Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Space-filling trees in sampling-based motion planning

Bc. Jaroslav Janoš

Supervisor: Ing. Vojtěch Vonásek, Ph. D.
Field of study: Open Informatics
Subfield: Computer Engineering
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Janoš Jaroslav**	Personal ID number:	**474435**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Measurement**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Space-filling trees in sampling-based motion planning

Master’s thesis title in Czech:

Vzorkovací techniky plánování pohybu

Guidelines:

1. Get familiar with motion planning [1], namely with sampling-based techniques (e.g., RRT, PRM, EST) for single-goal motion planning. Get familiar with basic extensions to motion planning over multiple goals [2,5,6].
2. Extend the method from [6] by considering Dubins Maneuvers [3,4] for 2D and 3D case.
3. Further extend the method [6] by polynomial maneuvers for robots in scenarios with a single goal. Consider primarily 2D environments in this task, optionally also 3D case.
4. Design 2D and 3D testing scenarios. Evaluate and compare the methods 2) and 3) with suitable algorithms from the OMPL library [7].

Bibliography / sources:

[1] LaValle, Steven M. Planning algorithms. Cambridge university press, 2006.
[2] S. N. Spitz and A. A. G. Requicha. Multiple-goals path planning for coordinate measuring machines. In IEEE International Conference on Robotics and Automation, volume 3, pages 2322–2327 vol.3, 2000.
[3] R. Pěnička, J. Faigl, P. Váňa, and M. Saska. Dubins orienteering problem. IEEE Robotics and Automation Letters, 2(2):1210–1217, 2017.
[4] J. Ny, E. Feron, and E. Frazzoli. On the dubins traveling salesman problem. IEEE Transactions on Automatic Control, 57(1):265–270, 2012
[5] D. Devaurs, T. Siméon, and J. Cortés. A multi-tree extension of the transition-based RRT: Application to ordering-and-pathfinding problems in continuous cost spaces. In IEEE/RSJ IROS, 2014.
[6] J. Janoš, V. Vonásek and R. Pěnička, "Multi-Goal Path Planning Using Multiple Random Trees," in IEEE Robotics and Automation Letters, vol. 6, no. 2, pp. 4201-4208, April 2021, doi: 10.1109/LRA.2021.3068679.
[7] I. A. Sucan, M. Moll and L. E. Kavraki, "The Open Motion Planning Library," in IEEE Robotics & Automation Magazine, vol. 19, no. 4, pp. 72-82, Dec. 2012, doi: 10.1109/MRA.2012.2205651.

Name and workplace of master’s thesis supervisor:

Ing. Vojtěch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **31.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

Ing. Vojtěch Vonásek, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to express my deepest gratitude to my advisor, Ing. Vojtěch Vonásek, Ph. D., for all materials he provided me, as well as advices and leadership throughout my whole research of space filling forest, lasting almost 2 years, and writing the thesis.

Special thanks also go to my second advisor, Ing. Robert Pěnička, Ph. D., especially for his advices about the state-of-art planning on polynomial trajectories and help with the research.

Last but not least, I would like to thank my family and my friends for their mental support not only during this research, but also thoughout my whole studies.

Computational resources were supplied by the project “e-Infrastruktura CZ” (e-INFRA CZ LM2018140) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20. May 2022

Abstract

Space Filling Forests are a family of sampling-based multi-goal motion planning algorithms. The first objective of this work is to extend it for single-goal planning in Euclidean spaces, for Dubins car and Dubins airplane, and for planning with polynomial trajectories. The second challenge is to propose a novel multi-goal path planner for Dubins vehicles. First, existing algorithms for motion planning were analyzed and compared with Space Filling Forest, and approaches for multi-goal path planning were studied. Subsequently, the proposed algorithms were implemented in Open Motion Planning Library and in a custom library of planning algorithms. These implementations were then compared with existing sampling-based algorithms. The benchmarks showed, that the Space Filling Forest algorithms for single-goal planning are suitable for environments with narrow passages, where they outperform classic planners, but they are not so successful in spacious high-dimensional spaces. Similar results were achieved with the new multi-goal algorithm, which found mediocre tours in mediocre times, but was able to link hard-to-connect cities. Space Filling Forests might therefore be an interesting alternative to classic planners.

Keywords: motion planning, sampling-based motion planners, Space Filling Forest, Open Motion Planning Library, robotics, Dubins vehicle, polynomial trajectories

Supervisor:
Ing. Vojtěch Vonásek, Ph. D.

