drive vehicles, we used the unicycle model. Moreover, for the Ackerman vehicle, we used the bicycle model. For these vehicles, we selected two reference robots. We chose Turtlebot3 [5] as a reference of the unicycle model and Agilex Hunter 2.0 [2] as the reference for bicycle model.

3.1 Differential Drive Vehicle (Turtlebot3)

The differential wheeled robot is a mobile robot whose travel is centered on two separate wheels mounted on each side of the robot body. Thus, by varying the relative rate of rotation of its wheels, it can adjust its direction, and that does not require additional steering motion. Vehicle wheels are actuated by two independent actuators. Because of the effect of these two independent rotational velocities, the vehicle can move at any point. In Equation 3.1, velocities of the vehicle can be defined. When we explained robot movement, we can say that while it is going forward, both wheels must be driven forward at the same speed. If wheels are driven in different directions at the same speed vehicle move on the spot, which is a wheelbase point of the vehicle. Moreover, while wheels are driven in the same direction but at different speeds, the vehicle moves on a point lying on the wheel axis, which is called the Instantaneous Centre of Curvature (ICC).

- x: Cartesian position of the vehicle along the x-axis
- y: Cartesian position of the vehicle along the y-axis
- θ: Orientation of the vehicle

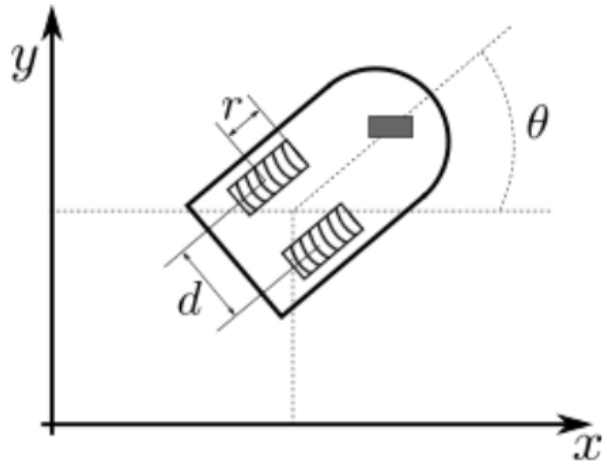


Figure 3.1: Differential drive model representation.

- v: Linear velocity of the vehicle
- ω_r: Rotational velocity for right wheel
- ω_l: Rotational velocity for left wheel
- ω: Angular velocity of the vehicle according to ICC

$$\begin{aligned} \dot{x} &= \frac{\omega_r + \omega_l}{2} r cos(\theta(t)) \\ \dot{y} &= \frac{\omega_r + \omega_l}{2} r sin(\theta(t)) \\ \dot{\theta}(t) &= \frac{\omega_r - \omega_l}{d} r \end{aligned} \tag{3.1}$$

We can model a differential drive vehicle robot as a unicycle vehicle model. A unicycle model can be defined as a vehicle with a single divertable wheel. Its configuration is described by the position of the wheel contact points (x and y) and the wheel orientation (theta). In Equation 3.2, differential drive model is defined, also in Figure 3.2 can be seen model of unicycle.

$$\begin{aligned} \dot{x} &= v(t) cos(\theta(t)) \\ \dot{y} &= v(t) sin(\theta(t)) \\ \dot{\theta}(t) &= \omega(t) \end{aligned} \tag{3.2}$$

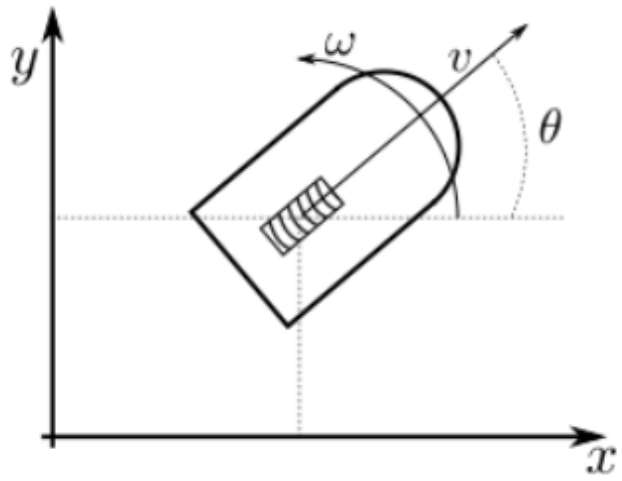


Figure 3.2: Unicycle model representation.

In this work, we used Turtlebot3 as a reference model robot, and we used its specifications in our simulations. Turtlebot3 is a small, compact, accessible, programmable, mobile robot based on ROS for use in educational, experiments, hobbies, and prototyping of goods and it aims to reduce the scale of the platform significantly and lower the price without losing its features and efficiency while providing expandability at the same time [5].

Turtlebot3 (in Figure 3.3) is well connected with ROS and packages of ROS [31]. ROS packages can be easily used in simulation platforms. Because of it, benchmarking, running the experiments, and further improvements becomes effortless. Moreover, tests can be easily performed for running a real environment.

3.2 Ackerman Steering Vehicle (Agilex Hunter 2.0)

Ackerman steering vehicles can be defined inside a geometric system of links in the steering of a car-like vehicles. According to Simionescu, it was designed by German carriage builder Georg Lankensperger, and after that, Rudolph Ackermann took its patent from England for horse-drawn carriages [43]. But according to King-Hele, Erasmus Darwin is the first inventor of this vehicle [22].

Nowadays, cars have four wheels, two in front and two in the back, which are parallel to each other. In this model, inside wheel follows a closer curve than the outside wheels while cars are taking corners. To do this, the

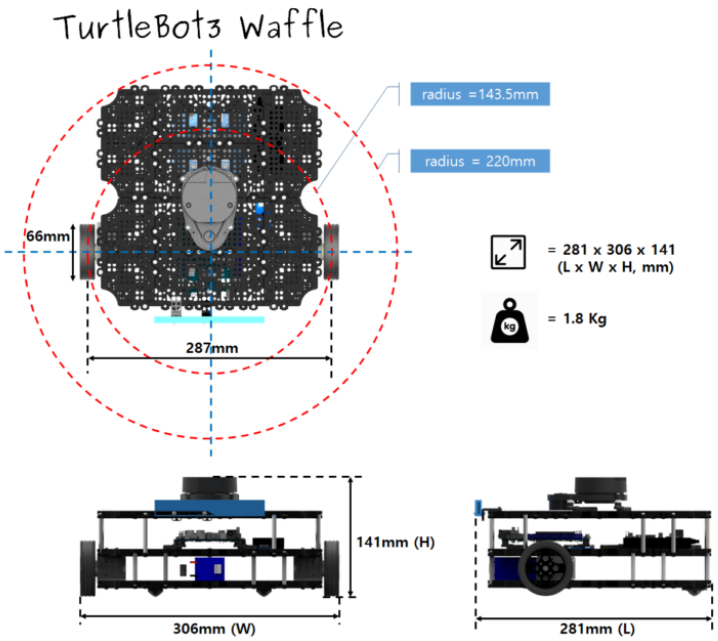


Figure 3.3: Specifications of Turtlebot3.

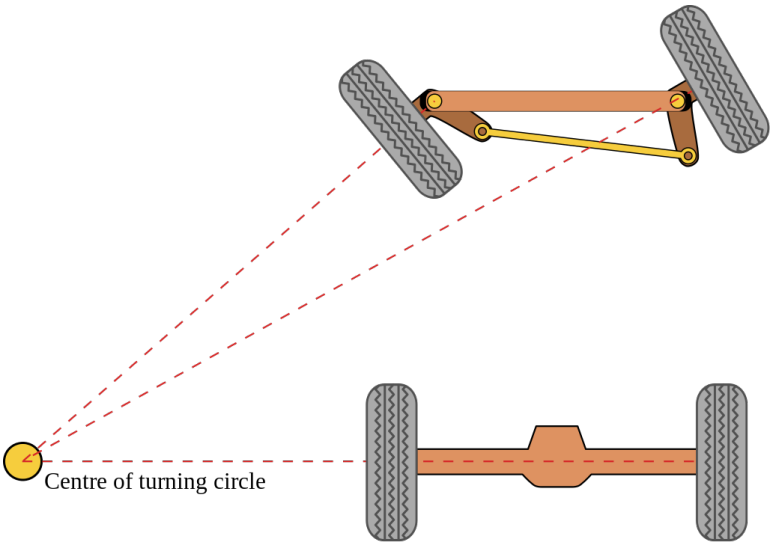


Figure 3.4: Ackerman steering geometry.

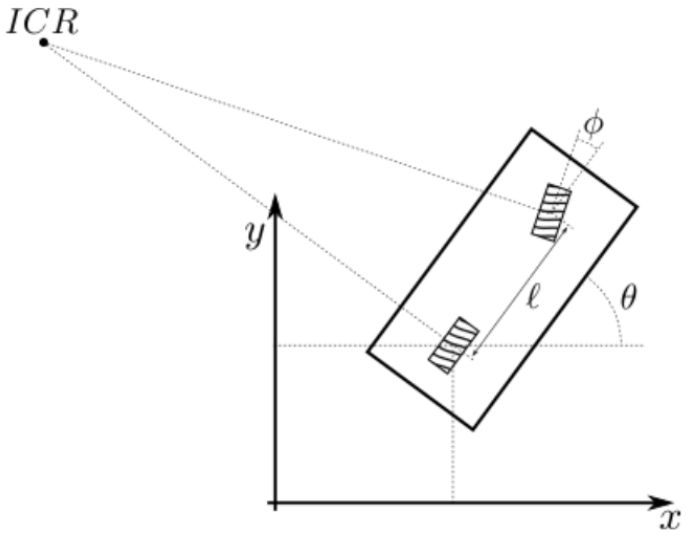


Figure 3.5: Bicycle model.

steering geometry must be arranged to rotate the inside wheel at a greater angle than the outside wheel, and Ackerman steering geometry (in Figure 3.4) offers a quick solution to this issue.

Ackerman steering vehicle can be simplified and modeled as a bicycle model in Figure 3.5. As known, the bicycle model has one wheel in front, which is orientable. Moreover, one wheel is fixed in the back. The vehicle can be configured by the positions of the rear wheel, the orientation of the vehicle, and the steering angle (x, y, θ, ϕ). When one give steer to front-wheel, vehicle move on the spot of Instantaneous Centre of Rotation. The kinematic model with three differential constraints of bicycle vehicle can be written as for rear-wheel speed in Equation 3.3, for front-wheel speed in Equation 3.4:

- x: Cartesian position of the vehicle along the x-axis
- y: Cartesian position of the vehicle along the y-axis
- θ: Orientation of the vehicle
- ϕ: Steering angle
- v: Linear velocity of the vehicle
- ℓ: Wheelbase

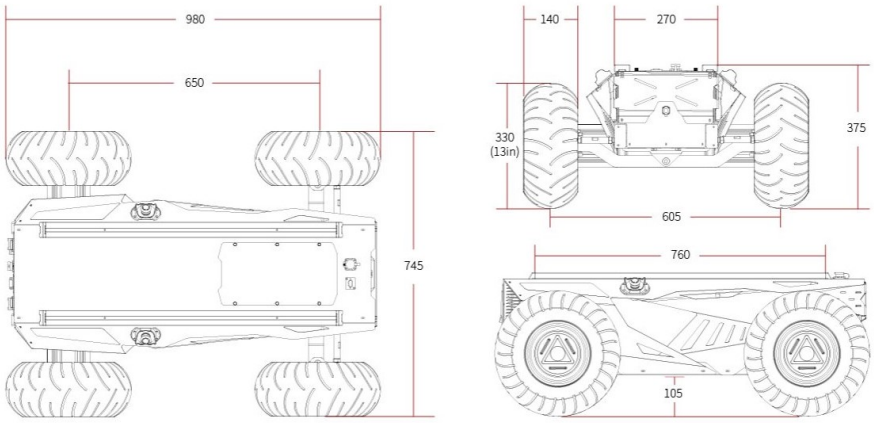


Figure 3.6: Specifications of Agilex Hunter 2.0.

$$\begin{aligned} \dot{x} &= v_r cos(\theta) \\ \dot{y} &= v_r sin(\theta) \\ \dot{\theta} &= \frac{v_r}{\ell} tan(\phi) \end{aligned} \tag{3.3}$$

$$\begin{aligned} \dot{x} &= v_f cos(\theta) cos(\phi) \\ \dot{y} &= v_f sin(\theta) cos(\phi) \\ \dot{\theta} &= \frac{v_f}{\ell} sin(\phi) \end{aligned} \tag{3.4}$$

In this work, we used Agilex Hunter 2.0 as a reference model robot, and we used its specifications in our simulations. This front wheel Ackerman steering robot is designed for low-speed driving scenarios [2]. Its specifications is given in Figure 3.6.

Specifications of Agilex Hunter 2.0:

- Dimensions: 980x745x380 mm
- Wheelbase: 650 mm
- Minimum turning radius: 1.6 m
- Maximum turning angle: 22°
- Drive form: Front-wheel Ackerman steering rear-wheel drive

Chapter 4

Voronoi Diagram and Environments

In this chapter, we give general information about the Voronoi graph. We answer how we generate the Voronoi graph in our environments. After that, we talk about benchmark environments as well as their specifications.

4.1 Voronoi Diagram

Voronoi Diagram is a region division problem in plane regions close to each of a given series of objects. Each region refers to one of the sites, with all points in one region being closer to the region's representative location than any other site [34]. Shortly, we can define the Voronoi diagram as the set of line segments separating the regions corresponding to the different points.

We can explain the Voronoi diagram with an example. In town, one wants to open a store, and s/he looks for a good location. If s/he selects very close to other stores, it can be a problem because the new store steals

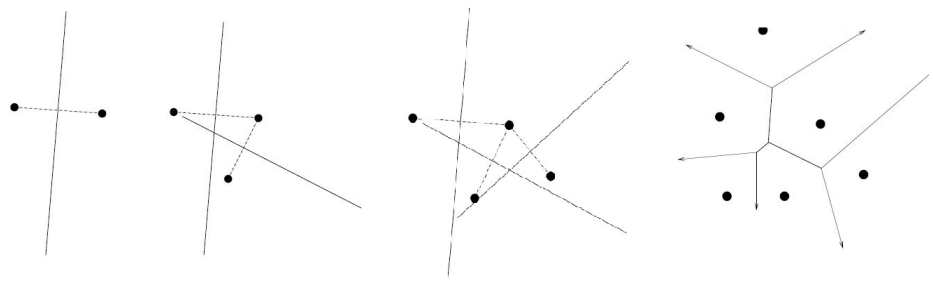


Figure 4.1: Voronoi diagram generation

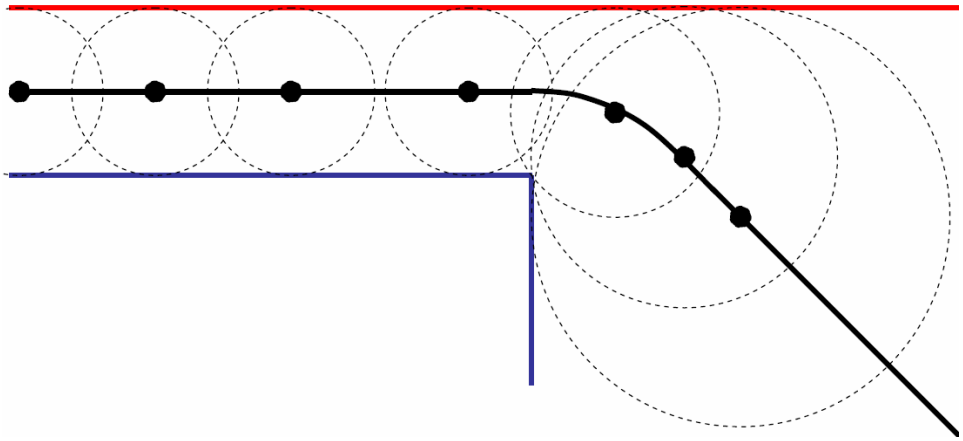
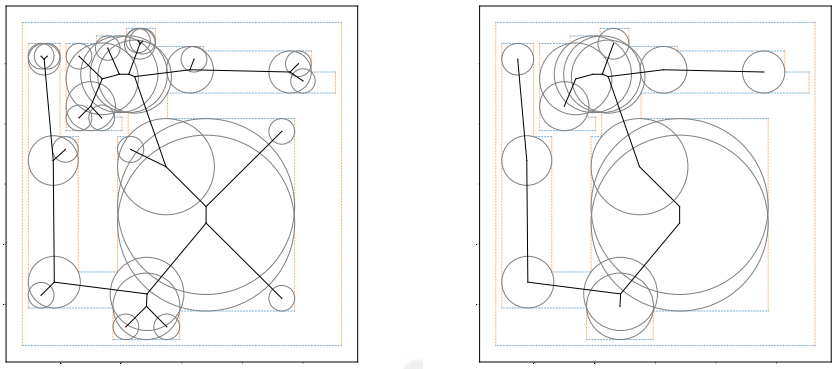


Figure 4.2: Voronoi graph generation with circle

other's customers. If s/he decides far from others, nobody goes to a distant shop. Using the Voronoi diagram, we can solve this problem. We can start with a collection of stores in a plane then, dividing the plane into regions, one region per store, where each region consists of those stores that are closer to the store of the region than to any others. Figure 4.1 shows step by step how the Voronoi diagram occurs.

According to Marbate and Jaini, Voronoi diagram has a wide usage area from geographical information system to robotics [32]. Furthermore, Voronoi diagram is used for path planning [15] [16] [32] [34]. But in our project, we use the Voronoi diagram to define the start and the final positions of the robot from the map. As knows, our obstacles are large objects in the real environment. Starting from a circle, that corresponds to robot radius with safety factor. Increasing robot radius, we can define the Voronoi graph (network) in between obstacles. In Figure 4.2 can be seen Voronoi circles and we use these circles center points as an initial and goal points in our environments. In Figure 4.3, we show the Airlab environment with Voronoi circles. For visualization purposes, we does not put real Voronoi circles because it is covering all maps. Thus, we reduce Voronoi graph like in 4.3(a) to see how it looks like. After that, we reduce it more and choose our initial and goal points from this map. In Figure 4.4, we can see red dots as initial points and green lines as a Voronoi graph that we use this graph distance for normalization of our metrics.



(a) Voronoi graph reduced version to clear (b) Voronoi graph more reduced verison to select points

Figure 4.3: Voronoi graph representation for Airlab

Figure 4.4: Voronoi graph for Airlab

4.2 Environments

In this section, we briefly talk about environments. For benchmarking, we select six different environments. Three of them are made of real data, and others are made of drawing programs. Moreover, we use various kinds of environments like a multi-room open space office or small hallway environment and so on.

4.2.1 Airlab

The Airlab environment, shown in Figure 4.5, is a handmade environment. It has some small corridors on the upside and left side, but the middle area of the environment is quite open to make a maneuver. The total length of the environment is 11.7 meters on the x-axis and 11.2 meters on the y-axis. As seen, this environment is considerably small. We select Airlab to comprehend how the robot is finding a path in a very small environment.