Abstrakt

Tato práce se zabývá rodinou algoritmů “Space Filling Forest” navrženou pro plánování pohybu mezi více cíli. Jedním ze záměrů práce je rozšířit tuto rodinu o plánování pro jeden cíl v Euklidovských prostorech, pro Dubinsovo auto i Dubinsův letoun, a pro plánování na polynomiálních trajektoriích. Dalším záměrem je pak navrhnout novou metodu pro plánování pohybu mezi více cíli pro Dubinsova vozidla. Algoritmy Space Filling Forest byly nejprve analyzovány a porovnány s jinými algoritmy pro plánování pohybu. Zároveň byly studovány postupy pro plánování cest mezi více cíli. Následně byly navržené metody implementovány v knihovně “Open Motion Planning Library” i ve vlastní knihovně plánovacích algoritmů. Obě implementace pak byly porovnány s ostatními plánovači. Tyto testy ukázaly, že nově navržené algoritmy plánování pro jeden cíl jsou vhodné v prostředích s úzkými průchody, kde překonaly klasické plánovače, ale nejsou příliš úspěšné ve velkých vícedimenzionálních prostorech. Podobných výsledků dosáhl i navržený plánovač cest mezi více cíli – nalezené cesty byly délkově a časově průměrné ale umožnil propojení těžce přístupných cílů. Navržené algoritmy tak mohou být zajímavou alternativou k existujícím plánovačům.

Klíčová slova: plánování pohybu, vzorkovací plánovače pohybu, Space Filling Forest, Open Motion Planning Library, robotika, Dubinsovo vozidlo, polynomiální trajektorie

Překlad názvu: Vzorkovací techniky plánování pohybu

Contents

Figures

Tables

Chapter 1

Introduction

In the recent years, new technology and robots have started to significantly step into our lifes. However, none of it would be possible without solving one of the crucial problems of robotics: how to transform high-level specification of human motion tasks to a sequence of simple instructions how to move. This is what algorithms for “motion planning” solve.

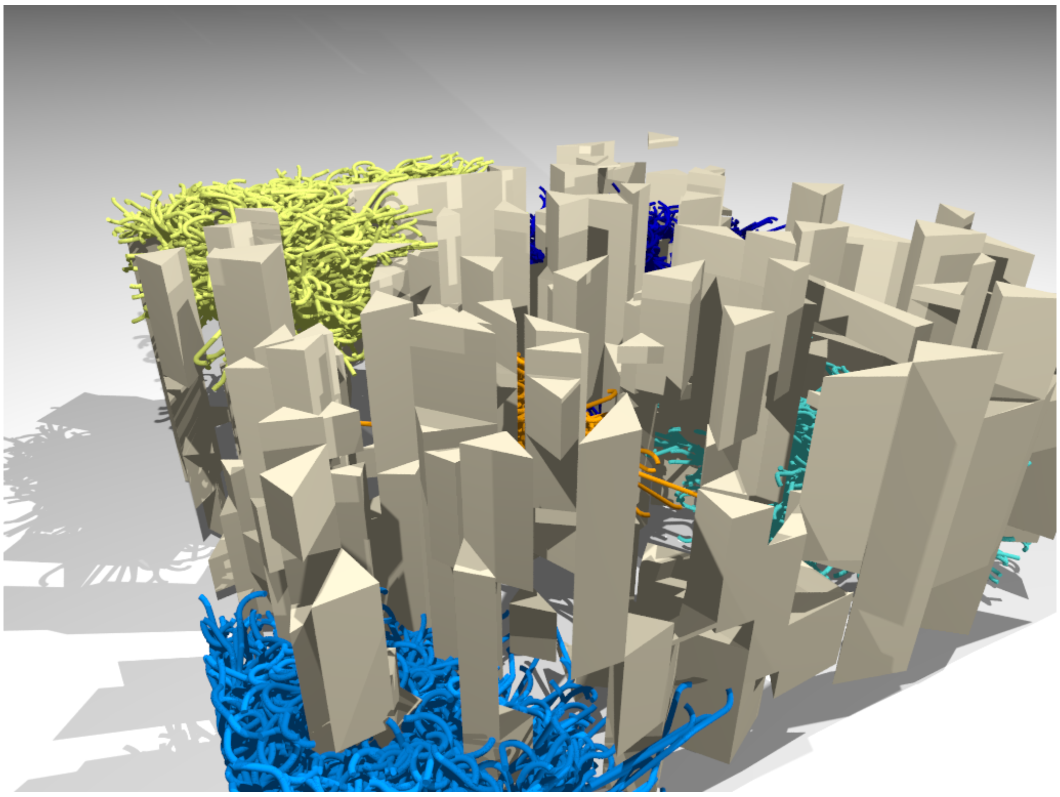


Figure 1.1: Growth of the space filling forest for multi-goal motion planning with Dubins airplane in Dense 3D environment

A basic task of motion planning consists of planning of a collision-free path or sequence of moves from a point A to another point B on a plane. However, motion planning also includes more complex tasks like planning in higher dimensions, for robots or robotic arms with many degrees of freedom, planning in uncertainty, i.e., in an unknown environment or with noisy data from sensors, or planning for robots with kinodynamic constraints, e.g., Dubins car or Dubins airplane.