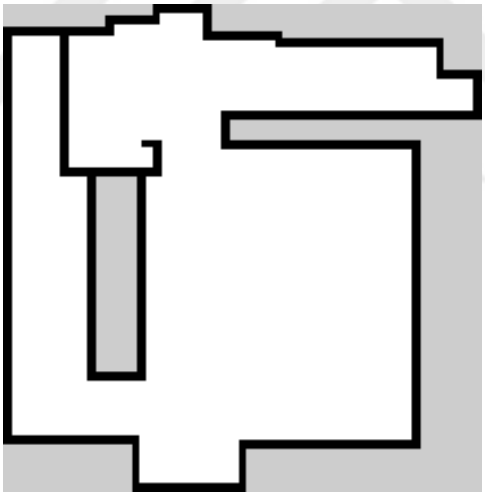


Figure 4.5: Airlab environment

4.2.2 7A-2

In Figure 4.6, 7A-2 environment can be seen. This environment appears as a very open space laboratory field. That includes quite wide hallways, halls, and corridors. On the other hand, it has some small rooms or offices, maybe toilets. But this environment has some tricky points that are small doors. The total area of the environment is 26.4x90.4 square meters. In Figure 4.6, it shows in horizontal position. In this environment, we investigate how the

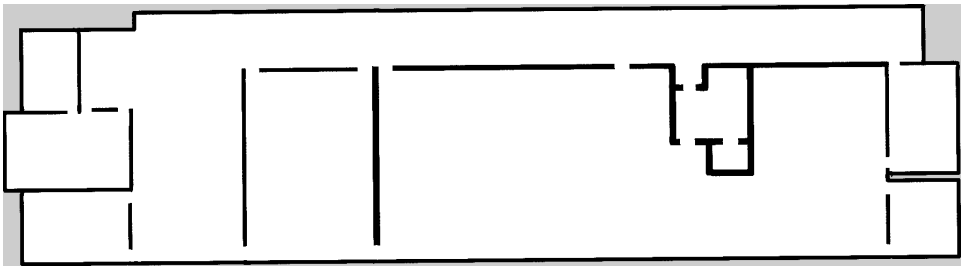


Figure 4.6: 7A-2 environment

robot moves in a wide-open environment and how small doors can affect the feasibility of finding a plan.

4.2.3 Office_b

Office_b environment (Figure 4.7) represents as an office. This environment consists of many small rooms and small corridors. Moreover, there are doors in between rooms. These specifications also make it a kind of real environment type of maze. The environment is also as big as 7A-2. The total width of the area is 60x24.5 square meters. Thanks to these specifications, we examine how robots act in a multi-room small corridor environment.

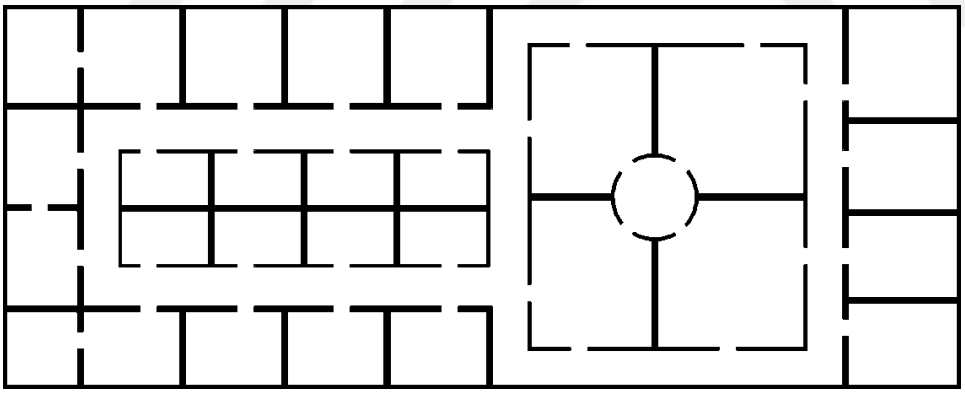


Figure 4.7: Office_b environment

4.2.4 Intel

Intel environment is a real data environment in Figure 4.8. It has different types of rooms, halls, and corridors. Furthermore, it has some obstacles inside the map. It is also a quite large environment. The total width of the area is 70x38 square meters. Intel environment shows how the robot finds a path in the real environment with a diversity of rooms and corridors.

Figure 4.8: Intel environment

4.2.5 Fr079

Fr079 environment can be seen in Figure 4.9. This environment is a real environment like Intel. It has one big corridor with small gates as well as it has varied rooms. Some rooms connected each other with small passages or unfeasible passages. Unfeasible passages mean there are some obstacles in between rooms. Thus, while making path planning, it can be a problem. Additionally, the total area of the Fr079 is 57.5x33.5 square meters. Here, we investigate how robots make motion planning inside these small rooms with many small obstacles although, the environment is significantly large.

4.2.6 Mexico

Mexico is a very wide-open environment. Its total area is 142x80 square meters. Figure 4.10 represents this real data environment. This environment is almost nine times bigger than office_b and five times bigger than 7A-2. Although it has too many free spaces for maneuver, there are some complex places. Some rooms are not available for robot also some rooms include obstacles that make difficult to plan. Using this environment, we observe that how the robot makes the plan for a quite big and complex environment and how its feasibility rate changes in this situation.

Figure 4.9: Fr079 environment

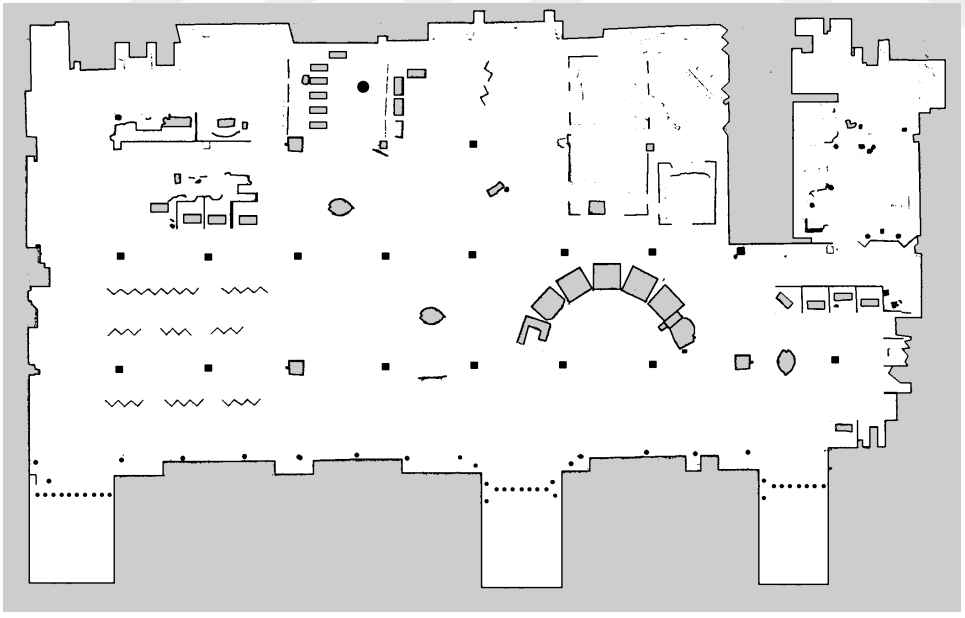


Figure 4.10: Mexico environment

Chapter 5

State Lattice and Motion Primitives

In this chapter, we elaborate on how to generate state lattice and how to make motion primitives according to differential drive and car-like vehicles for search-based motion planning algorithms.

5.1 State Lattice

The real world is viewed to tend to be in a continuous-time or space with high complexity. But, while we are making some applications in the robotic area, we need more easy and not complex solutions. To pass complexity and continuous problem, we must discretize time or space.

As mentioned in Section 2.3.1, search-based algorithms works with discrete environments. These discrete environments include states, and in between two states, there must be valid transitions. As a result of the discrete

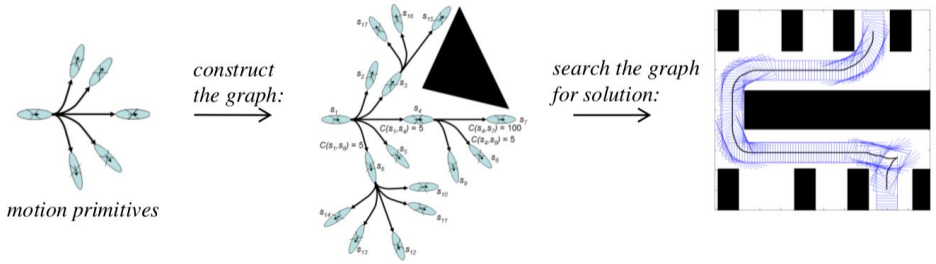


Figure 5.1: Path generation using with state lattice

source: http://sbpl.net/node/53

representation of states, the computational complexity of the algorithm is decreased. Moreover, Pivtoraiko and Kelly suggest a method for achieving the computational benefits of discretization when adhering to the motion constraints [38]. After that, they introduce State Lattice.

State lattice (in Figure 5.2) is a discretized sequence of all the system's reachable configurations. It is designed for nonholonomic motion planning tasks with the graph search method. State lattice is generated by discretizing the set of all reachable configurations that are in obstacle-free space according to resolution. Then using a search algorithm, it can find the feasible path from one point to another one with vehicle dynamics. Thus, one can see maneuvers of the vehicle according to kinetically feasible motion primitives.

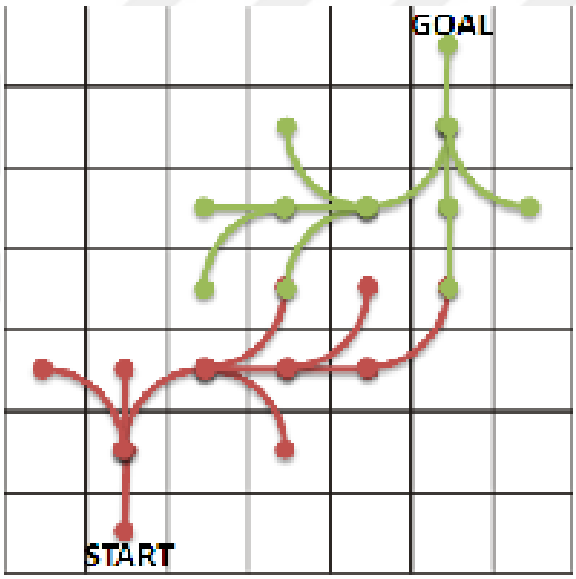


Figure 5.2: State lattice with motion primitives from start to goal

The state lattice is created by finding paths between any node in the grid and the origin using the inverse path generator. According to Likhachev and Ferguson, the discretization (or sampling) technique used to represent the states in the lattice, as well as the action space used for inter-state relations, are the two most important factors to consider when building a lattice [27].

State Space: Discretization of the environment according to resolution $\triangle x$ and $\triangle y$. In our work, we use a three dimensional (x, y, θ) state representation, where (x, y) represent the position of the wheel base of the vehicle in the environment. θ represents the orientation of the vehicle. The (x, y, θ) coordinates are important for computing the

validity of the poses of the vehicle in the environment.

Action Space: Presents the movement in the lattice. After discretization, motion plans need feasible motion paths from one (x_1, y_1) to (x_2, y_2) point in lattice. Thus, action space must define appropriate movements for each state in the lattice. In Equation 5.1, it is represented a feasible movement. Moreover, action space is constricting the motion primitives. In Figure 5.1, this constriction process can be seen in the middle by pointing how it is making obstacle avoidance.

$$\begin{pmatrix} x_1 + \triangle x_1 k \\ y_1 + \triangle y_1 k \\ \theta_1 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 + \triangle x_2 k \\ y_2 + \triangle y_2 k \\ \theta_2 \end{pmatrix} \forall k \in \mathbb{N} \tag{5.1}$$

SBPL provides a MatLab tool to build motion primitives for differential drive vehicles. On the other hand, we wrote our motion primitive generation code according to the kinematic constraints of the Ackerman steering vehicle. These motion primitives include short and kinematically feasible motion routes that can be performed by the robot.

5.2 Generating Motion Primitives for Differential Drive Vehicle

As explained before in this work we use (x, y, θ) environment in three dimensional space.

- x: Inside the state lattice, it gives the Cartesian x position of the robot. This value increases with resolution in the x-axis because the environment is sampled with resolution.
- y: Inside the state lattice, it gives the Cartesian y position of the robot. This value increases with resolution in the y-axis because the environment is sampled with resolution.
- θ: Gives the orientation of the robot. The positive side goes counterclockwise, and according to the lattice, it is sampled as well. We used $atan2(y, x)$ in MatLab for deciding θ values. $Atan2(y, x)$ 16 degree discretization can be seen in Figure 5.4. Thanks to this discretization, lines can stay in the lattice, so x and y can take values according to resolution.

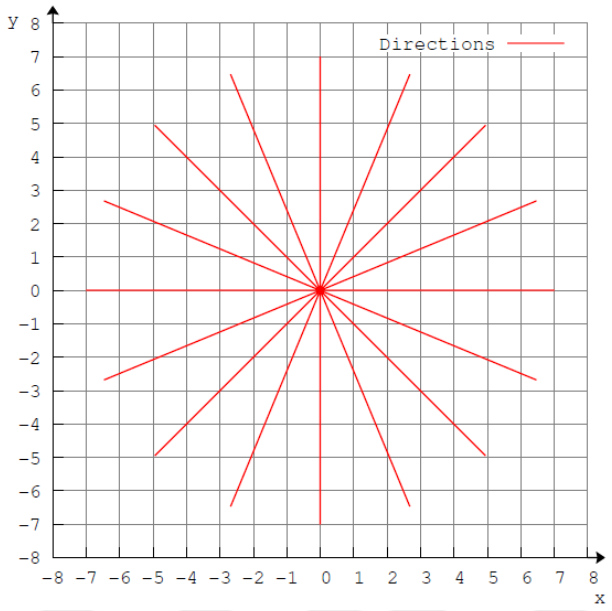


Figure 5.3: Uniform θ discretization

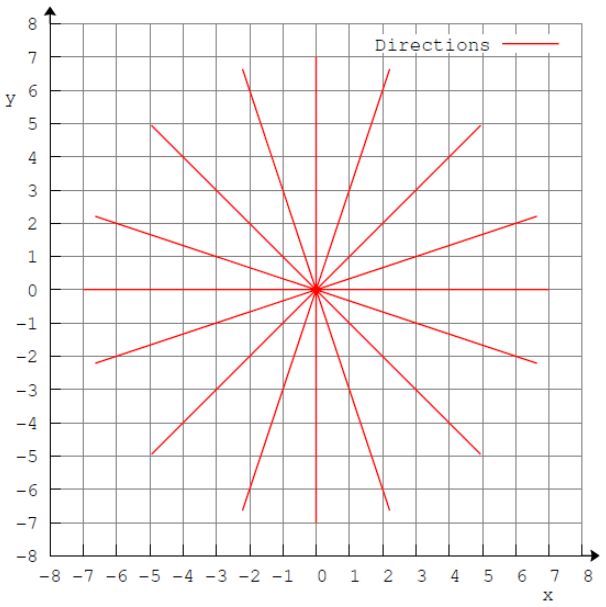


Figure 5.4: $atan2(\theta)$ discretization

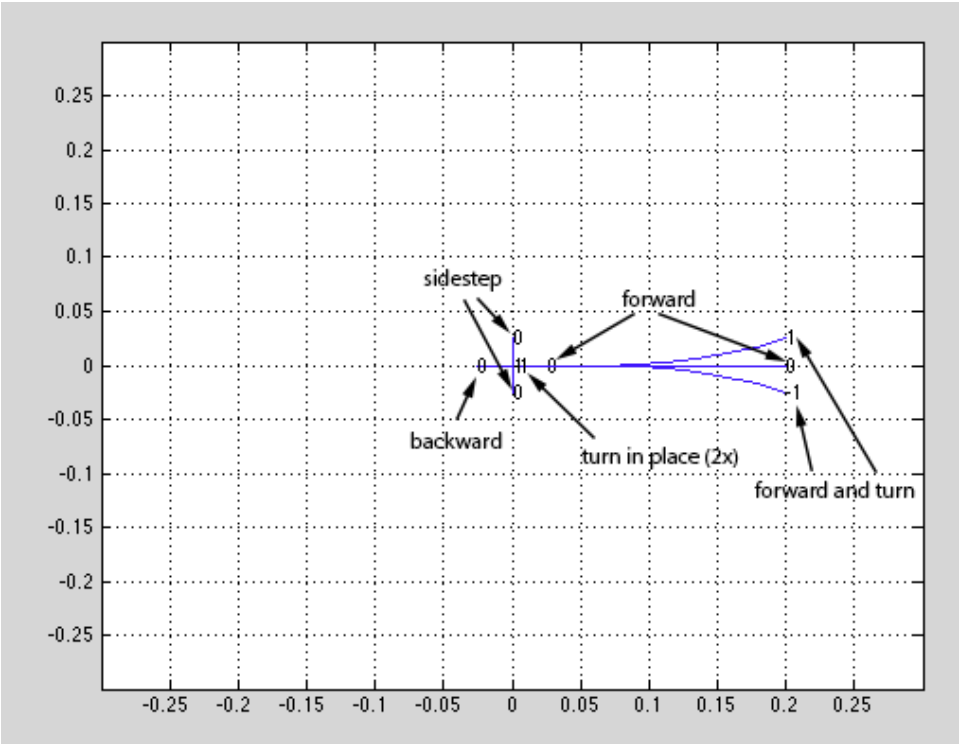


Figure 5.5: Unicycle motion primitives

Using MatLab primitive generation script, we generate primitives for unicycle vehicles. One can find this motion primitive generation code in the SBPL package. This script constructs forward, backward, turn in place, and forward turn arc motion primitives for a unicycle robot in the 3D state space (x, y, θ). The script generates primitives for a discrete set of 16 turning angles, in between $[0, 2\pi]$. The set of primitives generated for each turning angle can include one short and one long forward primitives, one short backward primitive, two turns in place primitives (clockwise and counterclockwise), and two forward-turn-arc primitives (clockwise and counterclockwise) in Figure[1] 5.5. All motion primitives start from the same starting point $(x = 0, y = 0)$, but they start different starting orientations according to 16 turning angle. Each generated motion primitives are defined by its type, its start pose, its end pose, its orientation in the 3D space (x, y, θ) also with cost multiplications.

Below section we give MatLab script description:

- resolution: Provides state lattice cell size of the (x, y) grid in meters.

[1] `http://sbpl.net/node/52`

- numberofangles: Provides number of discrete angles per circle. We defined 16 number of angels. This parameter determines the base turning angle as $\delta = f\frac{2\pi}{numberofangles}$. In our situation we defined numberofangles is equal to 16 so each angle is 22.5 degree. When we gave the integer number to angles it is start with 0 for 0 angle, 1 for 22.5, 2 for 45 and it goes like this to 15 for 342.5 degree.

- numberofprimsperangle: Provides what kind of movement the robot will make. If it is defined as 5, it includes one short and one long forward primitives, one short backward primitive, forward-turn-arc primitives (clockwise and counterclockwise). If it is defined as 7, it includes one short and one long forward primitives, one short backward primitive, two turns in place primitives (clockwise and counterclockwise), and two forward-turn-arc primitives (clockwise and counterclockwise). If one increases more, new primitives movements are added to the motion primitives sets.

- numofsamples: Provides number of discrete points on each primitive trajectory.

- Multipliers: Consist of different multipliers like forwardcostmult or backwardcostmult. It allows the user to define preferred actions. For example, increasing or decreasing the cost multiplier values, one can control which movement must be selected by the robot mostly.

- Motion Vectors: Predefined motion vectors according to lattice.

5.3 Generating Motion Primitives for Ackerman Drive Vehicle

In this work, besides the unicycle model vehicle, we used an Ackerman drive vehicle. In Section 3, we described and modeled the Ackerman drive vehicle as a bicycle model for simplification. In this section, we detailed how we made motion primitives according to Ackerman drive vehicle kinematics.

As known, car-like vehicles do not have sideway and turning place motions. This always causes some problems for car-like vehicles. To illustrate, car parking is an important problem for drivers. Drivers must make some maneuvers to park the car in a proper way. Because of these boundaries, car-like vehicles need feasible motion primitives according to their kinematic constraints. To generate appropriate motion primitives, we write a MatLab script with using the clothoid function.

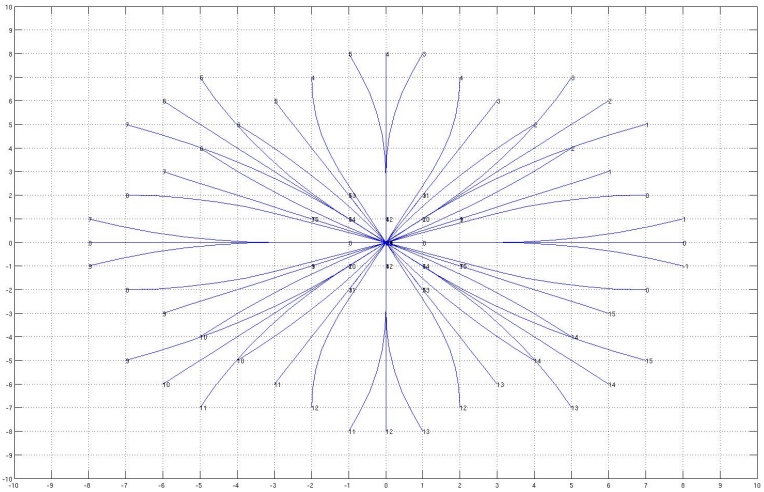


Figure 5.6: Unicycle motion primitives for 16 number of angles with 5 number of primitive per angle

source: http://sbpl.net/node/53

5.3.1 Clothoid Funciton

To build motion primitives according to kinematic constraints of the vehicle, we used the clothoid function in MatLab. Before talking about the script, we must explain some definitions. For general information, clothoid, also known as Euler spiral, is a curve, and it has a variety of work fields from optics to railroad engineering. Clothoid curvature changes linearly with its curve length. The curvature describes how much the curve direction changes over a small distance traveled for any part of a curve. Moreover, curvature gives some answers like how much unit vector changed in small arc-length and how much it is turning, type of questions. The curvature defined in Equation 5.2. If curvature changes linearly, the curve is called clothoid. If curve has constant curvature ($K' = 0$), it creates circle. If there is no curvature ($K' = K = 0$), then it is called straight line.

$$K(s) = K's + K \tag{5.2}$$

In Figure[2] 5.7, curvature and turning radius relationship is represented. Inside a given C curve in P point, the curvature can be calculated from given equation in 5.3.

[2] `https://en.wikipedia.org/wiki/Curvature`

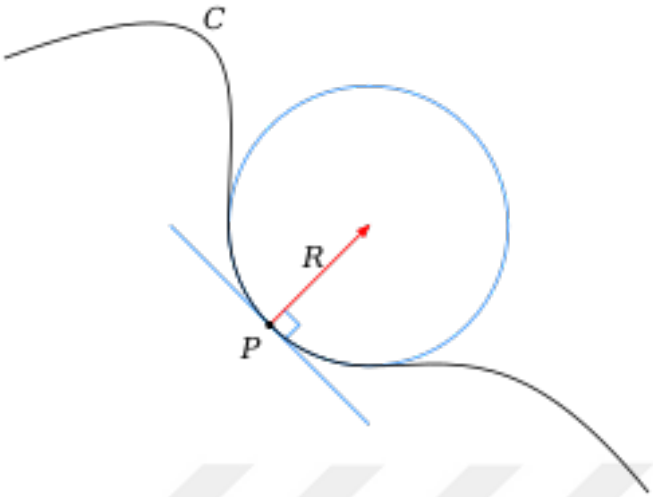


Figure 5.7: Curvature for radius R

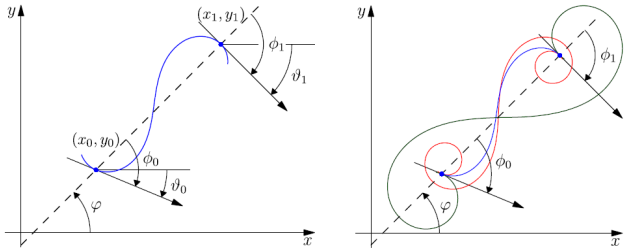


Figure 5.8: Hermite G^1 interpolation and its possible solutions [9]

$$K = \frac{1}{R} \tag{5.3}$$

Bertolazzi and Frego proposed a new method for computing G^1 Hermite interpolation numerically using a single clothoid segment [9]. Thus, they wrote G1fitting MatLab library as an open source library[3]. It is aimed to solve Hermite G^1 interpolation with a clothoid curve. In Figure 5.8, we can see Hermite G^1 interpolation fitting problem and its possible solutions. According to given two points in a plane with tangent directions some possible curves can be drawn.

Clothoid function are used three different functions from G1fitting library. Aim of these functions are to find a feasible clothoid curves for given start (x_s, y_s, θ_s) and final poses (x_f, y_f, θ_f) according to kinematic constraints. After finding curves, motion primitive generate script can use

[3] `https://github.com/ebertolazzi/G1fitting`

them. Clothoid function took start pose, final pose, middle point number, and minimum turning radius value. Start pose, final pose and middle point are used for G1fitting functions, minimum turning radius is used for checking for feasible paths. Finally, it returns clothoid curve points for motion primitive generation script.

Three used functions are defined below:

buildClothoid

buildClothoid function computes parameters of the G^1 Hermite clothoid fitting. It takes start and final poses and it returns L (the length of the clothoid curve from initial to final point), k (curvature at initial point), dk (derivative of curvature respect to arclength), and iter (Newton Iterations used to solve the interpolation problem).

For more detail about fitting problem, the curve satisfies the Equation 5.4. Thanks to Firesnel integral the solution of the ODEs (5.4) is given in Equation 5.5. For the given the two positions with their orientations, it can find curves that curves satisfies Equations 5.6.

- s: arc parameter of the curve
- θ: orientation of $(x'(s), y'(s))$
- $K(s)$: curvature at the point $(x(s), y(s))$

$$\begin{aligned} x(s') &= \cos\theta(s) \quad x(0) = x_0 \\ y(s') &= \sin\theta(s) \quad y(0) = y_0 \\ \theta(s') &= K \quad \theta(0) = \theta_0 \end{aligned} \tag{5.4}$$

$$\begin{aligned} x(s) &= x_0 + \int_0^s \cos\left(\frac{1}{2}K'\tau^2 + K\tau + \theta_0\right)dt = x_0 + sX_0(K's^2, Ks, \theta_0) \\ y(s) &= y_0 + \int_0^s \sin\left(\frac{1}{2}K'\tau^2 + K\tau + \theta_0\right)dt = y_0 + sY_0(K's^2, Ks, \theta_0) \end{aligned} \tag{5.5}$$

$$\begin{aligned} x(0) = x_0, \quad y(0) = y_0, \quad (x'(0), y'(0)) = (\cos\theta_0, \sin\theta_0) \\ x(L) = x_1, \quad y(L) = y_1, \quad (x'(L), y'(L)) = (\cos\theta_1, \sin\theta_1) \end{aligned} \tag{5.6}$$

The solution of interpolation by Equation 5.5 is a zero of the following nonlinear system (5.7) involving the unknowns L, K, K' [9]. Thus, buildClothoid solves this nonlinear equations using Newton-Raphson iterations

and finds the unknown coefficients of the clothoid curves. For more detail [9] and [10] explain all solution methodology.

$$\begin{aligned} x_1 - x_0 - LX_0(K'L^2, KL, \theta_0) &= 0 \\ y_1 - y_0 - LY_0(K'L^2, KL, \theta_0) &= 0 \\ \theta_1 - (\frac{1}{2}K'L^2 + KL + \theta_0) &= 0 \end{aligned} \tag{5.7}$$

pointsOnClothoid

This function provides finding clothoid curve points. It takes coordinate of an initial point, the orientation of the clothoid at the initial point, curvature at the initial point, the derivative of curvature concerning arc-length, the length of the clothoid curve or a vector of length were to compute the clothoid values and the number of points along the clothoid. As seen, some of these values are the return values of the buildClothoid function. Moreover, pointsOnClothoid returns a matrix that includes the X and Y coordinate of points of the clothoid. Using these points coordinate, we can find middle points of clothoid both visualization and mprim file write purpose.

evalClothoid

This function computes clothoid parameters along a clothoid curve. It takes the coordinate of the initial point, the orientation of the clothoid at the initial point, curvature at the initial point, the derivative of curvature with respect to arc-length, and the vector of the curvilinear coordinate where to compute clothoid. It returns to the x coordinate of the points of the clothoid at s coordinate, y coordinate of the points of the clothoid at s coordinate, angle of the clothoid at s coordinate, and curvature of the clothoid at s coordinate.

We used this function to check for the feasibility of the curves. As known, the Ackerman steering vehicle has a maximum steering angle that is a constraint for the vehicle. For a given car length L and maximum steering angle ϕ, we can obtain minimum turning radius $R = \frac{L}{\tan\phi}$. Also we know from Equation 5.3, curvature is defined as $K = \frac{1}{R}$. Putting minimum radius into the formulation, we can obtain feasible curvature for desired maximum steering angle. Thus, we can check all the clothoid curve points, if they are feasible for the desired curvature or not, so we can select completely appropriate curves for motion primitives.

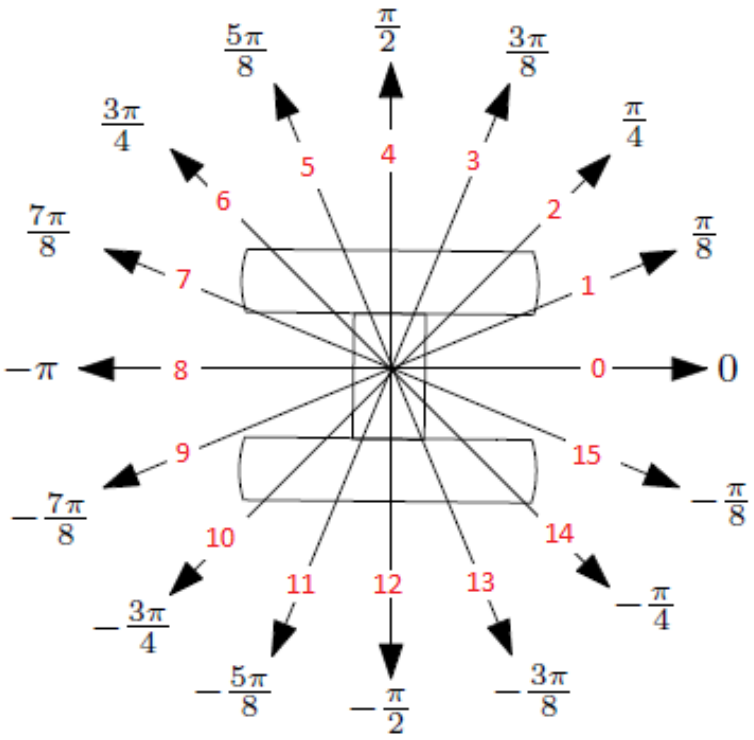


Figure 5.9: 16 number of angles with degrees

5.3.2 Motion Primitive Generation MatLab Script

For the Ackerman steering vehicle, we write a motion primitive generation MatLab script that works with SBPL. This script can be modified for different lengths of vehicles and their maximum turning angles, also different map resolutions. Changing these parameters script can generate a proper motion primitive file.

We define primitives according to 16 number of angles. Figure 5.9 shows 16 number of angles corresponding to their angle values. It starts with 0 angle number to 0 radian and finish with 15 angle number to $\frac{-pi}{8}$ radian. Moreover, in Figure 5.10 can be shown 16 number of angle in lattice. To defining number of angles radian values, we use $atan2(y, x)$ in MatLab. Thanks to $atan2(y, x)$, motion primitives always stays in lattice. All primitives must stay inside lattice so SBPL can read the primitives.

We generate 128 primitives for motion primitives. For each number of angles, four forward and four backward primitives are created in the same

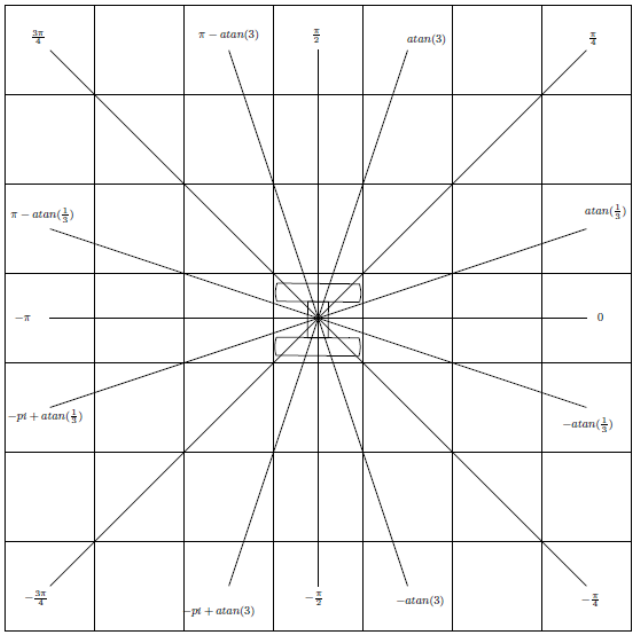


Figure 5.10: 16 number of angles inside the lattice

direction. To define primitives for one angle, we select one short forward primitive, which is from the same angle to the same angle with defined resolution. After that, two curved clothoid primitives are generated, which are the minimum feasible primitives according to the minimum turning radius of the vehicle. These curved primitives' angles are selected to go from their base orientation to one-up angle and one down angle. For example, if a robot in orientation 0 angles, it can only move to angle 1 and angle 15. Thus, there is one connection point inside the lattice for curved paths and last long forward primitive generated for this point. This long primitive has a lower cost value than other primitives, so we expect for far goals planner can find fast motion paths. Figure 5.11 represents 3 possible arrival orientations in the lattice. In the same way, we generate backward primitives.

Finally, using G1fitting library with motion generation MatLab script we obtain '.mprim' file according to SBPL. In Figure 5.12, generated motion primitives can be seen according to 30 degree maximum steering angle with 0.05 meter resolution. Inside the lattice red points (in Figure 5.13) show short straight primitives that are related to resolution and blue ones with black arrow represent long straight and curved clothoid curves. These all curves are feasible according to minimum turning radius.

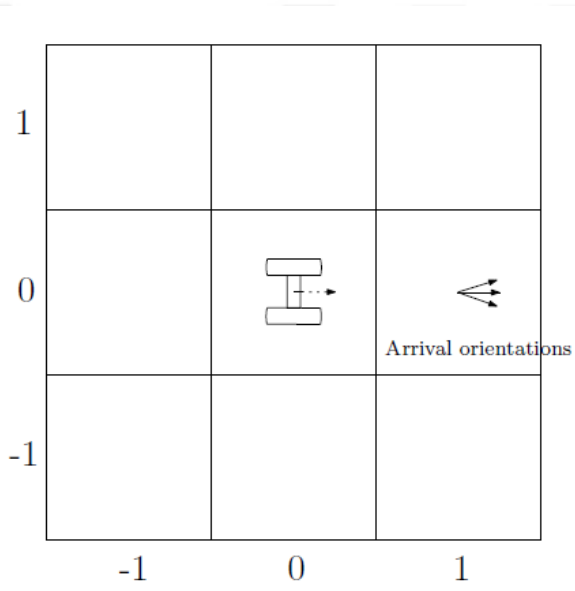


Figure 5.11: 3 different arrival orientation for one number of angle position

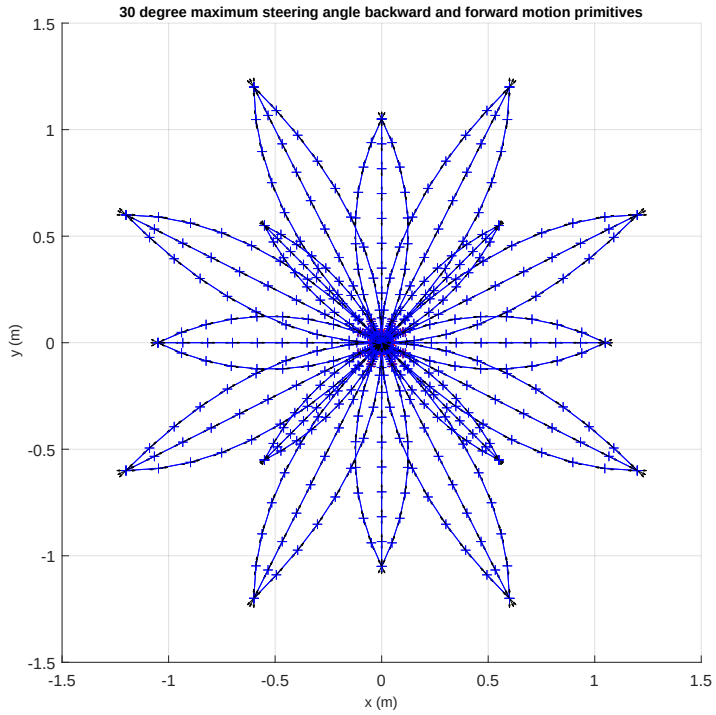


Figure 5.12: According to 30 degree maximum steering angle motion primitives (forward and backward)

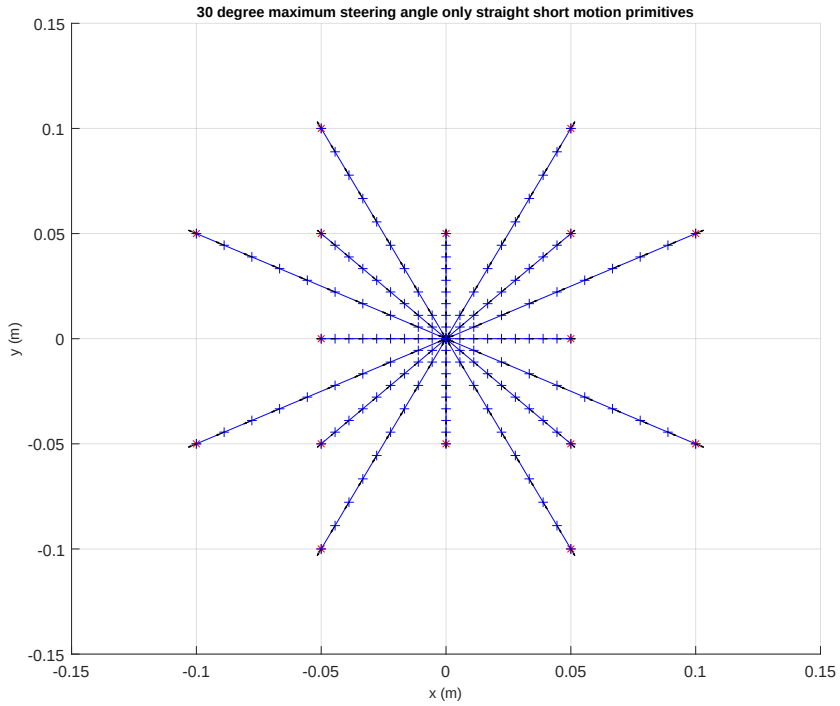


Figure 5.13: According to 30 degree maximum steering angle only straight short motion primitives

Chapter 6

Robot Operating System (ROS) and Planning Libraries

6.1 Robot Operating System (ROS)

Robot Operating System (ROS) is a robot software platform that has a flexible structure and is used in robot applications [1]. It contains many libraries, task structures, and tools for various robots. Pyo et al. mention that ROS, as a meta-operating system, produces, maintains and distributes program packages for a range of uses, and it has created an environment to deliver user-created packages [39]. On the other hand, it is very important to work with Ubuntu and to be open source. In this way, it has spread very quickly with the effect of being open source. Thus, ROS has become very popular today, and many resource books have been published. In addition, there are help pages on the Internet that answer many questions. Having a large number of developers also keeps the system up-to-date and advances.

Figure 6.1: ROS equation

source: https://www.ros.org/about-ros/

If to talk about ROS in more detail, some packages do various works in

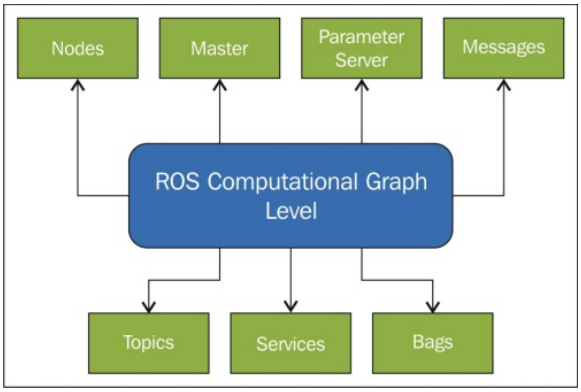


Figure 6.2: ROS computational graph level

ROS. These can be packages that run the robot or introduce sensors. Each package contains executable nodes. This node structure communicates with each other through messages. Quigley et al. explains very detailed in [40], the fundamental concepts of the ROS. For more clarity we explain ROS features briefly.