Even though some basic motion planning task might be solved in polynomial time (Euclidean shortest path problem in 2D with polygonal obstacles [1]) and some of the tasks are NP-hard (Euclidean shortest path problem in polyhedral environment [2]), motion planning is generally PSPACE-complete [3]. The most difficult part of the problem is finding all possible collision-free positions of the robot in space.

This field of robotics has been the focus of researchers since the second half of the 20th century. Many motion planning algorithms have been developed during this research from those that focus on simple tasks in low-dimensional environments, to complex algorithms for robots with kinodynamic constraints in high-dimensional environments.

Early methods, such as the retraction method or visibility graphs, were based on purely geometric features of the problem. Therefore, they were especially suitable for low-dimensional problems. In 1990s sampling-based methods were introduced to cope with high-dimensional systems with many degrees of freedom. The latest published motion planning algorithms from the family of the sampling-based algorithms are "Space Filling Forest" [4] and "Space Filling Forest Star" [5], further also denoted by "SFF algorithms".

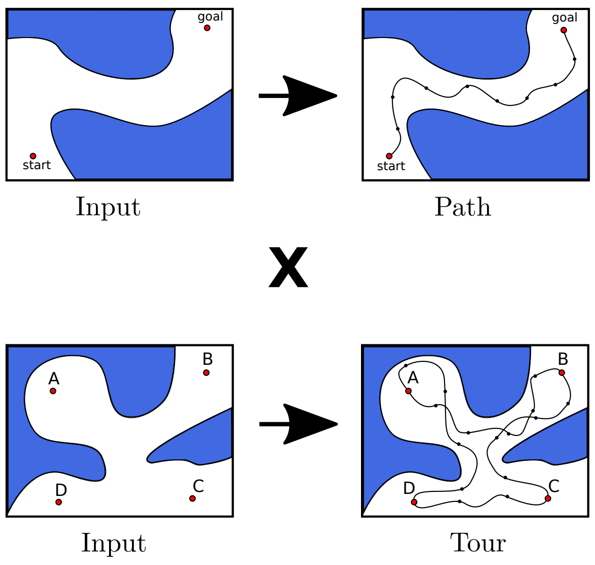


Figure 1.2: Single-goal (top) versus multi-goal (bottom) motion planning

1.1 Goals

Both algorithms are designed for "multi-goal" motion planning, i. e., planning of a tour between given points of interest. One of the main challenges of this work is the extension of these algorithms also for single-goal motion planning, i. e., for finding a path between a given start and a goal point, and compare it with existing single-goal motion planners. The difference between single-goal and multi-goal motion planning is depicted in Figure 1.2.

The classic workflow of single-goal motion planners is shown in Figure 1.3 and is as follows: a roadmap, tree, or multiple trees that contain given start and goal points are created first. Subsequently, a path connecting the start and goal points is extracted from this graph. The path may consist of straight-line segments or more complex curves, such as Dubins curves. Before deploying on a real robot, the path is usually also transformed to a trajectory using some external trajectory generator, where time and robot dynamics are also taken into account.

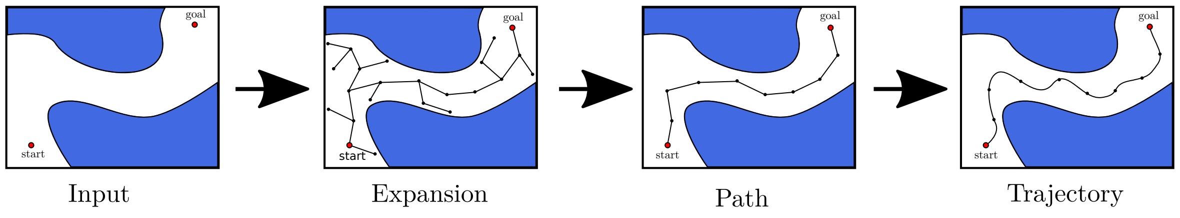


Figure 1.3: Classic workflow of the single-goal motion planners

For the SFF algorithms the same principles are applied. In addition, state-of-the-art planning on polynomial trajectories is also implemented. In this way the graph is expanded with smaller subtrajectories with respect to the time and robot dynamics of the robot, so the trajectory is obtained in the end directly. This process is illustrated in Figure 1.4.