Node: Nodes are the executable unit of ROS. It can be scripts from Python or compiled source code from C++. Nodes can be single or designed modular.

Messages: Messages are communication elements between nodes. Messages can consist of other messages.

Topics: Topics are used by publishing messages from one node to another node. In this situation sender publishes the topic, receiver subscribes to the topic. One node can both subscribe and publish multiple topics.

Services: Services are another method of communication for nodes on the ROS. Services are based on a call-and-response model. This make main difference from topics. While topics work publish-subscribe, services only provide data when they are called.

Parameter Server: Parameter server stores related configuration values of nodes. Giving new values to the parameter server, one can reconfigure related node parameters even while running the ROS.

Master: Master is a general control element on ROS. The role of master is managing the nodes and tracking publishers and subscribers in ROS ecosystem.

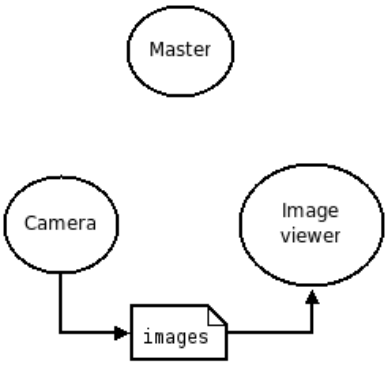


Figure 6.3: ROS master, topics and message

Bags: Bags are the recording and visualization data log element in ROS.

In Figure 6.3, camera and image viewer can be seen as a node. Image topic holding some image messages publishing from camera node subscribing to image viewer node. At the top master is controlling these nodes and topics.

In this work, the most important reasons for choosing the ROS is its superior capabilities in terms of simulation, having different tools for many robots, supporting many examples in different software languages, consisting of a very modular structure, and finally being an open-source with an active community.

6.2 Navigation Stack

ROS Navigation Stack [3] is a group of ROS packages that written for navigation tasks. According to Zheng, the navigation stack's role is to process data from odometry, sensors, and the environment map to establish a stable route for the robot to navigate [48].

Figure[1] 6.4 shows navigation stack setup. Main part of navigation stack is the move_base. Move_base includes global and local planner inside and it subscribes other information from outside. Map, sensor, odometry, and AMCL data constitute outside data. Map server publishes map data to the global planner so the robot can make a global plan. Moreover, Adaptive Monte Carlo Localization Algorithm (AMCL) works with robot sensors and odometry data and publishes robot positions for both local and global planners. Using all these data move_base first makes a global plan after that

[1] `http://wiki.ros.org/navigation/Tutorials/RobotSetup`

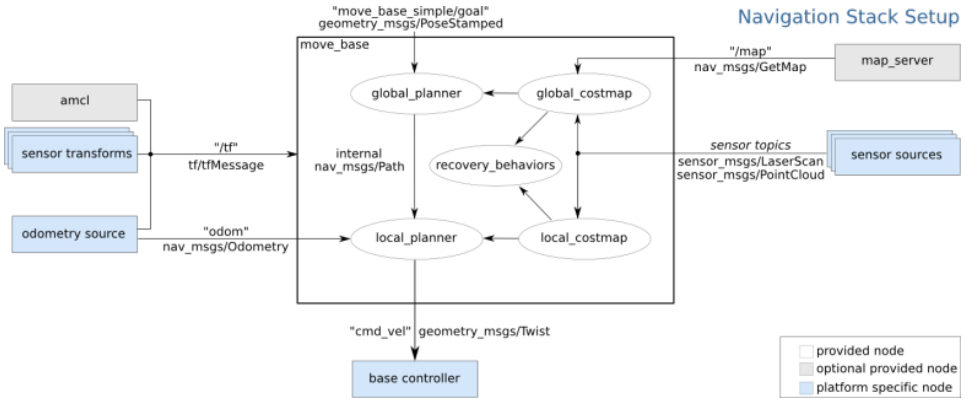


Figure 6.4: ROS Navigation Stack

according to sensor data generates a local plan that can follow the global plan at the same time avoids from obstacle to publishing feasible velocity commands to the controller.

Navigation Stack includes three different local and global planner packages. In this project, we only use one global planner that is global_planner package. this package is an advanced version of old Navfn package. It comprises A* and Dijkstra algorithms. Changing configuration parameters it can be set. In addition, Zheng explains parameters and how to tune them in [48].

6.3 Search-based Planning Laboratory (SBPL)

SBPL is an open source stand-alone C++ library which has no dependencies other than the C/C++ standard library. SBPL includes different search-based planning algorithms with various discrete environment classes. SBPL can be used easily in Linux platform and it has packages for ROS.

As mentioned, SBPL uses search-based planning algorithms. The search-based method uses graph search from the environment, and it has two main problems. The first one is how to turn the environment into a graph, and the second one is how to find the best solution from this graph in between the start and goal points.

Firstly we mention about how to discretize the environment. In SBPL, there are several classes for this purpose. Figure[2] 6.5 shows these classes. These are proposed for several objectives but in out project we use EnvironmentXYTHETALAT that made for robot navigation. EnvironmentXY-

[2] `http://docs.ros.org/en/diamondback/api/sbpl/html/classDiscreteSpaceInformation.html`

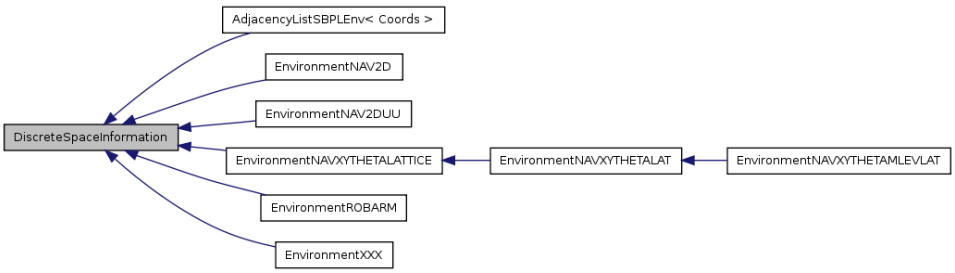


Figure 6.5: DiscreteSpaceInformation class reference

THETALAT is extended version of EnvironmentNAVXYTHETALATTICE. EnvironmentNAVXYTHETALATTICE is the base class for configuration of the planning environment. It uses lattice-based graph problem that is described by the position and the orientation (x, y, θ). Moreover, it accepts motion primitives.

Second problem is planner part. SBPL include different search-based planners. Figure[3] 6.6 shows these planners. Also in Chapter 2, we explain some of these algorithms. Although our environments static, in our work we use ADPlanner for future tasks. ADPlanner include AD* algorithm for finding fast solution for complex and dynamic environments.

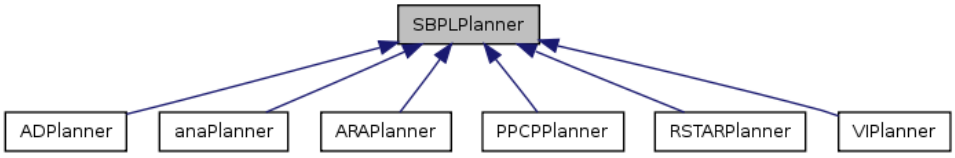


Figure 6.6: SBPLPlanner class reference

For communication in between SBPL library and ROS we use sbpl_lattice_planner plugin from ROS packages [17]. The SBPL lattice planner can be used as the global planner for move_base. The planner will generate a path from the initial position to a desired goal position using motion primitives with ARA* or AD* algorithms.

6.4 Open Motion Planning Library (OMPL)

In robotics community Open Motion Planning Library (OMPL) is one of the most widely used planning library that developed at Kavraki Lab in Rice University. According to Sucan et al. OMPL is an open source standalone C++ software library for motion planning problems both education

[3] `http://docs.ros.org/en/diamondback/api/sbpl/html/classSBPLPlanner.html`

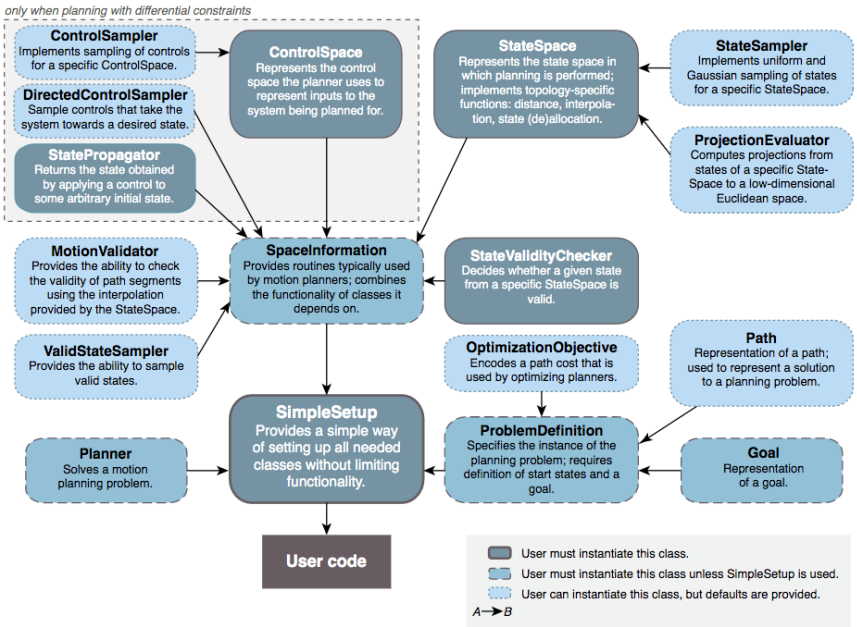


Figure 6.7: The hierarchy of OMPL.

and industrial purposes (This libraries target is mostly motion planning researchers, robotic educators, and robotics industry users.)[46]. It includes many sampling-based motion planning algorithms. On the other hand, it is well integrated with ROS, and its ROS packages available for everyone.

In Figure[4] 6.7 is given to show OMPL hierarchy. According to given hierarchy the following groups reflect the principles of the OMPL methodology identified in sampling-based motion planning:

StateSampler: Provides method for uniform and Gaussian sampling.

NearestNeighbors: Provides planners with a common interface for performing nearest neighbor searches.

StateValidityChecker: Collusion checking with an environment obstacle according to robot constraints.

MotinValidator: Checks validity in between two states. Called a local planner.

OptimizationObjective: Provides a common interface for integrating different cost functions for optimization.

[4] `https://ompl.kavrakilab.org/OMPL_Primer.pdf`

ProblemDefinition: Defines the planning problem specifying the start states and a goal state.

Setting up a planner using OMPL has four main steps. The first one is space initialization. According to constraints and bounds, space must be specified. The second one is configured to space information. Space information includes validity checking and distance information for the nearest neighbor. The third one is problem definition that defines the planning problem specifying the start states and a goal state as well as the optimization objective. The last one is the planner, which is the actual planner that starts the planning and solves the motion planning problem.

Chapter 7

Benchmarking

In this chapter, we talked about benchmarking specifications, starting with benchmarking scripts to continuing to the configuration of benchmarking. After that, we touched on what kind of benchmarking metrics we selected.

7.1 Structure of Benchmarking Scripts

In this section, we briefly mentioned the structure of benchmarking scripts. We answered some questions like what kind of scripts it includes and what kind of tasks they are making.

execute_grid_benchmarking script

We defined run specifications in this script. Using execute_grid_benchmarking script, we can select which environment we will use, what kind of configuration file we execute, which folder will it save results, how many runs we will execute, and so on.

global_planning_benchmark_run script

We developed global_planning_benchmark_run script to create run yaml files and launching to related ROS launch files. According to given benchmarking configurations (given to execute_grid_benchmarking script), first it composed run configuration files for move base, global planner,and supervisor also it prepared folder structures. Secondly, global_planning_benchmark_run script launched related ROS launch files like roscore launch (for starting roscore), rviz launch (for visualization purpose to start rviz), move_base.launch (for

Figure 7.1: Voronoi graph feasible initial nodes representation for fr079 environment.

navigation purpose to start move base), global planning benchmark supervisor launch (controlling benchmarking), and rosbag_recorder launch (recording all data while benchmarking continuing). Finally, it shutdowns all launched after benchmarking finished process.

global_planning_benchmark_supervisor script

Supervisor script is developed for managing the flow of benchmarking. It starts with creating subscribers, publishers, services, actions, and csv files (that files created for collecting benchmark data). Moreover, it finds all initial nodes in the map using the Voronoi graph method (in Figure 7.1 all Voronoi points represented with blue dots. But some of points are not feasible for vehicle and we select our initial and goal nodes from red dots that are feasible for vehicle). For each initial node, we decide to select three different goal points. One of them are selected from the farthest node to the initial node and the others are selected randomly from the rest of the points. Also, according to selected these points, we record mean and minimum passage width environment metrics data.

As mentioned before, the supervisor starts to move base. This means that it sends goal and initial nodes with a random orientation to the motion planner to make a plan in between these nodes. Here, planners can be global planners inside navigation stack, SBPL, or OMPL planners. When move base return with feasible path plan, the supervisor counts this path as a feasible path and save related information from the path like euclidean or Voronoi distances, execution time, start and endpoint positions and orien-

tations, all path, finally, feasibility value (if feasible 1, if not 0). If it is not a feasible path, the supervisor saves it as an unfeasible path. After touching all initial points and finding three different paths from one initial node supervisor finishes its duty and saves all data to relevant csv files.

7.2 Configurations of Benchmarking

ROS Global Planner

For ROS global planner, we used only one parameter, which is planner type. We used Dijkstra and A* planners for unicycle model vehicles. On the other hand, we added robot radius and lethal cost parameters for different benchmarking runs. If one wants to change vehicles different than Turtlebot3 or if one wants to make benchmarking for various types of vehicles robot radius parameter can be changed. Various lethal cost parameters can give different results. We used only one parameter for the lethal cost that is 253 to use more area in the environment. This high value causes more narrow paths to walls. When cost decreased, paths getting far from walls, and planners choose middle points while a route is passing from doors.

OMPL Global Planner

For OMPL global planner configuration similarly to ROS global planner we select planner type as RRT*, RRTconnect, and PRM*. Again, we keep robot radius and lethal cost value same as ROS global planner but it can be changed for future improvements or different tests. Moreover, we define time out parameter for benchmarking. As known OMPL planners has a time bound and choosing 2, 5, and 10 seconds, we compared performances of planners.

SBPL Global Planner

For SBPL, we investigate main two different vehicles. One is a unicycle model vehicle, and the other one is a bicycle model vehicle that has steering geometry. Both of them include planner type, robot radius, and resolution configuration parameters. We make some runs, and we realize that both AD* and ARA* gave quite similar results since we use a static environments. We decide to use only AD* planner for future improvements in dynamic environments.

We generate different primitives for benchmarking, plus we examine what kind of results we will get when we changed primitives. For the unicy-

cle model, we generate three different primitives which are 7, 9, 11 primitive per angle. That means we select 7, 9, or 11 primitives that contain backward, forward, sideways, and curved primitives for each angle. 112, 144, 176 backward and forward primitives are generated for 16 angles. On the other hand, we investigated 4 different maximum steering angles with only forward or forward and backward move types for the bicycle model. Steering angles are defined as 20, 25, 30, 50 maximum steering angles for the same number of total primitives. With this combination, we looked, is various steering angles affect maneuverability with a backward move for different environments.

7.3 Metrics of Benchmarking

Metric selection is important for benchmarking tasks. Different approaches for metric selection can be seen in these works [8] [12] [35] also there are various methods for metric selection. In our work, we selected six different metrics in the central two sections. We separate metrics for environment metrics that give information about environments so we can compare them. Furthermore, performance metrics that we can analyze which planner better than the others.

7.3.1 Environment Metrics

For comparing different environments, we define three different environment metrics. We select these metrics to show how challenging environments we use in benchmarking. Mean passage and minimum passage width gave information about how wide the environment is. On the other hand, the number of obstacles traversed gave information about how crowded the environment with obstacles which can be tables, chairs, or walls.

Mean Passage Width

Mean passage width is a metric to show how the environment is challenging for comparing with size or largeness. We obtained the mean passage width value from the Voronoi graph for each path. While we are generating the route from the start point to the goal point, we collect Voronoi radius values for each path. As known from the Voronoi explanation part (Chapter 4), each path includes many Voronoi nodes, which are generated with Voronoi radius value. Taking mean values of these Voronoi radius values, we obtain mean passage width for environments. This metric shows us how robots far

from walls on average. This value is important for comparing environments because it shows maneuverability capacity and wideness of environments.

Minimum Passage Width

Minimum passage width is a metric to show the tightest passage in the environment. We obtain this value again using the Voronoi radius value from the path, and we find the minimum radius for the start and goal route. Mostly, this value comes from doors, and it gives information about the challenging part of the environment.

Number of Obstacle Traversed

The number of obstacles traversed shows the complexity of the environment. It is counted from obstacles, that are touched to a straight line drawn in between the starting and ending points on the map. If the obtained number of obstacle traversed values came high, we can make comment as environment complexity is high. Moreover, we can say making a maneuver or finding a plan can be difficult.

7.3.2 Performance Metrics

For comparing different global planners, we define three different performance metrics. We select normalized planning time, normalized planning length, and feasibility rate metrics to show performances of the global planner algorithms for various environments and to remark what kind of configuration effect planners positively or negatively. Moreover, we normalize metrics values dividing with Voronoi path value for each path. Thus, we compared them on the same scale both for planning time and planning length. While obtaining planning time, we divide the time value for a created path with its Voronoi path value from the start point to the end point. Similarly, we make the same thing for normalized planning length. We divide the path length value with its Voronoi path value. As a result of it, we get normalized performance metrics values. On the other hand, we define the feasibility rate for each path. If the planner finds a possible path for a given start and end point, it returns a path feasible. Otherwise, it returns an unfeasible path value.

Normalized Planning Time

Normalized planning time is a metric value that shows planners' performance on a time scale. Normalized planning time is defined as the measured time in

between the initial and goal node path divided to Voronoi length value that again obtained from the same initial and the goal node. We use normalized value because of making equitable comparisons for global planners. Each time value is scaled according to their Voronoi paths. These Voronoi paths are extracted from the general Voronoi graph (this general Voronoi graph and its paths are always the same for all points, and it is generated once at the beginning of the supervisor process.). Finally, we gather normalized time values for each environment and compared fairly the global planner algorithms considering environments.

Normalized Planning Length

Normalized planning length is a metric value that shows the performance of the planners on a scaled distance. Similarly, like planning time metrics, we use Voronoi length for normalization value in the planning length metric. We take actual planner length from global planners for the given start node and the goal node. Then, this planning length is divided into Voronoi length to obtain normalized planning length. We use normalized planning length metric to compare planners how optimal paths they generated. Besides, we investigate the maneuverability of vehicles. For example, if the normalized planning length value is higher than one for vehicles that have motion primitives, we can say the planner found a longer path than the Voronoi path. That meant the vehicle made some maneuvers to find a valid path with the correct orientation.

Feasibility Rate

Feasibility rate is a metric that shows how many times a path is found in one run. Fundamentally, the feasibility rate is counted for each path 1 or 0. After that, all values are collected and take the mean of them to find the run feasibility rate for a specific environment with the given global planner. This value shows for one run according to given initial and goal poses how many paths aborted and how many possible routes found.

Chapter 8

Results and Comparisons

In this chapter, we presented benchmarking run results, and we made comparisons about them. This thesis is not necessarily related to performing the benchmarking run results. It is about to mention the procedure. Mainly, this thesis is about to exploring how to implement and use the results of a global planning benchmark.

First, we introduced environment metric results, and we made comments about environments for showing results and comparison purposes. After that, we presented performance metrics results with different comparisons. Fundamentally, we mentioned six comparisons.