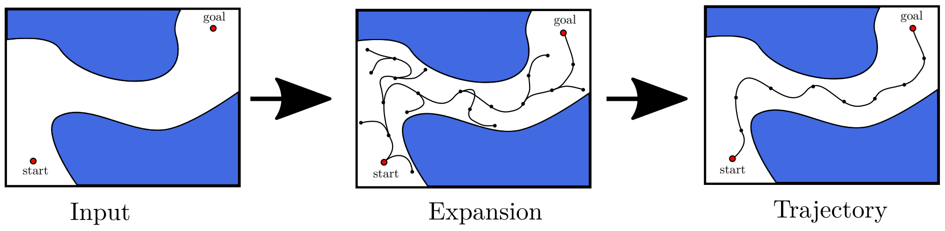
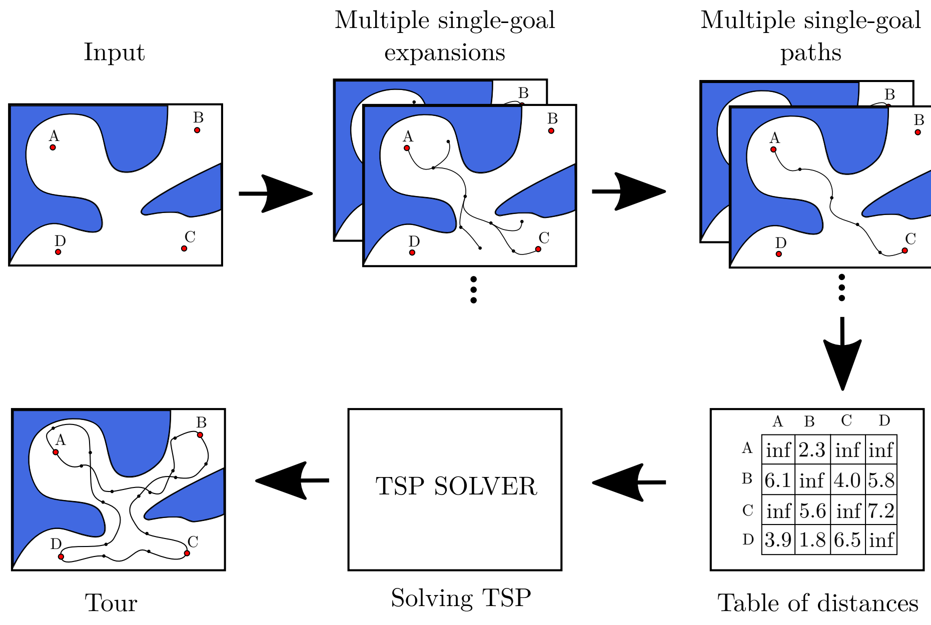


Figure 1.4: Workflow of planning on polynomial trajectories

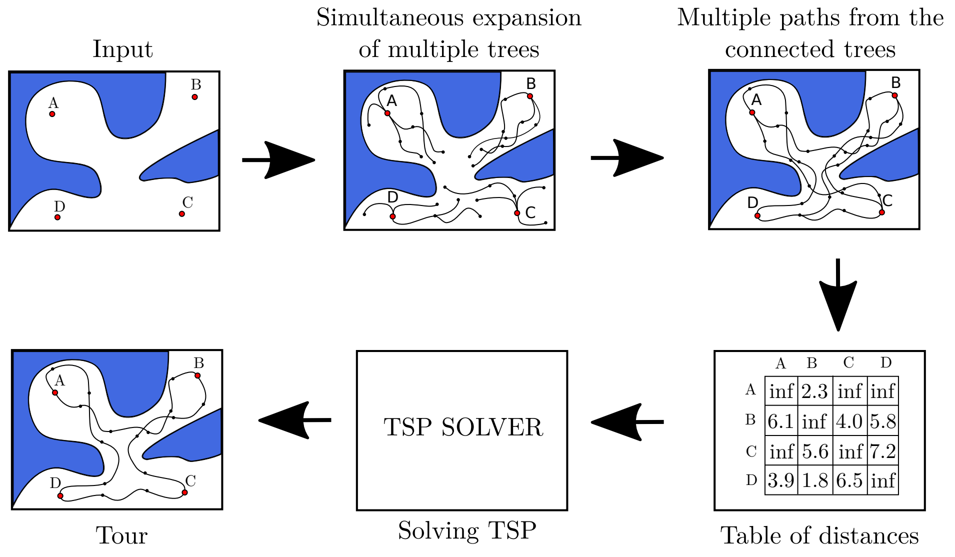
The second challenge of this work is to propose a novel multi-goal motion planning algorithm based on SFF for planning with Dubins curves. The current approach for solving this problem, shown in Figure 1.5, is to find a path for each pair of points to be visited using single-goal motion planning methods. Then the Travelling Salesman Problem is formulated as finding the shortest tour from the lenghts of found paths. After its solution the paths forming the final tour are selected according to the obtained results.