- ROS Global Planner comparisons that include Dijkstra and A* algorithms.
- OMPL library comparisons that include RRT*, PRM*, and RRTConnect algorithms.
- SBPL library comparisons for unicycle primitive base planner.
- SBPL library comparisons for bicycle primitive base planner.
- Unicycle base planner comparisons with motion primitives and without motion primitives.
- Unicycle SBPL comparisons with bicycle SBPL.

8.1 Environments Comparisons

In Chapter 4, we described environments, but we did not compare them. In this chapter, we analyzed environments according to environment metric

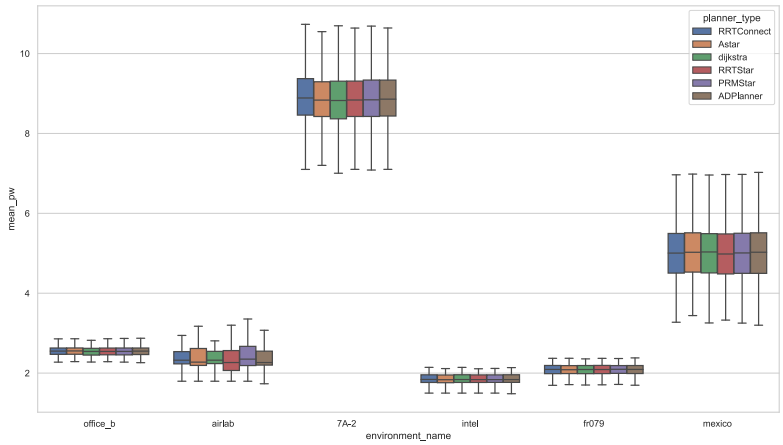


Figure 8.1: Mean passage width box plot for Dijkstra, A, AD*, PRM*, RRT*, and RRTConnect algorithms*

results.

As mentioned before, we explained environment metrics in Chapter 7. These environment metrics are mean passage width, minimum passage width and number of obstacle traversed.

We looked all planner results according to environment metrics and we did not see any difference for different planners as expected. Thus, we decided to give only unicycle type planners results.

8.1.1 Mean Passage Width

By looking mean passage width box plot in Figure 8.1, we clarified 7A-2 is the widest environment in all six environments. It has quite open spaces to make the maneuver. On the other hand, 7A-2 is smaller than the Mexico environment. We expect making a plan in the 7A-2 environment should be faster than the Mexico environment but, it is not an easy assumption because of separated walls in 7A-2. These walls are entirely separate rooms from each other with narrow doors. Thus, it can make plans harder than we predict.

Although Mexico is way bigger than 7A-2, it is the second widest environment because of tiny obstacles and small passages. Comparing the robot size Mexico is a reliable environment to make the maneuvers. However, small scattered obstacles and the large size of the environment make finding a plan harder.

	Airlab	7A-2	Office_b	Intel	Fr079	Mexico
Unicycle	1.09	0.96	0.88	0.83	0.83	0.99
Bicycle	1.36	0.96	0.88	0.98	0.87	1.08

Table 8.1: Minimum passage width for all environments

Office_b third widest environment according to metric results. It includes small rooms, passages, and corridors. As a result of this, the mean passage width of the environment decreases, and environment turns from office to kind of maze. Thus, making plans becomes harder for global planners.

Airlab is the tiniest one inside these six environments. Although it consists of two small corridors mean passage width value is not the lowest one. The wide-open space that is into the middle of the Airlab makes the environment the third smallest one comparing with the mean passage width. However, we cannot say making a plan is simple in Airlab. Airlab is considerably tight and making maneuver pretty hard. Because of it, finding a feasible path is difficult, especially for big car-like vehicles.

Intel and Fr079 are created from real environment data. They include small passages, tiny rooms, doors, and different types of small obstacles. Because of these objects, Intel and Fr079 are the tightest ones according to the mean passage width metric even though it is as big as 7A-2.

8.1.2 Minimum Passage Width

Minimum passage width table is given in Table 8.1. This value shows minimum passage length inside the environments. As seen, we separate the bicycle model and unicycle model because some passages are quite small for bicycle robots. For feasibility reasons, we decide to select proper start and goal points according to robot size. In Airlab, Intel, Fr079, and Mexico environment, one can see the differences.

Table 8.1 are showed results. Fr079 and Intel have one of the tightest passages in all six environments. Although Airlab has the widest passages, Airlab is a quite small area as well it has long corridors when compared total Airlab field. Moreover, Mexico and 7A-2 are wide open areas, but they also have some small passages that makes planning harder.

8.1.3 Number of Obstacle Traversed

The number of obstacle traversed metrics represents the complexity of the environments. Fr079 and Intel are the most crowded and complex ones, according to our defined metric. There are various obstacles and walls in these

	Airlab	7A-2	Office_b	Intel	Fr079	Mexico
Unicycle	1	3	5	6	8	5
Bicycle	0	3	5	5	5	5

Table 8.2: Number of obstacle traversed

two environments. In complexity order, the third row is taken by Office_b that has multiple walls with Mexico. We expect Mexico more complex than Office_b but, because of the wide-open area and random points, the number of obstacle traversed value decreases for Mexico. Finally, the simplest environment is viewed as Airlab.

8.2 Planners Comparison

We compared planners' performance according to given various types of environments. While comparing, we used three different metrics, and we obtain some results. According to people desires, they can create different type of metrics and they can compare from another perspective. However, we decide to use with normalized planning length, normalized planning time and feasibility rate metrics.

In these comparisons, our aim is not to show which planner algorithm is the best. Our main purpose is using our methodology, people can choose their global planners with related vehicles (unicycle or bicycle) according to similarities with our environments.

8.2.1 ROS Global Planners Comparison

In this section, we examined ROS global planners, which are Dijkstra and A*, performance in given different environments. In Figure 8.2, we saw that A* is faster than Dijkstra, but paths of A* are not as short as paths of Dijkstra. A* uses a grid path into the map. Because of this reason, A* finds longer paths than Dijkstra in ROS global planner. Moreover, A* is finding paths in a short time in general because of its heuristic value.

When we compared planners for different environments, we saw that both Dijkstra and A* have a 100% feasibility rate for all environments. Thus, the feasibility rate metric is not important for us while we are comparing them.

In Figure 8.3 and Figure 8.4, we discussed planners performances. Looking for box plots, we can make these comments. First, if one looks for a faster planner, one can use A* for all environments. Furthermore, if one does not

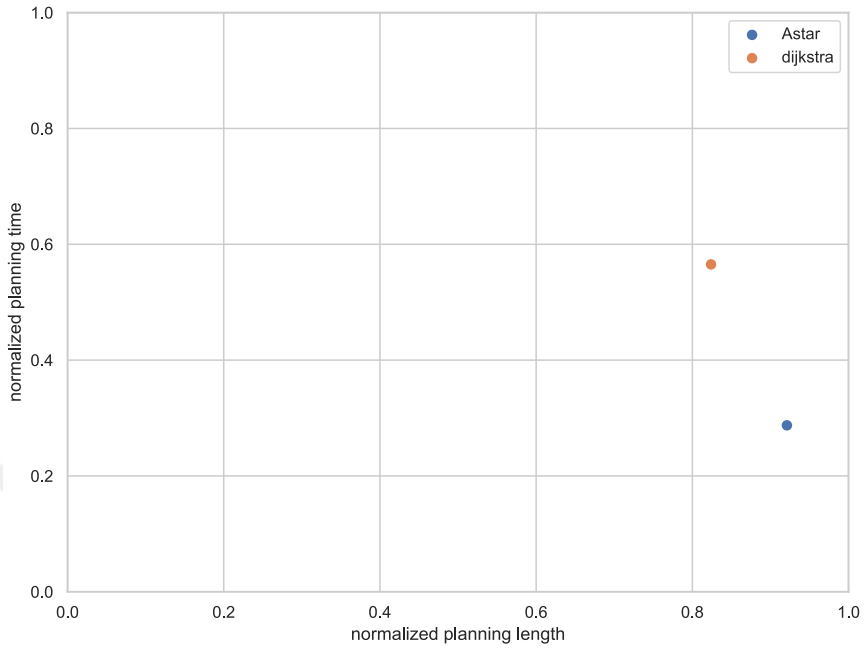


Figure 8.2: Performance metric comparison for A and Dijkstra for mean across all environments*

care about time but cares about the shortest path, one can choose Dijkstra again for all environments. Finally, we can say environments do not affect ROS global planner, one can select A* for fast path finding speed, Dijkstra for small paths.

8.2.2 OMPL Global Planners Comparison

Our second comparison are made for OMPL global planners. Although planning time must change for each environment, we keep same time for every environments. We decided time out as 5 seconds. In Figure 8.5, we can see for 5 seconds time out how changed feasibility rates of planners. According to these graph, we can say RRTConnect feasibility rate is highest because RRTConnect starts to search both from initial and goal point. In Figure 8.6, we can examine environment effects of feasibility rate. We can say when the environment getting bigger feasibility rate decreases. To illustrate, feasibility rate of Airlab is 100% for all planners but for Mexico feasibility rate is very low. Moreover, complexity of environment is effecting feasibility rate. We can see it from Intel environment. When the environment getting complex feasibility rate also decreases.

For all environments, we also compared the performance of OMPL global

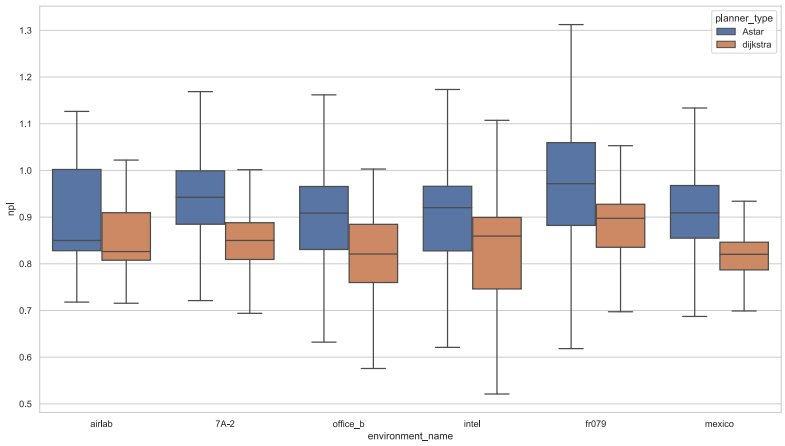


Figure 8.3: Normalized planning length for ROS Global Planners

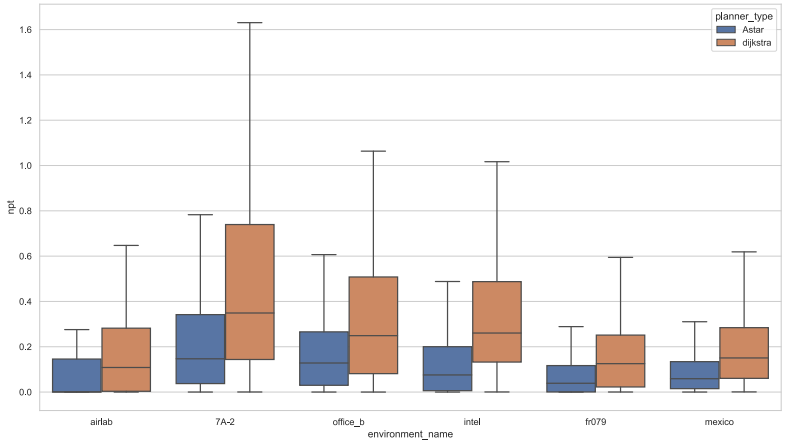


Figure 8.4: Normalized planning time for ROS Global Planners

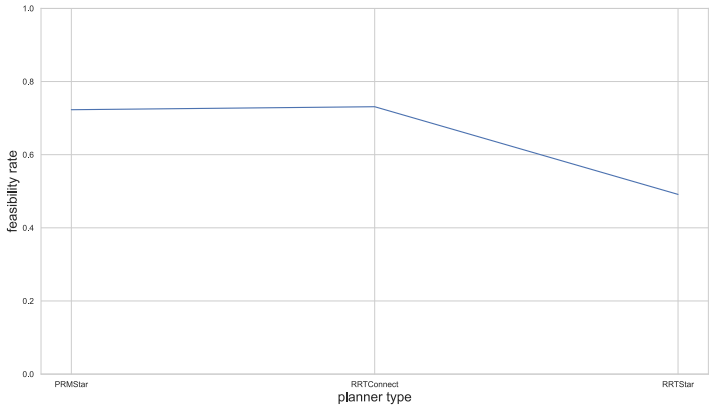


Figure 8.5: OMPL global planners feasibility rates for mean across all environments.

planners by looking at normalized path length and normalized path time. Analyzing Figure 8.7, we can say both RRT* and PRM* are asymptotically optimal planners comparing with length. When we checked the results, we can see they found more or less similar paths that are shorter than RRT-Connect. However, PRM* has good performance in most of the our environments according to our metric outcomes for normalized planning length. Although RRTConnect has a high feasibility rate and the fastest time, it finds long paths. As a result, if one wants to use OMPL planners for unicycle vehicles, one can choose PRM* for most of our environments comparing with planning length. One also can choose RRTConnet for faster plans. However, one must know sampling-based planners are not good feasibility rate for large or complex environments.

We also analyzed PRM* algorithm for different time out values and we compared PRM* with RRTConnect. We decided 2, 5, and 10 seconds as configuration parameters. We looked feasibility rate for environments. As known, when time out increased in sampled-based planners, finding path ratio increase and path length decreases. Figure 8.8 shows the very small amount of increase of feasibility rate with time out. As a result of it, we can say according to our results 2, 5, or 10 seconds do not have significant effect into the feasibility rate. In planning length perspective, from Figure 8.9 planning length get shorter when time increased in almost all environments. In this figure, there can be some small interesting outcomes because of random point effect. Furthermore, comparing with RRTConnect, PRM* with 2 seconds time out has better performance both planning time and planning length in Office_b, Mexico and Fr079 environments.

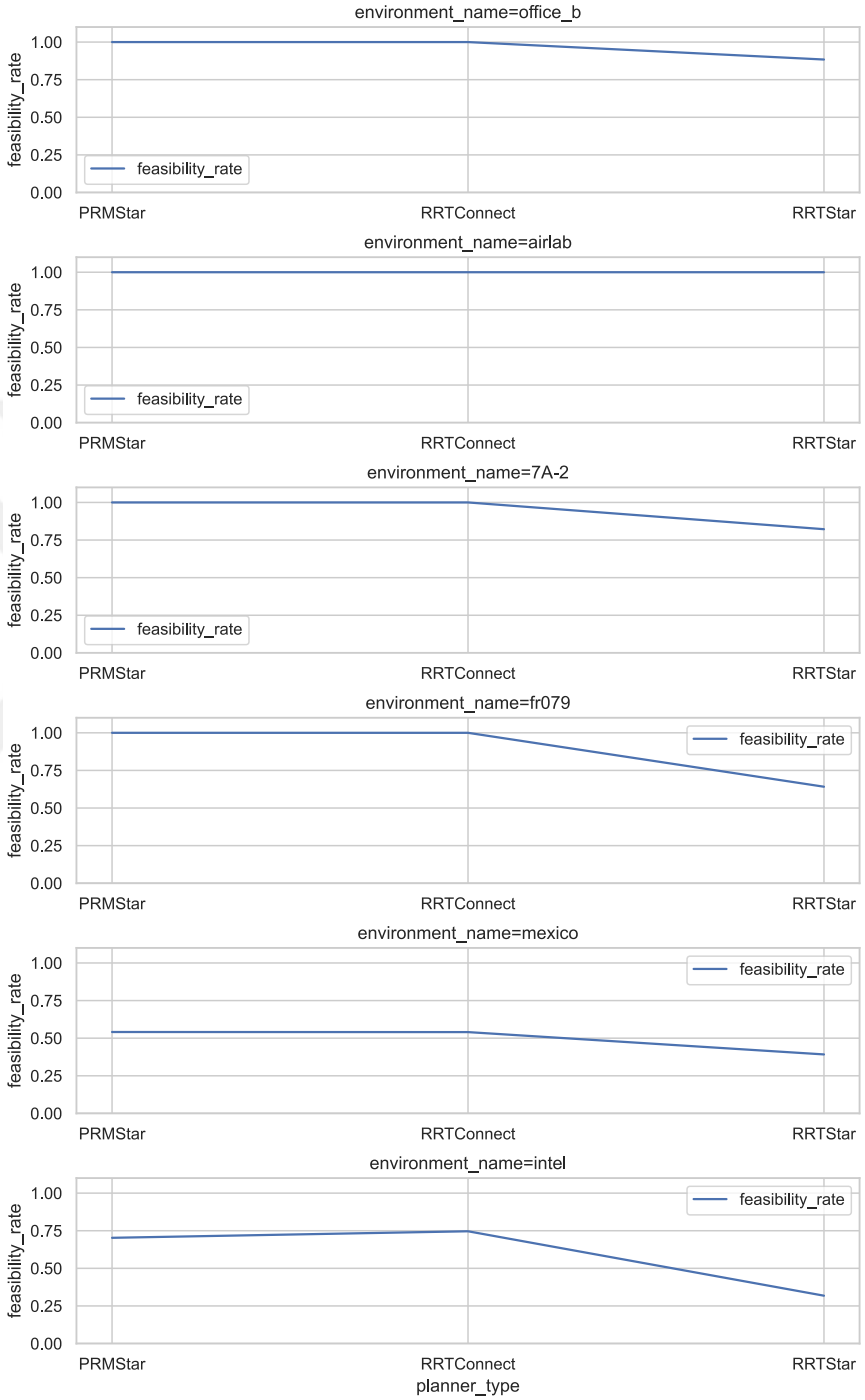


Figure 8.6: OMPL global planners feasibility rates for each environments.

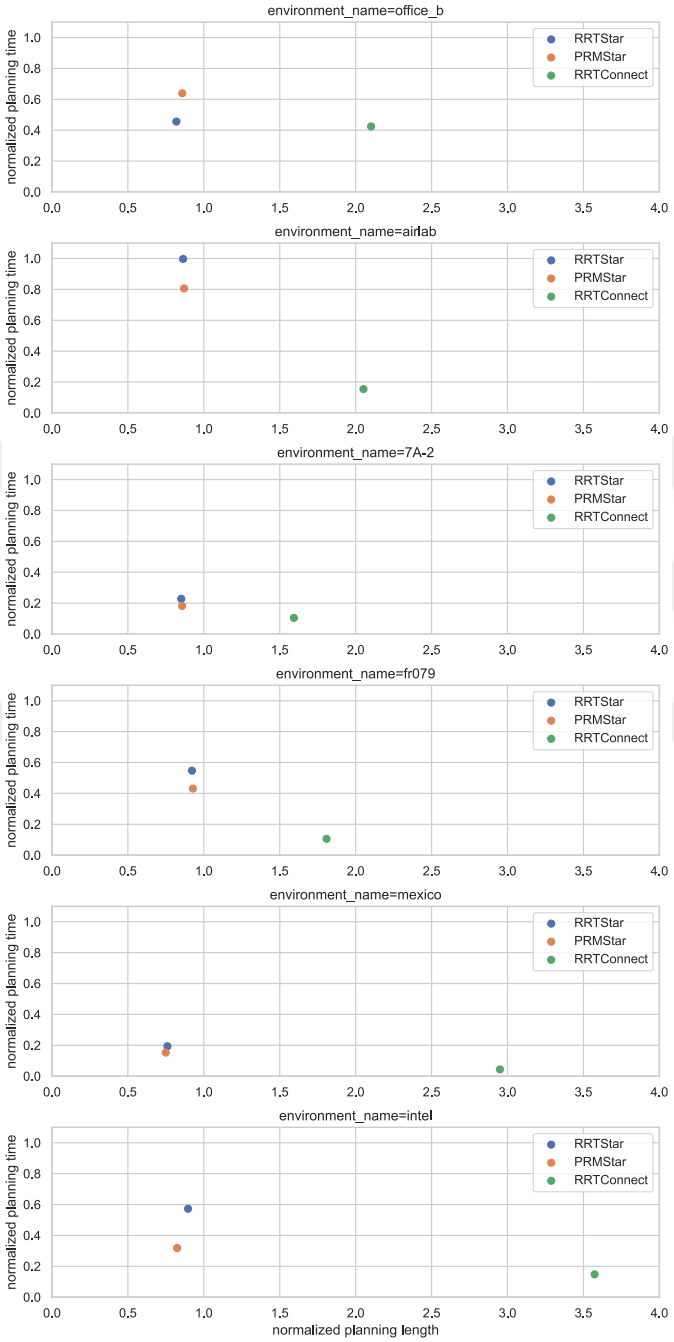


Figure 8.7: OMPL global planners performance metric results for each environments.