	A	B	C	D
A	inf	2.3	inf	inf
B	6.1	inf	4.0	5.8
C	inf	5.6	inf	7.2
D	3.9	1.8	6.5	inf

Figure 1.5: Classic workflow of the multi-goal motion planning for a robot with kinodynamic constraints

The proposed approach is to grow multiple trees with Dubins curves in SFF manner and extract paths from connected trees. Subsequent steps are similar to the classic approach, i. e., a Traveling Salesman Problem is formulated, solved and the final tour is contructed from the selected paths. This simplified workflow is visualized in Figure 1.6.


Input
Simultaneous expansion of multiple trees
Multiple paths from the connected trees
A
B
C
D
TSP SOLVER
Tour
Solving TSP
Table of distances

	A	B	C	D
A	inf	2.3	inf	inf
B	6.1	inf	4.0	5.8
C	inf	5.6	inf	7.2
D	3.9	1.8	6.5	inf

Figure 1.6: Proposed novel workflow of the multi-goal motion planning for a robot with kinodynamic constraints using space filling forest

1.2 Thesis structure

The organization of this thesis is as follows: Chapter 2 defines the fundamental terms related to motion planning, followed by an analysis of single-goal motion planning, its state-of-the-art algorithms and methods dealing with constrained robots in Chapter 3. In Chapter 4 introduces multi-goal motion planning methods along with a definition and an overview of the Travelling Salesman Problem and similar combinatorial problems related to multi-goal planning.

Then, in Chapter 5, the Space Filling Forest algorithm and its variant are discussed. The following Chapter 6 deals with the implementaion of the proposed algorithms both in the Open Motion Planning Library as well as in the custom library, together with an overview of techniques and libraries used.

Both implementations of SFF-based algorithms were benchmarked and compared using resources described in Chapter 7. Results of these benchmarks are analysed in detail in Chapter 8 for single-goal planners and in Chapter 9 for multi-goal planners.

Chapter 2
Motion planning

One of the most ancient motion planning problems is the "Piano Mover's Problem". Having a precise model of a house and a piano, the goal is to move the piano out of one room to another without hitting any wall or other obstacle [6]. This problem is proven to be PSPACE-complete [3].

Even though the majority of research of motion planning focuses on robotics, where the motion planning problems have been studied since the late 1960s for both robotic manipulators [7] and mobile robots [8], the scope of applications is much wider. The algorithms might be deployed e. g., for solving puzzles [6], car assembly [9] or protein research in biology [10].

Motion planning methods solve the problem only on the basis of geometry, i. e., the dynamics and time are usually not considered. Algorithms which take such limitations into account and assign path to the time domain are usually referred to as "trajectory planning" algorithms. They might be built on the solution of motion planning algorithms — trajectories are constructed on the resulting path waypoints [6, 11]. Such an approach is called "decoupled". Its main advantage is its modularity, e. g., it is possible to use different motion planners with the same trajectory generator. However, sometimes is is advantegeous to solve both problems at once, such as trajectories can produce unfeasible trajectories, as described in Section 3.4.

Unlike dynamics, prior knowledge of the shape and location of the robot as well as the spatial arrangement of the environment is essential for motion planning. Such knowledge might come directly from user or might be obtained through sensing [12].

Before stepping into particular algorithms, it is important to define basic terms as well as the motion planning problem itself.

2.1 Basic terms

Let $\mathcal{W}$ be "world" or "workspace", i. e., the operational space where the motion will be planned. Generally, $\mathcal{W} = \mathbb{R}^2$ for 2D space or $\mathcal{W} = \mathbb{R}^3$ for 3D space. As stated by La Valle [6], the world typically contains two entities, a) robots, bodies (subsets of the world) to be controlled by the plan of motions, and b) obstacles, subsets of the world that are permanently occupied.