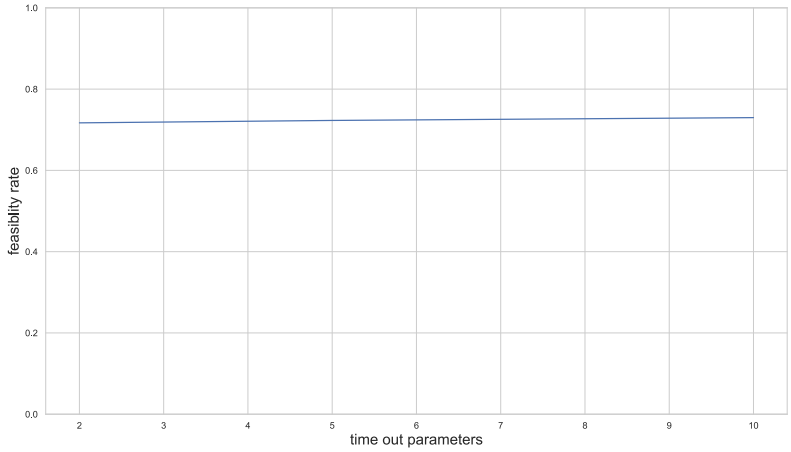


Figure 8.8: PRM global planner feasibility rate according to increased time out (2, 5, and 10 seconds) for collected mean across all environments.*

In conclusion, if one wants to select one planner from OMPL according to similar environments like ours, one can choose PRM* algorithm for asymptotically optimal lengths or the RRTConnect algorithm for good planning time, depending on our results. Moreover, PRM* with fewer time-out configuration parameters can be chosen for fast planning time instead of the RRTConnect algorithm. Time out variable is a design choice for OMPL planners. The time-out parameter is important according to the size or complexity of the environment. The designer must choose this parameter carefully to define planning time for environments.

8.2.3 SBPL Global Planners Unicycle Vehicle Comparison

Our third results are about the comparison between SBPL global planners with motion primitives for unicycle vehicles. For this test, we defined an initial epsilon (ϵ) as 3 and a decrease epsilon step value as 0.2, also the allocated planning time is set to 20s. For motion primitives, we used SBPL MatLab motion primitive generator and we generated three different motion primitives that are based on 7, 9, and 11 primitives per angle for 16 angles. Totally 112, 144, 179 motion primitives are generated for unicycle vehicle.

Previously we made some runs, and we saw ARA* and AD* do not have significant differences since we used static environments. Thus, for future works, we decided to go with AD* planner.

When we checked our results in Figure 8.10, we saw there is no sig-

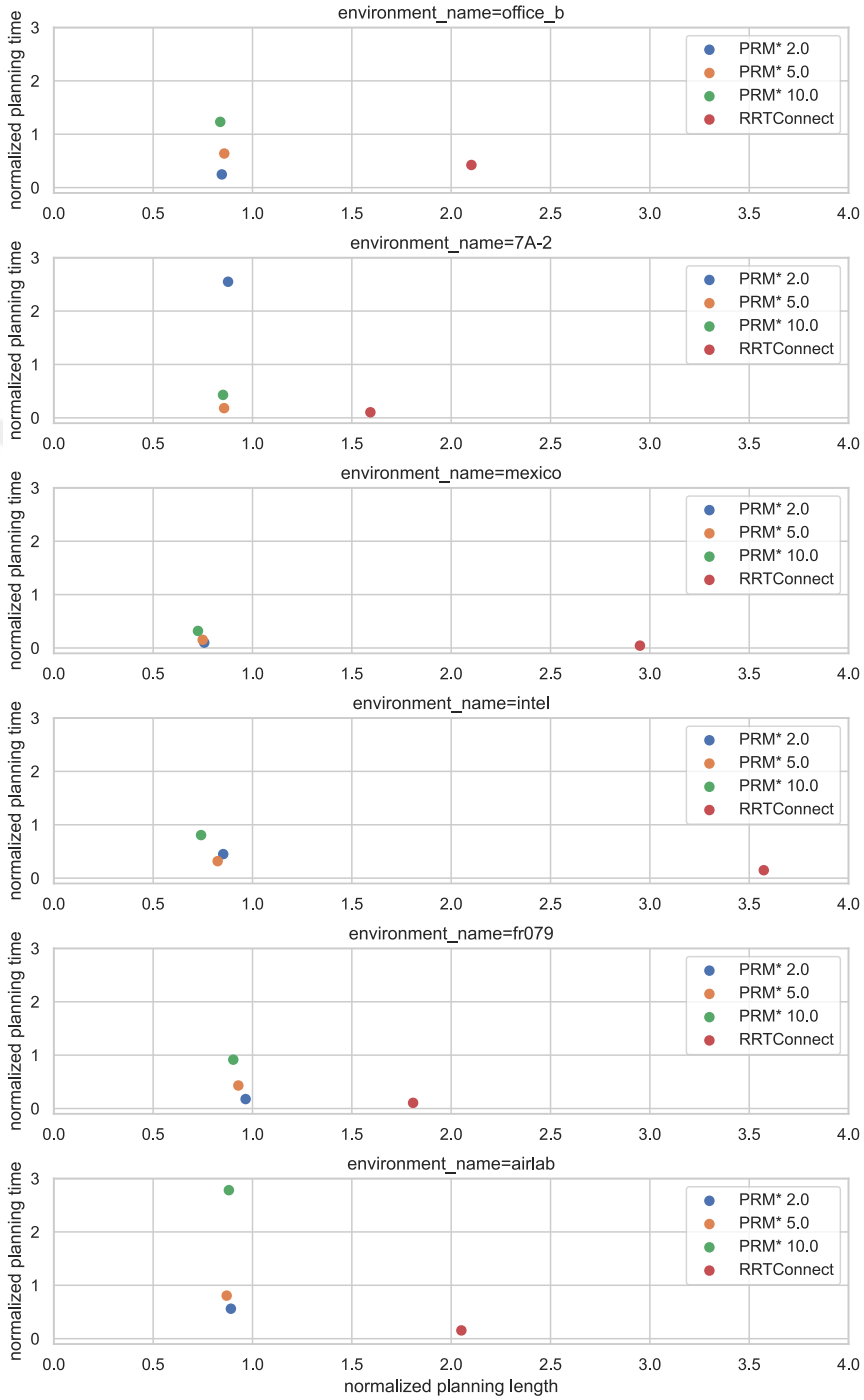
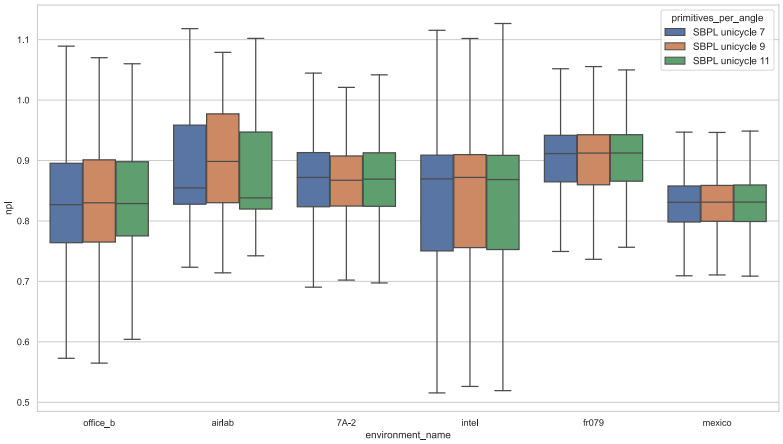
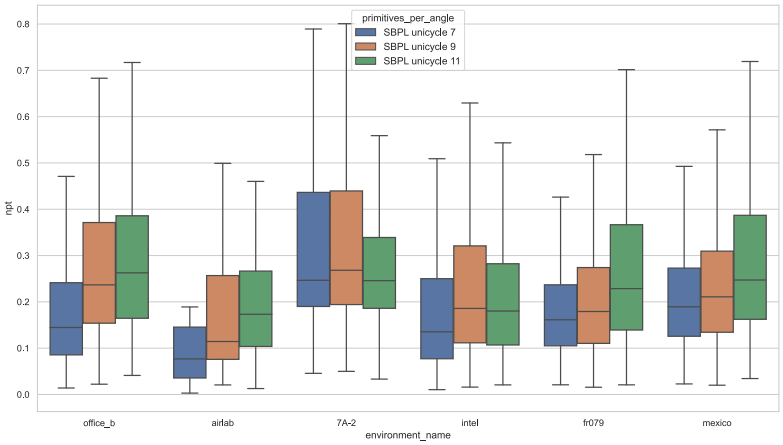


Figure 8.9: PRM global planner performance according to increased time out (2, 5, and 10 seconds) compared with RRTConnect planner for collected each environments.*



(a) Normalized planning length for SBPL global planner



(b) Normalized planning time for SBPL global planner

Figure 8.10: SBPL global planner unicycle vehicle run results according to given environment for 7,9 and 11 primitives per angle configurations

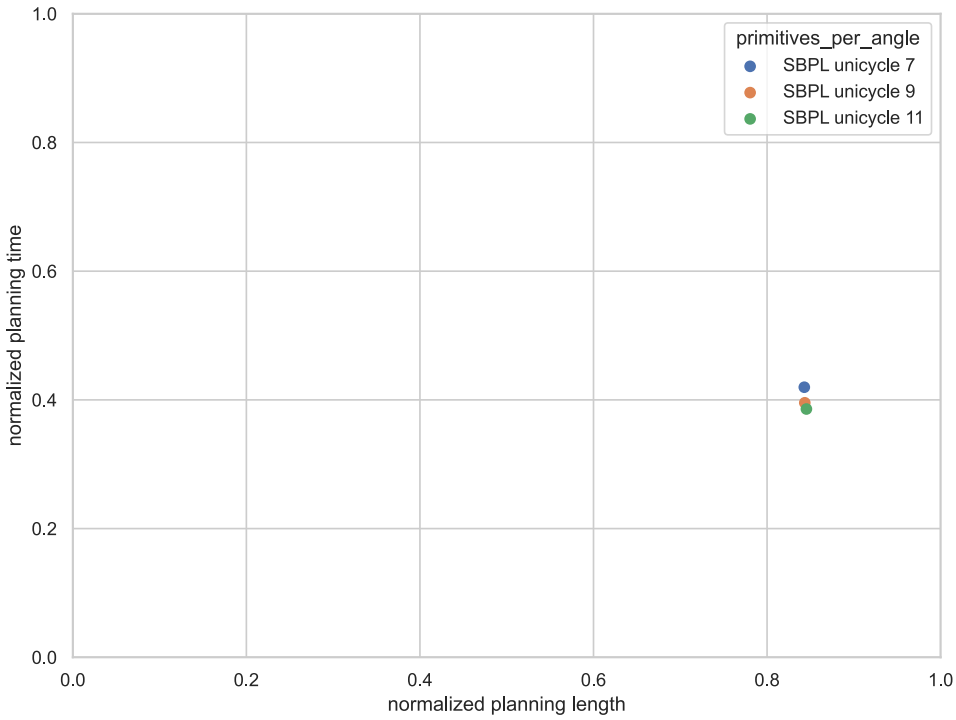


Figure 8.11: SBPL global planner unicycle vehicle performance run results for mean across all environments

nificant difference between planning lengths. The vehicle is a differential drive, so the number of primitives is not affecting the length of paths. Although expected sight is fewer primitives can make a faster plan, according to our run results we saw that 11 primitives per angle faster than the others when we collect all runs in Figure 8.11. For this result, our comment is when increased primitives finding path becoming easy because the planner is searching faster with more primitives for the goal point in the map. Moreover, random positions and orientations can affect these results. Sometimes given arbitrary remote points with tough orientation can be a problem for fewer primitives, and it can take some time for them.

The feasibility rate is 1 for all environments except Mexico. We saw some small feasibility rate decrease in Mexico (0.96 average feasibility rate for Mexico). Given random points, low epsilon value and allocated time can cause these small feasibility rate changes. Increasing epsilon value or allocated time can also increase feasibility rate of Mexico. Moreover, we saw from Figure 8.12, when we increased primitives per angle value feasibility rate did not increase considerably. The increase of feasibility rate is from 0.98 to 0.99 for 7 to 11 primitives per angle is almost negligible comparing

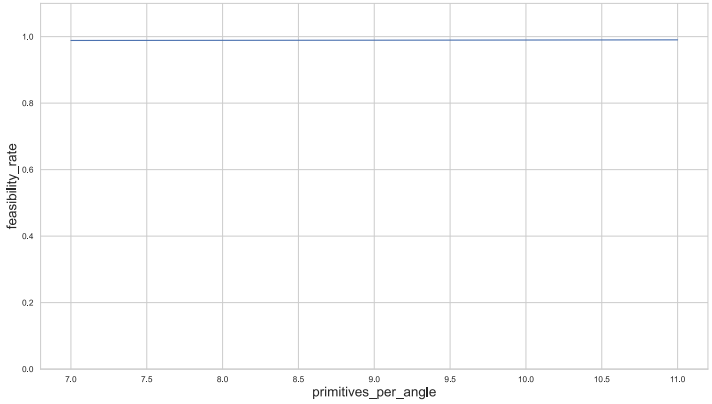


Figure 8.12: SBPL global planner unicycle vehicle feasibility rate for mean across all environments

the total feasibility of planners.

Since feasibility rate is 1 for all environments we can make comments about performance of primitives per angle for environments. Our results are given in Figure 8.13. According to these results, AD* planner with 7 primitives has a good performance for very large or very small environments. In small or big size environments, it can search very fast because it has fewer primitives. On the other hand, AD* planner with 11 primitives has a good performance rest of the environments except Fr079. In Fr079, we saw 9 primitives have better performance. Because of arbitrary points and random orientations, this result can occur. Thanks to more primitives, AD* can find paths faster than the others in complex and middle-size environments.

For the final comment, if someone wants to use SBPL planners for selected environments to take into account minimizing time, one can choose a few primitive per angles for very large or very small environments. Otherwise, using more primitives gives better results for both times and lengths.

8.2.4 SBPL Global Planners Bicycle Vehicle Comparison

Our fourth results are about the comparison between SBPL global planners with motion primitives for Ackerman steering vehicles. We defined the same initial epsilon value, decreased epsilon step, and allocated planning time as SBPL unicycle for this test. We used SBPL motion primitive generator, which we created a MatLab script for generating motion primitives. We produced four different primitives that are based on 20, 25, 30, and 50

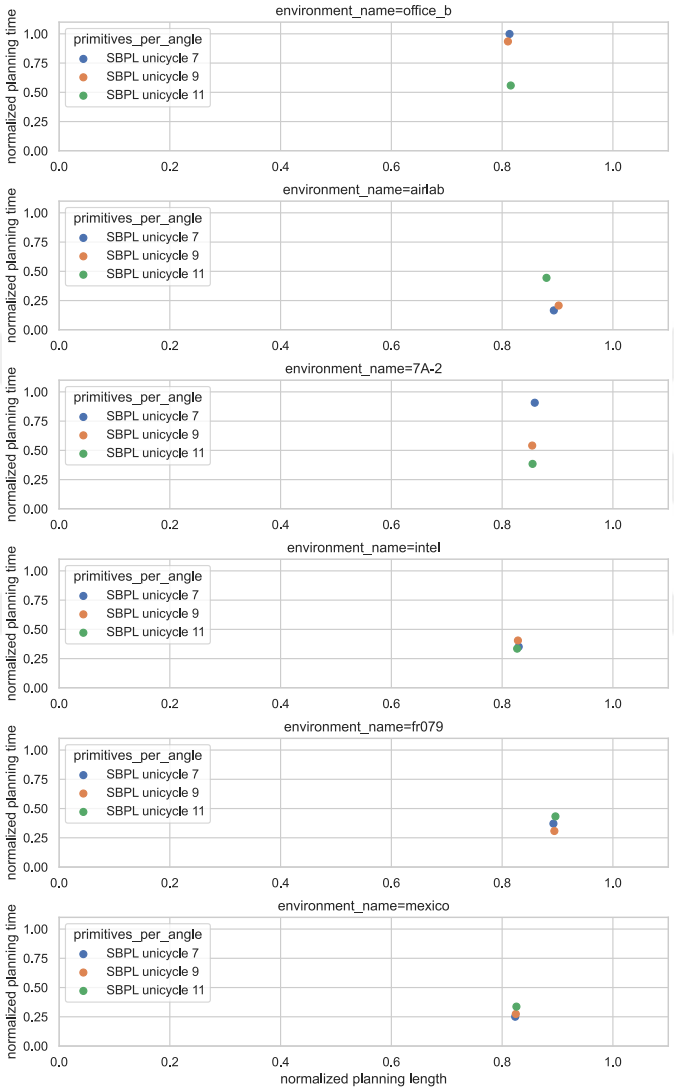


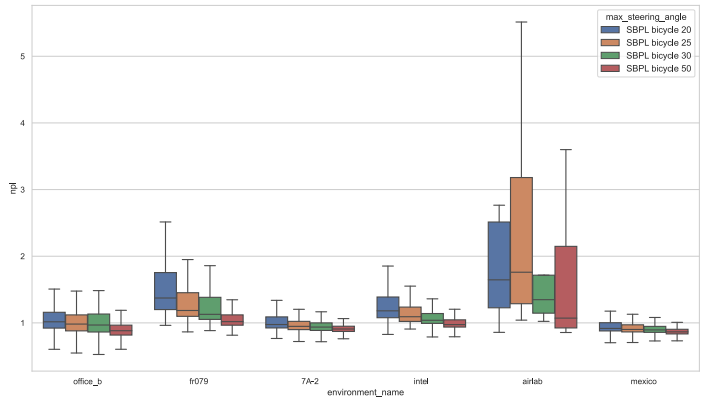
Figure 8.13: SBPL global planner unicycle vehicle performance metric results for each environments

maximum steering angles for 16 primitive angles. We selected these values according to our reference Agilex Hunter 2.0 vehicle. All generated primitives include the same number of total primitives, which consist of 32 short forward and backward, 32 long backward and forward, 64 curved long backward and forward motion primitives. In total, 128 motion primitives were generated for each maximum steering angle to make a fair comparison. On the other hand, we generated 64 only forward motion primitives for only forward movement.

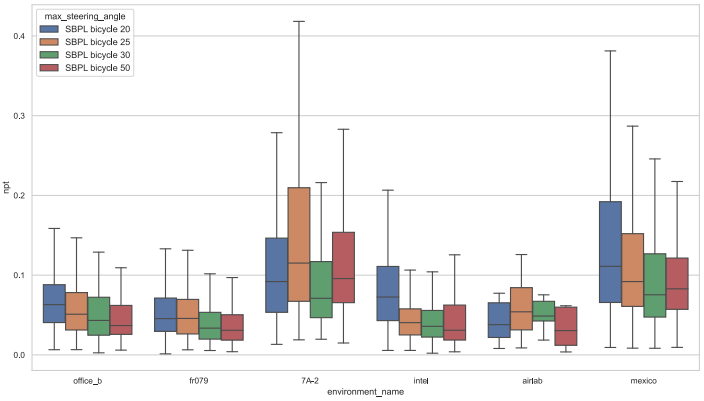
By looking at Figure 8.14(a) can be seen considerably high planning length. Because of steering geometry, the planner made different maneuvers to find a correct position with true orientation. These back and forward maneuvers increase the way taken. We can see this maneuver effect clearly in small or complex environments such as Airlab, Fr079, or Intel. These small or complex environments cause to make too much maneuver to robot and increase path length. Moreover, Mexico and 7A-2 types of environments have enough space to make the maneuver, so planning length is not affected as much as small environments.

High planning time can be seen in wide-open or slightly large environments such as 7A-2 or Mexico from Figure 8.14(b). While the environment is getting bigger or wide-open, the planner searches for more possibilities and tries to find optimal routes decreasing epsilon value. This process takes some time, and it increases planning time. Moreover, when Airlab environment are investigated, it can be expected low planning time. According to our results, Airlab planning time is not low but finding a path in a small environment is very tough for slightly big car-like vehicles. Planners need too many maneuvers to find correct orientation in small environments. Because of it, high planning time value is acceptable and normal for small environments.

We shared Figure 8.15 for investigating the effect of maximum steering angles. As seen, increased maximum steering angle gives high performance for planning length according to our run results. Moreover, the backward and forward movement has better performance than only forward movement. There are two remarkable results in our runs. The first one is 50 degrees of maximum steering angle with the only forward movement has better performance than 30 degrees of maximum steering angle with both forward and backward movement. This means that sometimes steering angle value is more important than forward and backward moves to make maneuvers. The second one is 20 degrees of maximum steering angle for backward and forward motion gives a better result from 25 degrees one. This can happen because of arbitrary given goal points and their random orientations. Since



(a) Normalized planning length for SBPL global planner



(b) Normalized planning time for SBPL global planner

Figure 8.14: SBPL global planner car-like vehicle run results according to given environment for 20, 25, 30, and 50 maximum steering angle configuration

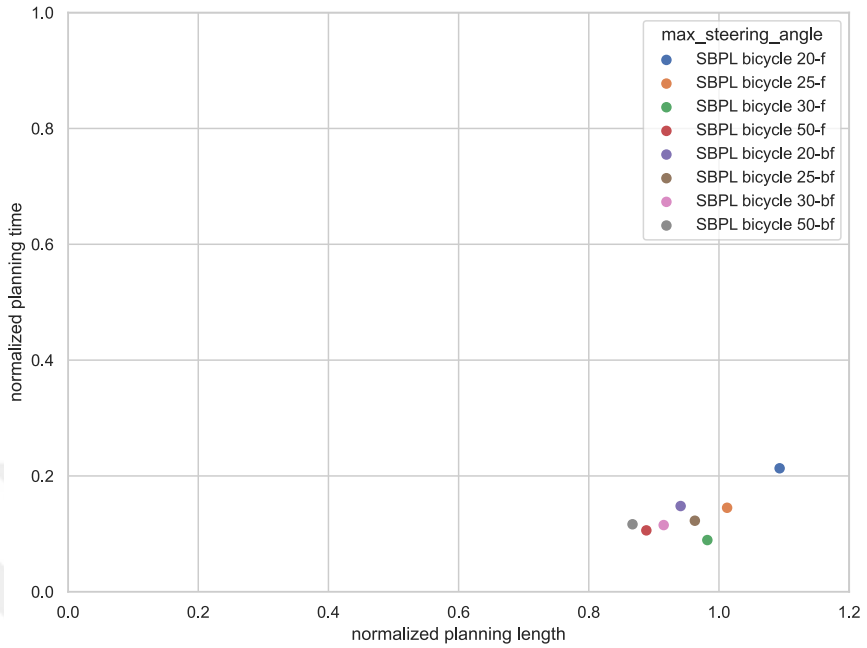


Figure 8.15: SBPL global planner car-like vehicle performance metric results for mean across all environments

there are no significant changes between 25 and 20 degrees, by luck, 25 degree maximum steering angle one took more difficult goal points than 20 degrees, and it needs too many maneuvers to find the goal point with the correct orientation. Thus, the planning length of 25 degrees increases by comparing with 20 degrees one.

On the other hand, by looking to Figure 8.15, we cannot compare directly planning time for different maximum steering angles. We cannot make a straight expression about the best planner in our run result because of arbitrary points and their random orientations. Tough positions for points can cause some extra times for planners even their maneuver capability is high.

We gave Figure 8.16 to compare backward and forward primitives effects with different maximum steering angle configurations on our all environments. Although we did not see considerable changes for normalized planning time, we realized significant changes in normalized planning length. While environments are getting larger or wide-open, the importance of big maximum steering angle value with backward and forward movement increases. High steering angle meas small turning radius and better maneuverability with short paths.

We realized some exceptional results in the Airlab environment. Accord-

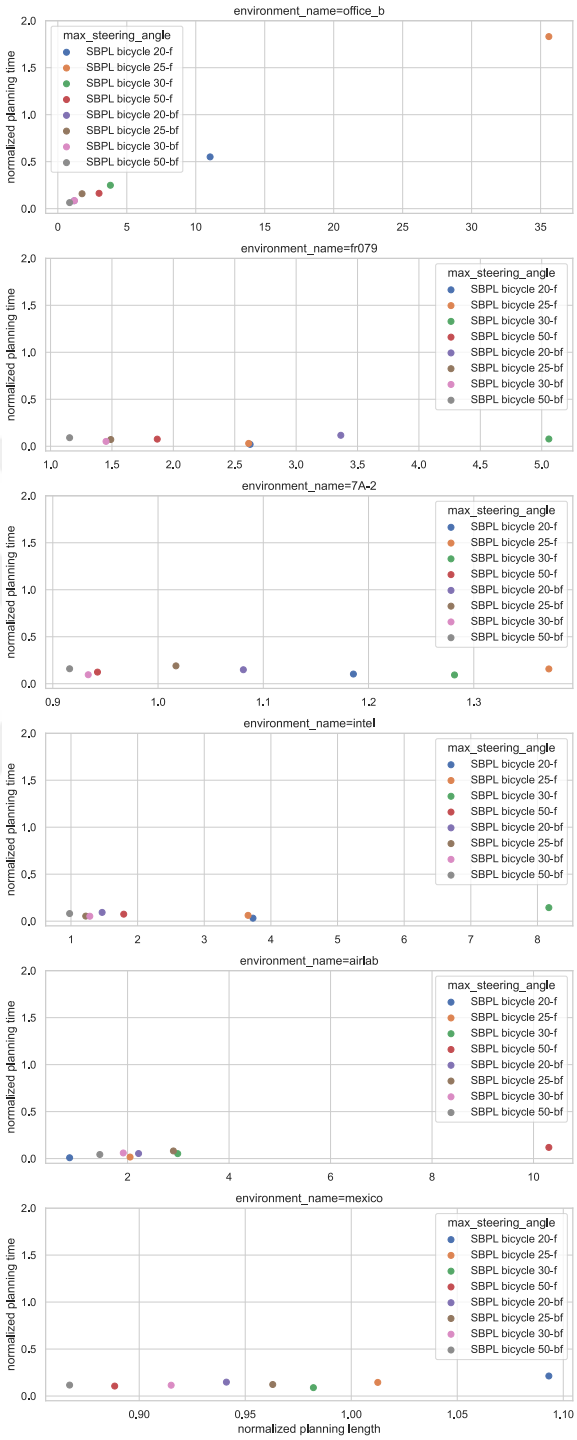


Figure 8.16: SBPL global planner bicycle vehicle performance metric results for each environments

ing to results, 20 degrees only forward movement has the best result but, this is not expected. When we checked 20 degrees with only forward movement performance results, we commented that it only found a few paths in Airlab, by luck, these random points and orientations are selected to make the plan easy even for a very low maximum turning angle value. In conclusion, random points cause problems in some situations. Except for 20 degrees only forward movement result in Airlab, we can say if one selects a high degree of maximum steering angle vehicle, one can obtain better performance for all environments. If one has a wide-open or large environment and s/he is not caring about planning length. S/he can select a low maximum steering angle with a backward and forward movement or even a high steering angle with only forward movement (in 7A-2 and Mexico environment results shows, 50-f similar performance with 30-bf one). On the other hand, complex environments mostly need high steering with forward and backward movements. If one has complex environments like Intel or Fr079, s/he may see a better result in planning length by using high steering with forward and backward motion primitives.

Final comments are made about feasibility rate. According to our results, we recognized that the feasibility rate of the car-like vehicle in our environments are slightly low (Figure 8.17). This can be caused by some reasons. One of them is we defined the same epsilon value as 3 for all environments also for both unicycle and bicycle vehicles to make a comparison between environments and vehicles. Increasing epsilon value can increase the feasibility rate. The second reason can be the primitive generation. While we are generating primitives, we only defined one short primitive distance and one long primitive distance (according to kinematic constraints of the car-like vehicle) inside the lattice. If we increase the motion primitive's length, the planner can touch more points in the lattice. Thus, we can improve the feasibility rate of the planner. The other reason can be primitives search angles. While we are creating primitives, we select one up and one down orientations from the initial orientation. Increasing primitives for different orientations can increase search orientation limits for tough positions. Thus, the feasibility rate of planners can increase. The final reason can be primitive angles. We used 16 angles to generate primitives. Raising it to 32 or more can increase search points in the lattice. Then, the feasibility rate can increase. These all results also can be affected by other performance metrics. To obtain a better outcome, these changes can be made for primitives for future work.

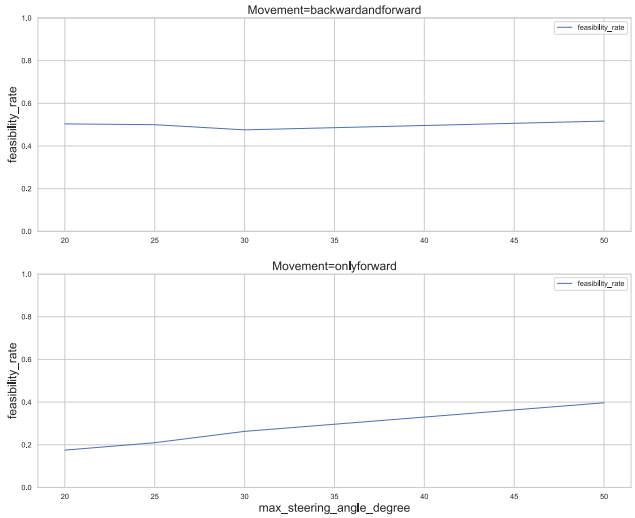


Figure 8.17: SBPL global planner bicycle vehicle feasibility rate metric results for maximum steering angle with movement

8.2.5 Global Planners Comparison for Unicycle Vehicles

In this section our results are about the comparison between all global planners for unicycle vehicles. We combined all run results for the unicycle vehicle, and we made comments according to various types of environments.

In general comparison according to our run result we obtained Figure 8.18. Dijkstra is the slowest one and RRTConnect is the fastest one in our results. Moreover, Dijkstra, all AD* planners, RRT*, PRM* have a good performance in planning length for all collected environment.

To understand which planner is the best according to the performance of environment benchmark runs, we decide to calculate the objective function. We defined objective function for high feasibility rate and low normalized planning length and time. Thus, we wrote formula in Equation 8.1, to maximize feasibility rate with given $w_1 = 1$ weight and to minimize both normalized planning length and normalized planning time with given $w_2 = w_3 = 1$ weights. We specified equal weights for our performance values, but one can define different weights according to desires. To illustrate, if someone prefers high feasibility rate, s/he can increase its weights. Moreover, if someone does not care about feasibility rate but cares about high performance in planning time, s/he can decrease feasibility rate weight and increase normalized planning time weight. Different combinations can be

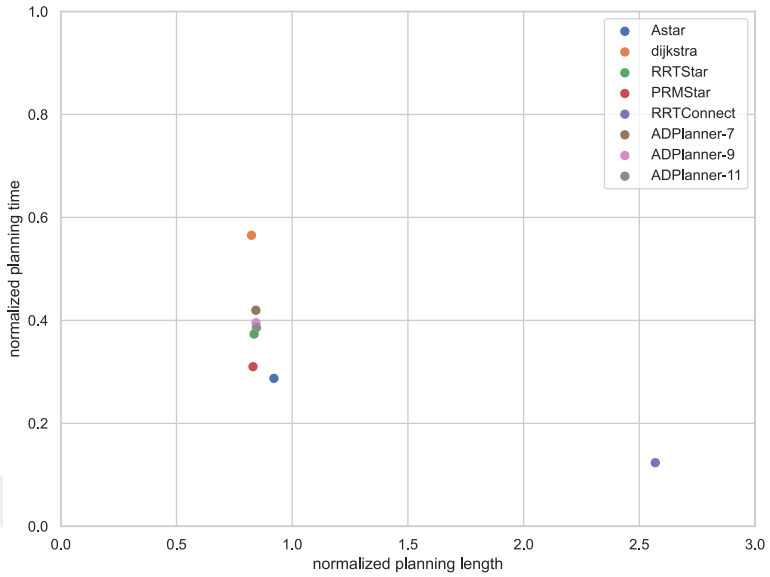


Figure 8.18: All unicycle based global planners performance metric results for mean across all environments.

applied in weights.

$$w_1(FeasibilityRate) - [w_2(NormalizedPlanningTime) + w_3(NormalizedPlanningLength)] \quad (8.1)$$

According to the defined objective function equation, we created a table from planners' performances for all environments. In Table 8.3, the top rows show the best planner and its configurations for environments. According to results, A* has good performance in complex (Intel and Fr079) or quite big (Mexico) environments. AD* also has good performance in Intel, Fr079, and Mexico environments with all motion primitives configurations. Additionally, we observed all search-based algorithms have better results than all sampling-based algorithms in large or complex environments. For Office_b environment, we saw that AD* performed best with 11 primitives per angle configuration. Moreover, RRT* and PRM* performed considerably well in Office_b environment. For a small environment (Airlab), sampling-based global planners show one of the worst performances comparing with search-based global planners. AD* planner with fewer primitives gets better results, and besides, A* planner gets second order for high performance. When we investigated the 7A-2 environment that is the wide-open one, we can see OMPL global planners on top lines. PRM* is the best and RRT* is

the third one. On the other hand, the AD* planner performed well for 11 and 9 primitives per angle configurations.

environment_name	global_planner_name	planner_type	lethal_cost	primitives_per_angle	npt	npl	fr	objectiveFunc
office_b	SBPLLatticePlanner	ADPlanner	253.0	11.0	0.558384	0.815530	1.000	-0.373915
	OmplGlobalPlanner	RRTStar	253.0	-	0.455720	0.819292	0.884	-0.391012
		PRMStar	253.0	-	0.639068	0.857952	1.000	-0.497020
	SBPLLatticePlanner	ADPlanner	253.0	9.0	0.934797	0.810473	1.000	-0.745270
	ROSGlobalPlanner	dijkstra	253.0	-	0.999420	0.800263	1.000	-0.799683
	SBPLLatticePlanner	ADPlanner	253.0	7.0	0.998629	0.813341	1.000	-0.811970
	ROSGlobalPlanner	Astar	253.0	-	1.292378	0.879323	1.000	-1.171701
	OmplGlobalPlanner	RRTConnect	253.0	-	0.424127	2.101743	1.000	-1.525869
airlab	SBPLLatticePlanner	ADPlanner	253.0	7.0	0.165982	0.893349	1.0	-0.059331
	ROSGlobalPlanner	Astar	253.0	-	0.189595	0.895510	1.0	-0.085106
	SBPLLatticePlanner	ADPlanner	253.0	9.0	0.207573	0.902005	1.0	-0.109578
				11.0	0.444291	0.880097	1.0	-0.324388
	ROSGlobalPlanner	dijkstra	253.0	-	0.682712	0.854536	1.0	-0.537248
	OmplGlobalPlanner	PRMStar	253.0	-	0.806578	0.869991	1.0	-0.676569
		RRTStar	253.0	-	0.997420	0.863375	1.0	-0.860795
		RRTConnect	253.0	-	0.154521	2.051244	1.0	-1.205765
7A-2	OmplGlobalPlanner	PRMStar	253.0	-	0.181374	0.856977	1.000000	-0.038351
	SBPLLatticePlanner	ADPlanner	253.0	11.0	0.384408	0.854938	1.000000	-0.239346
	OmplGlobalPlanner	RRTStar	253.0	-	0.227871	0.851406	0.822086	-0.257191
	SBPLLatticePlanner	ADPlanner	253.0	9.0	0.541225	0.854294	1.000000	-0.395519
	ROSGlobalPlanner	Astar	253.0	-	0.610969	0.927059	1.000000	-0.538028
		dijkstra	253.0	-	0.819753	0.832752	1.000000	-0.652505
	OmplGlobalPlanner	RRTConnect	253.0	-	0.104087	1.593314	1.000000	-0.697401
	SBPLLatticePlanner	ADPlanner	253.0	7.0	0.907194	0.858885	1.000000	-0.766079
intel	ROSGlobalPlanner	Astar	253.0	-	0.227696	0.890896	1.000000	-0.118592
	SBPLLatticePlanner	ADPlanner	253.0	11.0	0.335238	0.827100	1.000000	-0.162338
				7.0	0.351491	0.829931	1.000000	-0.181423
				9.0	0.404874	0.828518	1.000000	-0.233392
	OmplGlobalPlanner	PRMStar	253.0	-	0.318013	0.824497	0.702899	-0.439611
	ROSGlobalPlanner	dijkstra	253.0	-	0.907659	0.816243	1.000000	-0.723902
	OmplGlobalPlanner	RRTStar	253.0	-	0.572238	0.896253	0.318116	-1.150375
		RRTConnect	253.0	-	0.148122	3.573262	0.746377	-2.975007
fr079	SBPLLatticePlanner	ADPlanner	253.0	9.0	0.308109	0.894293	1.000000	-0.202402
	ROSGlobalPlanner	Astar	253.0	-	0.249393	1.003629	1.000000	-0.253022
	SBPLLatticePlanner	ADPlanner	253.0	7.0	0.371134	0.892594	1.000000	-0.263728
				11.0	0.432938	0.896248	1.000000	-0.329185
	OmplGlobalPlanner	PRMStar	253.0	-	0.431135	0.928758	1.000000	-0.359893
	ROSGlobalPlanner	dijkstra	253.0	-	0.523344	0.868551	1.000000	-0.391895
	OmplGlobalPlanner	RRTStar	253.0	-	0.547401	0.922064	0.641628	-0.827838
		RRTConnect	253.0	-	0.105988	1.808229	1.000000	-0.914218
mexico	ROSGlobalPlanner	Astar	253.0	-	0.131780	0.908038	1.000000	-0.039818
	SBPLLatticePlanner	ADPlanner	253.0	7.0	0.250709	0.823605	0.969035	-0.105279
	ROSGlobalPlanner	dijkstra	253.0	-	0.296700	0.809764	1.000000	-0.106465
	SBPLLatticePlanner	ADPlanner	253.0	9.0	0.274071	0.824700	0.969207	-0.129563
				11.0	0.336943	0.825789	0.968694	-0.194038
	OmplGlobalPlanner	PRMStar	253.0	-	0.152425	0.749287	0.541037	-0.360675
		RRTStar	253.0	-	0.193703	0.760444	0.392000	-0.562148
		RRTConnect	253.0	-	0.043920	2.949951	0.540444	-2.453427

Table 8.3: Objective function table with performance rates for all planners.