Let $\mathcal{A}$ be a robot opearating in $\mathcal{W}$, and $\mathcal{B}_1, \ldots, \mathcal{B}_j$ obstacles. For these holds true

$$\mathcal{A} \subset \mathcal{W}, \tag{2.1}$$

$$\mathcal{B}_i \subset \mathcal{W}, \forall i \in \{1, \ldots, j\}, \tag{2.2}$$

$$\mathcal{O} = \bigcup_{\forall i \in \{1,\ldots,j\}} \mathcal{B}_i, \tag{2.3}$$

where $\mathcal{O}$ represents the "obstacle region", i. e., an union of spaces occupied by the obstacles.

Next, let q be the robot configuration. "Configuration" is a specification of position and orientation of the coordinate frame of the robot $\mathcal{A}$ with respect to the coordinate frame of the world $\mathcal{W}$ [12]. Configuration space $\mathcal{C}$ is then defined as the set of all possible configurations q of robot $\mathcal{A}$. More precisely, configuration space of the robot is a "manifold" of dimension n, where n is the number of degrees of freedom of the robot [6]. The term $\mathcal{A}(q)$ denotes a subset of $\mathcal{W}$ occupied by $\mathcal{A}$ in the configuration q.

For a rigid body in 2D space, the configuration must represent the position of the robot and its rotation. The configuration space is therefore equal to

$$\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}, \tag{2.4}$$

where $\mathbb{S}$ is the manifold of 2D rotations.

For a rigid body in 3D space, the configuration space is a 6-dimensional manifold,

$$\mathcal{C} = \mathbb{R}^3 \times \mathbb{S}^3, \tag{2.5}$$

where the three dimensions specify the position of the robot and remaining three dimensions its rotation.

In order to obtain a motion which is safe for traversal by robot $\mathcal{A}$, collisions with obstacles must be avoided. Such a motion must be composed of configurations from subset $\mathcal{C}_{free}$ of configuration space $\mathcal{C}$ called "free space",

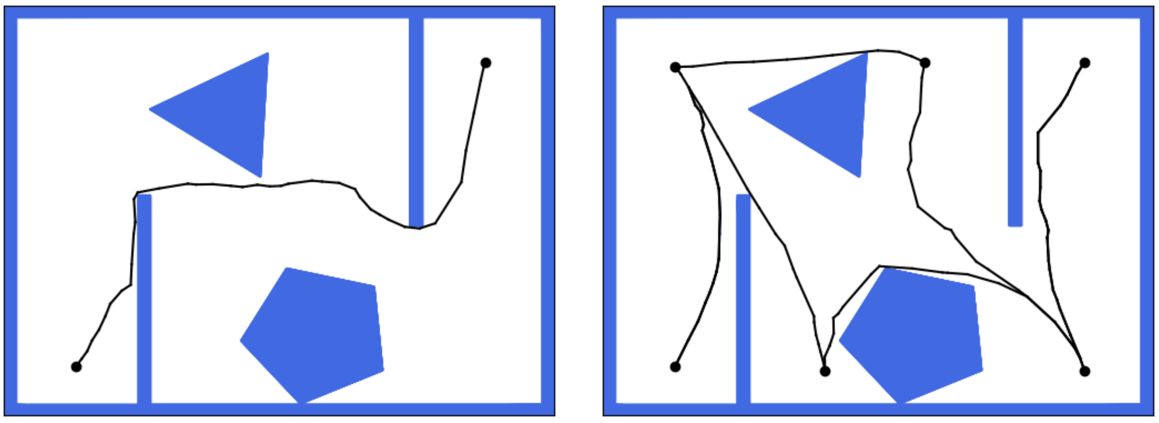
$$\mathcal{C}_{free} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} = \emptyset\}. \tag{2.6}$$

Please note that the $\mathcal{C}_{free}$ space is an open set, as the robot can move infinitely close to an obstacle, but cannot touch it (i. e., have exactly one common point). Set that contains configurations of the robot, which touches an obstacle, is also denoted by "closure" of the free space $\mathrm{cl}(\mathcal{C}_{free})$.

Similarly, set $\mathcal{C}_{obs}$ of configurations of the robot that collides with any obstacle is called "obstacle region",

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}, \tag{2.7}$$

$$\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}. \tag{2.8}$$



(a) : Single-goal planning **(b) :** Multi-goal planning

Figure 2.1: Comparison of single-goal and multi-goal planning (generated by SFF* algorithm)

Every description of a motion planning problem should include a definition of two important configurations q_{init} and q_{goal}, i. e., the initial and the desired final position of the robot. The collision-free "path" τ connecting these two configurations is then defined as

$$\tau : [0,1] \rightarrow \mathcal{C}_{free}, \tag{2.9}$$

where $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$, therefore $q_{init},\ q_{goal} \in \mathcal{C}_{free}$.

The complete motion planning algorithm must provide a continous path without collisions as defined in (2.9), or correctly state that no feasible solution exists.

There can be either one goal configuration — in this case the problem is called as *single-goal planning problem* — or multiple goal configurations, called in a similar way as *multi-goal planning problem*. In the latter case a set of goal locations is given and the task is to find the shortest tour through these locations. Under the term "tour" is meant a closed path that visits each goal exactly once [13]. The difference between single-goal and multi-goal motion planning is depicted in Figure 2.1 — while only one path from the initial to the goal configuration is computed in the problem shown in the figure on the left, in the picture on the right, multiple paths between cities are connected into one tour. This problem together with approaches to its solution, will be defined later in Section 4.2.

Most of the presented motion planning algorithms consist of two fundamental steps: at first they transform (or approximate) the problem into a graph search problem and, secondly, find the shortest path using some graph search algorithm. This can be solved by classic algorithms as Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra algorithm, or using probably the most widely used algorithm today, A* algorithm [14].

2.1.1 Description of rotation in 3D spaces

The description of the object´s rotation in 3D space is more complex than in 2D space, because there are several ways to represent the orientation of the robot. Most used representations of 3D rotations are rotation matrices, Euler angles and unit quaternions.

Rotation matrices are 3×3 orthonormal matrices with an unit determinant. They are the most natural way to represent orientation in space, though there are several issues linked with their usage. Most serious issues are the space requirements on computing systems and numerical precision issues, when using floating point arithmetics. In addition, it is not easy to define the distance between two matrices and interpolate between them, which are common operations required in path planning [15].

Another option is the yaw–pitch–roll formulation, also called “Euler angles”. Threedimensional body can be rotated about three orthogonal axes — *x, y,* and *z*. Yaw is the rotation about the z-axis, pitch is the rotation about y-axis and roll is the rotation about x-axis. The order in which particular rotations are performed is crucial because different order leads to different final orientation of the body in space [6]. In addition, one orientation can be represented by multiple different sets of angles, some of them even by an infinite number of sets of angles — these are called “gimbal locks”. This causes serious issues with sampling, distance measurement and interpolation between sets of Euler angles [15].

The most suitable representation of rotations in 3D space for motion planning are the “unit quaternions”. A quaternion is an imaginary number in the form $a + bi + cj + dk$ where $a, b, c, d \in \mathbb{R}$ and $i^2 = j^2 = k^2 = ijk = -1$. Quaternion is considered as unit when $a^2 + b^2 + c^2 + d^2 = 1$. As shown by Euler, an arbitrary orientation can be achieved by a single rotation about a single axis [16]. The corresponding unit quaternion is in the form

$$Q = (\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \left(\cos\frac{\theta}{2}, v_x \sin\frac{\theta}{2}, v_y \sin\frac{\theta}{2}, v_z \sin\frac{\theta}{2}\right), \tag{2.10}$$

where $v = (v_x, v_y, v_z)$ is the axis of rotation and θ is the angle of rotation. The quaternion algebra offers an easy way to accumulate angles, interpolate between them and sample [15, 16].

Chapter 3

Related works for single-goal motion planning

As mentioned in the previous chapter, the motion planning problems might be divided according to the number of goal configurations into single-goal problems and multi-goal problems. The first category of problems is discussed in this chapter. First, the oldest basic approaches are presented, followed by more modern sampling-based methods for solving high-dimensional problems. Last but not least techniques for single-goal planning with kinodynamic constraints are introduced as well as trajectory planning with polynomial trajectories.

3.1 Basic algorithms

This section discusses the basic approaches to solving single-goal motion planning problems. Most of them work only in special cases, i. e., in environments with additional constraints. It can be 2D spaces with polygonal obstacles and a point or disc-shaped robot. Although relatively old and primitive, they are still applicable to some of today´s problems because they are stable and robust. Based on their general methodology, they can be divided into three main categories: roadmaps, methods based on cell decomposition and potential field methods [12, 11].

"Roadmap" of an environment is a network of one-dimensional lines or curves lying in $\mathcal{C}_{free}$ or $\mathrm{cl}(\mathcal{C}_{free})$. The resulting path consists of a subpath connecting q_{init} with a point A on the roadmap, another subpath connecting q_{goal} with a point B on the roadmap, and the path on the roadmap connecting points A and B. [12, 11] Multiple possibilities are known for constructing a roadmap of n-dimensional environment, two of which will be briefly introduced — visibility graph and retraction.