8.2.6 SBPL Global Planners Comparison Between Unicycle and Bicycle Vehicles

This last comparison is about vehicles. We compared differential drive vehicles with Ackerman steering vehicles for various environments. As known, we used unicycle kinematic model for differential drive vehicle and bicycle model for Ackerman steering vehicle. For comparing we also used Equation 8.1 as an objective function. As previously mentioned, we did not use weight

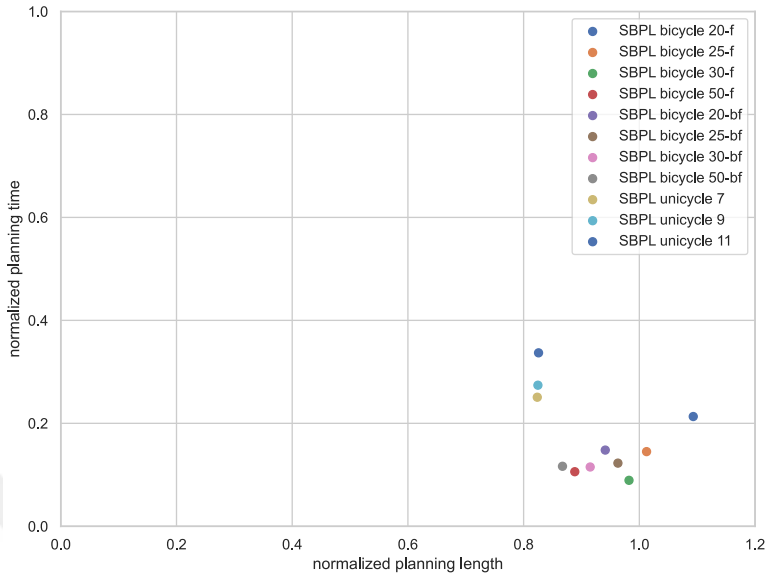


Figure 8.19: SBPL global planners unicycle vs bicycle performance metric results for mean across all environments.

for the objective function, but according to desires, weights can be changed, and new results can be obtained.

In Figure 8.19, we gave all SBPL planners performance results for collected all environments. According to our run result, we saw unicycle vehicle finds shorter paths than bicycle ones. The reason of this is that bicycle vehicles make maneuvers to find the right path. These maneuvers increase taking the path. On the other hand, we realize bicycle vehicles find faster paths comparing to unicycle ones. We made further comments according to our observation for these results. Bicycle vehicles with given our motion primitives cannot solve tough planning problems such as if given goal position close to a wall with hard orientation planner is aborting these type of tough plans because planner cannot find feasible path for relaying on vehicle kinematic constraints. As a result of it, bicycle vehicle planner finds more easy and short paths so, its planning time decreasing comparing with unicycle one. We gave more details in Section 8.2.4 bicycle motion primitives effects on the results and probable solution methods such as increasing epsilon value or adding extra primitives.

When we compared unicycle and bicycle vehicles for different environments, we obtained Table 8.4. We can mention that unicycle vehicle has better performance almost all environments. As remarked before, making path planning with a unicycle vehicle is easier than finding a plan for a

bicycle vehicle. Unicycle vehicle has rotated in place movement comparing with steering geometry it has a significant advantage, and we can see its high performance in objective function results from the Table 8.4.

In conclusion, if someone has similar environments like ours, s/he can choose a differential drive vehicle for good performance. If s/he prefers the Ackerman steering vehicle for better performance, s/he can select backward and forward movement vehicles with a high maximum steering angle. Thus, the vehicle can make better maneuvers in the environment.

environment_name	global_planner_name	direction	max_steering_angle	primitives_per_angle	kinematic_model	npt	npl	fr	objectiveFunc
office_b	ADPlanner			11.0	unicycle	0.558384	0.815530	1.000000	-0.373915
		backwardandforward	50.0		bicycle	0.064476	0.854333	0.528340	-0.390469
			20.0		bicycle	0.083778	1.182686	0.544534	-0.721929
				9.0	unicycle	0.934797	0.810473	1.000000	-0.745270
				7.0	unicycle	0.998629	0.813341	1.000000	-0.811970
		backwardandforward	30.0		bicycle	0.085507	1.208057	0.461538	-0.832025
			25.0		bicycle	0.158898	1.756068	0.513185	-1.401782
		onlyforward	50.0		bicycle	0.163411	2.987606	0.455466	-2.695551
			30.0		bicycle	0.249000	3.810805	0.255061	-3.804744
			20.0		bicycle	0.550938	11.039777	0.089069	-11.501646
			25.0		bicycle	1.831192	35.598285	0.153846	-37.275631
fr079	ADPlanner			9.0	unicycle	0.308109	0.894293	1.000000	-0.202402
				7.0	unicycle	0.371134	0.892594	1.000000	-0.263728
				11.0	unicycle	0.432938	0.896248	1.000000	-0.329185
		backwardandforward	50.0		bicycle	0.090844	1.155254	0.526596	-0.719503
			30.0		bicycle	0.051486	1.451647	0.443262	-1.059870
			25.0		bicycle	0.071584	1.490599	0.485816	-1.076368
		onlyforward	50.0		bicycle	0.075751	1.868987	0.297872	-1.646866
			25.0		bicycle	0.030222	2.613465	0.058511	-2.585176
			20.0		bicycle	0.019991	2.625314	0.026596	-2.618709
		backwardandforward	20.0		bicycle	0.116600	3.363840	0.489362	-2.991078
		onlyforward	30.0		bicycle	0.077373	5.058093	0.132979	-5.002488
7A-2	ADPlanner			11.0	unicycle	0.384408	0.854938	1.000000	-0.239346
				9.0	unicycle	0.541225	0.854294	1.000000	-0.395519
		backwardandforward	50.0		bicycle	0.159221	0.915983	0.523148	-0.552056
			30.0		bicycle	0.095605	0.933667	0.462963	-0.566309
		onlyforward	50.0		bicycle	0.123649	0.942468	0.497685	-0.568432
		backwardandforward	25.0		bicycle	0.190037	1.017016	0.537037	-0.670016
			20.0		bicycle	0.148926	1.081050	0.509259	-0.720716
				7.0	unicycle	0.907194	0.858885	1.000000	-0.766079
		onlyforward	30.0		bicycle	0.093420	1.281682	0.391204	-0.983899
			20.0		bicycle	0.102898	1.185616	0.254630	-1.033884
			25.0		bicycle	0.157409	1.371318	0.298611	-1.230115
intel	ADPlanner			11.0	unicycle	0.335238	0.827100	1.000000	-0.162338
				7.0	unicycle	0.351491	0.829931	1.000000	-0.181423
				9.0	unicycle	0.404874	0.828518	1.000000	-0.233392
		backwardandforward	50.0		bicycle	0.080940	0.979643	0.560764	-0.499819
			25.0		bicycle	0.053823	1.221420	0.463542	-0.811702
			30.0		bicycle	0.052426	1.282814	0.496528	-0.838713
			20.0		bicycle	0.093175	1.467533	0.500000	-1.060708
		onlyforward	50.0		bicycle	0.074361	1.791611	0.347222	-1.518750
			25.0		bicycle	0.061710	3.657339	0.069444	-3.649604
			20.0		bicycle	0.031830	3.731315	0.020833	-3.742312
			30.0		bicycle	0.143983	8.170904	0.085069	-8.229818
airlab	ADPlanner			7.0	unicycle	0.165982	0.893349	1.000000	-0.059331
				9.0	unicycle	0.207573	0.902005	1.000000	-0.109578
				11.0	unicycle	0.444291	0.880097	1.000000	-0.324388
		onlyforward	20.0		bicycle	0.009048	0.857740	0.037037	-0.829750
		backwardandforward	50.0		bicycle	0.043460	1.453015	0.592593	-0.903882
			30.0		bicycle	0.059852	1.915313	0.296296	-1.678869
			20.0		bicycle	0.053866	2.216651	0.518519	-1.751999
		onlyforward	25.0		bicycle	0.016471	2.049049	0.074074	-1.991445
		backwardandforward	25.0		bicycle	0.081407	2.903255	0.481481	-2.503180
		onlyforward	30.0		bicycle	0.052743	2.987968	0.037037	-3.003674
			50.0		bicycle	0.119113	10.302022	0.370370	-10.050764
mexico	ADPlanner			7.0	unicycle	0.250709	0.823605	0.969035	-0.105279
				9.0	unicycle	0.274071	0.824700	0.969207	-0.129563
				11.0	unicycle	0.336943	0.825789	0.968694	-0.194038
		backwardandforward	50.0		bicycle	0.116572	0.867171	0.485651	-0.498093
			30.0		bicycle	0.115187	0.915064	0.495319	-0.534932
			25.0		bicycle	0.122712	0.962969	0.505221	-0.580460
		onlyforward	50.0		bicycle	0.106028	0.888430	0.403814	-0.590644
		backwardandforward	20.0		bicycle	0.148053	0.941120	0.491130	-0.598044
		onlyforward	30.0		bicycle	0.089306	0.982153	0.321076	-0.750383
			25.0		bicycle	0.145059	1.012508	0.307314	-0.850253
			20.0		bicycle	0.213230	1.093198	0.289082	-1.017346

Table 8.4: Objective function table with performance rates for SBPL planners.

Chapter 9

Conclusion and Future Work

In this work, benchmarking of global planners is made in Robot Operating System (ROS). Benchmarking script is written for different planners. Environment metrics and performance metrics are defined for comparisons and results. Various environments are chosen to see the performance of planners for differential drive and Ackerman steering vehicles.

Search-based and sampling-based planners are used to solve planning problems. ROS Global Planner package is used for search-based planners that include Dijkstra and A* algorithms. Also, the SBPL package is used for the AD* algorithm. Different types of primitives are generated for unicycle and bicycle vehicles. Already prepared motion primitive generation library is used to generate motion primitives for unicycle vehicles. On the other hand, we wrote our motion primitive generation MatLab script according to bicycle vehicle kinematic constraints. Finally, benchmarking is run for defined configuration, and comments are made about results.

To compare the performance of the planners, a series of planning queries are defined for six different environments. The results are divided into categories. The first category is ROS Global Planner compassion. A* planner is seen faster than Dijkstra in contrast, Dijkstra makes shorter paths than A* for all environments. The second category is OMPL global planners comparisons. RRTConnect is seen as the fastest algorithm, but it finds the longest paths. On the other hand, PRM* and RRT* act differently for each environment, but mostly they find the shortest paths. In the third category, SBPL AD* planner is examined for unicycle model vehicles. When the motion primitives increased, planning time decrease is seen. Since AD* is an optimal planner, planning lengths are found slightly similar for all configurations. In the fourth category, SBPL AD* planner is examined

for bicycle model vehicles with both backward and forward movement with various maximum steering angles. Mostly, backward and forward moves give better time and length solutions than the only forward movement for planning problems. Moreover, when the maximum turning angle increased, maneuverability also increases, and planning length decreases are seen. But, the feasibility rate is seen low for all environments, and possible solutions are given in the thesis. In the fifth category, all global planners are compared for unicycle model vehicles given six different environments by using a defined objective function. Results show that for different environments, different planners have better performance. For the final category, SBPL global planner is compared for both unicycle and bicycle model vehicles in different environments. Mostly, unicycle model vehicle gives better objective function with planning length than bicycle model vehicle.

In conclusion, this thesis gives comparisons for different planners with unicycle and bicycle model vehicles in various type of environments.

For future work, bicycle model motion primitives can be improved, and its feasibility rate can be increased. Benchmarking runs can be tried in many different environments, or benchmark run numbers can be increased for more results. Thus, comments become more reliable. Moreover, some dynamic environments can be used for different purposes or scenarios. Various types of vehicles can be used to increase the perspective of benchmarking. Finally, another type of performance metric can be defined for diverse desires.

Bibliography

[1] About ros
`https://www.ros.org/about-ros/` (accessed: 20.02.2021).

[2] Agilex
`https://www.agilex.ai/?lang=en-us` (accessed: 10.02.2021).

[3] Navigation
`http://wiki.ros.org/navigation` (accessed: 20.02.2021).

[4] Open motion planning library
`https://ompl.kavrakilab.org/` (accessed: 13.02.2021).

[5] Robotis e-manual
`https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#features` (accessed: 10.02.2021).

[6] Search-based planning library
`http://sbpl.net/Home` (accessed: 15.02.2021).

[7] Open motion planning library: A primer
`http://ompl.kavrakilab.org`
kavraki lab rice university, 2020.

[8] J. Baltes. A benchmark suite for mobile robots. *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*.

[9] Enrico Bertolazzi and Marco Frego. G1 fitting with clothoids. *Mathematical Methods in the Applied Sciences*, 38, 03 2014.

[10] Enrico Bertolazzi and Marco Frego. On the g 2 hermite interpolation problem with clothoids. *Journal of Computational and Applied Mathematics*, 341:99–116, 10 2018.

[11] Daniele Calisi, Luca Iocchi, and Daniele Nardi. A unified benchmark framework for autonomous mobile robots and vehicles motion algorithms (movema benchmarks). 01 2008.

[12] Benjamin Cohen, Ioan A. Sucan, and Sachin Chitta. A generic infrastructure for benchmarking motion planners. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[14] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A guide to heuristic-based path planning. 01 2005.

[15] Santiago Garrido, Mohamed Abderrahim, and Luis Moreno. Path planning and navigation using voronoi diagram and fast marching. *IFAC Proceedings Volumes*, 39(15):346–351, 2006. 8th IFAC Symposium on Robot Control.

[16] Santiago Garrido and Luis Moreno. *Mobile Robot Path Planning using Voronoi Diagram and Fast Marching*. 05 2015.

[17] Martin Guenther. Sbpl_lattice_planner `https://wiki.ros.org/sbpl_lattice_planner`, 2018.

[18] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *CoRR*, abs/1110.2737, 2011.

[19] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[20] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[21] Lydia Kavraki, Petr Svestka, J.C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12:566 – 580, 09 1996.

[22] Desmond King-Hele. Erasmus darwin's improved design for steering carriages–and cars. *Notes and Records of the Royal Society of London*, 56(1):41–62, 2002.

[23] Sven Koenig and Maxim Likhachev. D*lite. In *Eighteenth National Conference on Artificial Intelligence*, page 476–483, USA, 2002. American Association for Artificial Intelligence.

[24] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 995–1001 vol.2, 2000.

[25] Steven LaValle and James Kuffner. Randomized kinodynamic planning. *I. J. Robotic Res.*, 20:378–400, 01 2001.

[26] Steven Michael. LaValle. *Planning algorithms*. Cambridge University Press, 2014.

[27] Maxim Likhachev and Dave Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.

[28] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613–1643, 2008.

[29] Maxim Likhachev, David Ferguson, Geoffrey Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. pages 262–271, 01 2005.

[30] Maxim Likhachev, Geoffrey Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. volume 16, 01 2003.

[31] Darby Lim. turtlebot3 `http://wiki.ros.org/turtlebot3` (accessed: 10.02.2021), 2018.

[32] Ms. Puam Marbate and Ms. Prachi Jaini. Role of voronoi diagram approach in path planning. *International Journal of Engineering Science and Technology (IJEST)*, 5(3):527–532, 2013.

[33] Kurt Mehlhorn and Peter Sanders. *"Chapter 10. Shortest Paths" (PDF). Algorithms and Data Structures: The Basic Toolbox.*, pages 191–215. 2008.

[34] Seda Milos and Václav Pich. Robot motion planning using generalised voronoi diagrams. pages 215–220, 08 2008.

[35] Mark Moll, Ioan A. Sucan, and Lydia E. Kavraki. Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. *IEEE Robotics Automation Magazine*, 22(3):96–102, 2015.

[36] Walter Nowak, Alexey Zakharov, Sebastian Blumenthal, and Erwin Prassler. Benchmarks for mobile manipulation and robust obstacle avoidance and navigation. 05 2010.

[37] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, 2016.

[38] M. Pivtoraiko and A. Kelly. Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3231–3237, 2005.

[39] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. *ROS Robot Programming*. ROBOTIS Co.,Ltd., 2017.

[40] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[41] Jacob T Schwartz and Micha Sharir. On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, 4(3):298–351, 1983.

[42] Tristan Schwörer. Global planner `http://wiki.ros.org/global_planner` (accessed: 12.02.2021).

[43] Petru Simionescu. *Named Contributions to MMS: Bridging History and Terminology*, pages 1141–1150. 06 2019.

[44] Christoph Sprunk, Jörg Röwekämper, Gershon Parent, Luciano Spinello, Gian Diego Tipaldi, Wolfram Burgard, and Mihai Jalobeanu. An experimental protocol for benchmarking robotic indoor navigation. *Experimental Robotics Springer Tracts in Advanced Robotics*, page 487–504, 2015.

[45] Anthony Stentz. The d* algorithm for real-time planning of optimal traverses. 04 2011.

[46] Ioan A. Sucan, Mark Moll, and Lydia E. Kavraki. The open motion planning library. *IEEE Robotics amp; Automation Magazine*, 19(4):72–82, 2012.

[47] Jian Wen, Xuebo Zhang, Qingchen Bi, Zhangchao Pan, Yanghe Feng, Jing Yuan, and Yongchun Fang. Mrpb 1.0: A unified benchmark for the evaluation of mobile robot local planning approaches, 2020.

[48] Kaiyu Zheng. Ros navigation tuning guide `http://kaiyuzheng.me/documents/navguide.pdf`, 2016.

Appendix A

User Manual

A.1 How to Run

In this work, we used Ubuntu 18.04 with ROS Melodic.

Step 1) Create ROS work-space (note: ROS Melodic version required)

```
$ mkdir -p ~/w/catkin_ws/src
$ cd ~/w/catkin_ws
$ catkin build -j 6
```

Step 2) Download environment from github (note: one must run decompress_dataset_files.py file from performance_modelling/performance_modelling_py/environment or you can just decompress only related environments compressed files (we are using airlab, 7A-2, office_b, mexico, fr079 and intel environments))

```
$ mkdir -p ~/ds/performance_modelling
$ git clone https://github.com/AIRLab-POLIMI/performance_modelling_
test_datasets.git
$ mv ~/ds/../performance_modelling_test_datasets ~/ds/../test_datasets
```

Step 3) Install related ROS packages

```
$ sudo apt-get install ros-melodic-catkin \
    ros-melodic-map-server \
    ros-melodic-move-base \
    ros-melodic-global-planner \
    ros-melodic-dwa-local-planner \
    ros-melodic-filters \
```

```
    ros-melodic-sbpl \
    ros-melodic-sbpl-dbgsym \
    ros-melodic-sbpl-recovery \
    ros-melodic-sbpl-recovery-dbgsym \
    ros-melodic-ompl \
    ros-melodic-ompl-dbgsym \
    ros-melodic-turtlebot3 \
    ros-melodic-tf-conversions \
    ros-melodic-tf2-sensor-msgs
```

Step 4) Setup SBPL package

```
$ git clone https://github.com/sbpl/sbpl.git
$ cd sbpl
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Step 5) Download codes from github

```
$ cd ~/w/catkin_ws/src
$ git clone --branch=master https://github.com/AIRLab-POLIMI/
performance_modelling.git
$ git clone --branch=melodic-devel https://github.com/fcelitok/
global_planning_performance_modelling.git
$ git clone --branch=master https://github.com/fcelitok/
move_base_ompl.git
$ git clone --branch=melodic-devel https://github.com/ fcelitok/
navigation_experimental.git
$ cd ~/w/catkin_ws/
$ catkin build
```

Step 6) Install all dependencies

```
$ cd ~/w/catkin_ws/src/performance_modelling/
$ ./install_dev.sh
$ ./install_dependencies.sh
$ cd ~/w/catkin_ws/src/global_planning_performance_modelling/
$ ./install_dependencies.sh
```

(this global_planning_performance_modelling package include very general run. If one want to run only one map or for specific configurations one must follow down part)

Step 7) Change configuration according to specific primitives or runs
* open execute_grid_benchmark.py

```
$ cd ~/w/catkin_ws/src/global_planning_performance_modelling/
src/global_planning_performance_modelling_ros
```

-change line 20:
default="~/ds/performance_modelling/test_datasets/dataset/(environment name)"
# (environment name) ⟶ airlab or 7A-2...
-change line 27:
(reads configuration file so one must change also configuration file from in same path according to desire)
default="~/w/catkin_ws/src/global_planning_performance_modelling/config/benchmark_configurations/global_planning_grid_benchmark_config.yaml"
* for changing configuration yaml file.

```
cd ~/w/catkin_ws/src/global_planning_performance_modelling/config/
benchmark_configurations
```

Then change related configurations from /global_planning_grid_benchmark_(xxx).yaml

Step 8) Run

```
$ rosrun global_planning_performance_modelling execute_grid_benchmark.py
--no-shuffle --ignore-previous-runs --gui
```