The *visibility graph*, as introduced by Lozano-Peréz [17], is an undirected graph $G(N,\ L)$, where N is the union of the vertices of all obstacles and positions of the start and goal configurations of the robot. The set of edges L is the set of all links $(n_i,\ n_j)$ such that $n_i,\ n_j \in N$ and the straight line connecting them does not overlap any obstacle. So these lines thus do not lie in $\mathcal{C}_{free}$ but in $\mathrm{cl}(\mathcal{C}_{free})$. An example of a visibility graph and the resulting path is depicted in Figure 3.1a. This definition only applies to a point robot, otherwise the shape of the robot must be approximated by a disc or polygon and obstacles must be inflated. The roadmap is then constructed on the inflated set of obstacles.

The *retraction approach* introduced in [18] for disc-shaped robots, later extended in [19], builds on the idea that the safest way to move the body through an environment with obstacles, is to keep it as far as possible from any obstacle, i. e., equidistant from at least two obstacles all time. This idea was apparently introduced in [20] and leads to construction of “Voronoi diagrams”. These can be constructed in $\Theta(n \log n)$ time [21] and for polygonal environments they consist of linear or parabolic line segments.

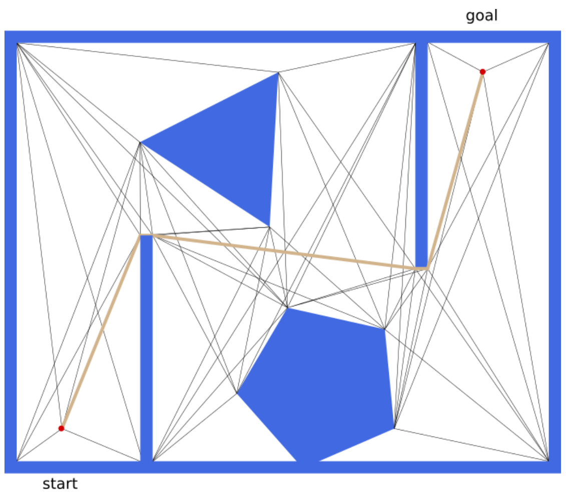
As the name of another general approach, “cell decomposition”, suggests, these methods subdivide a given environment into smaller regions called cells. These cells can then be transformed into a “connectivity graph” that can be searched using standard graph search methods. The connectivity graph usually consists of midpoints of borders of two adjacent cells and/or of centroids of the cells. Graph nodes can then be connected by an edge only if they belong to adjacent cells or even to the same cell [12, 11]. Another option is to directly use the vertices and borders of the cells. The resulting path consists, as in the roadmap approach, of subpaths connecting the initial and goal configurations with the connectivity graph, and a path on the connectivity graph.

In general, there are two main methods of space decomposition — exact decomposition and approximate decomposition.

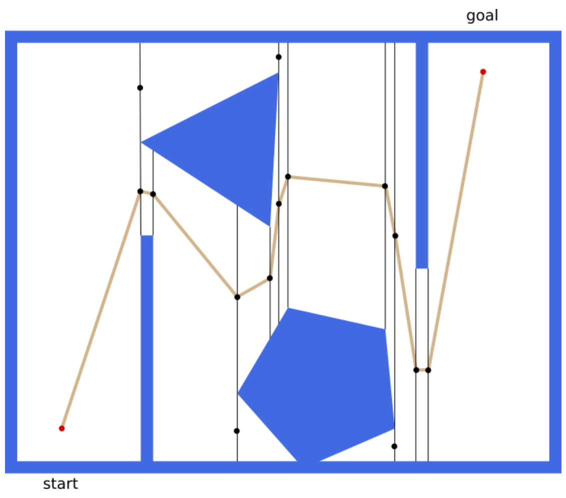
Exact decomposition methods divide the free space of the environment exactly, i. e., the union of the resulting cells equals the same free space. One such method is trapezoidal decomposition, where the environment is divided into trapezoidal and triangular cells [12], as shown in Figure 3.1b. Another option is to utilize triangulation methods and divide the environment into triangles [22], as shown in Figure 3.1d. One of the advantages of triangulation is the ability to improve the path quality by refining the division. For both methods, polygonal obstacles are required.

However the exact methods are not always appropriate, e. g., due to the complexity of the environment. In such cases, it is possible to deploy *approximate methods* of environment decomposition. In these methods, the union of the resulting cells does not equal free space of the environment, but it is still a subset of free space [12]. A typical method is a decomposition to small squares or rectangles of the same size (forming a grid, e. g., as in Figure 3.1c) or even multiple sizes (recursive decomposition of cells containing boundaries of free space and obstacle region). The second variant is also called “quadtree” decomposition (2D space) or “octree” decomposition (3D space), because the environment decomposed using these methods might be represented by a tree of degree 4 (or 8 respectively) [23, 11].

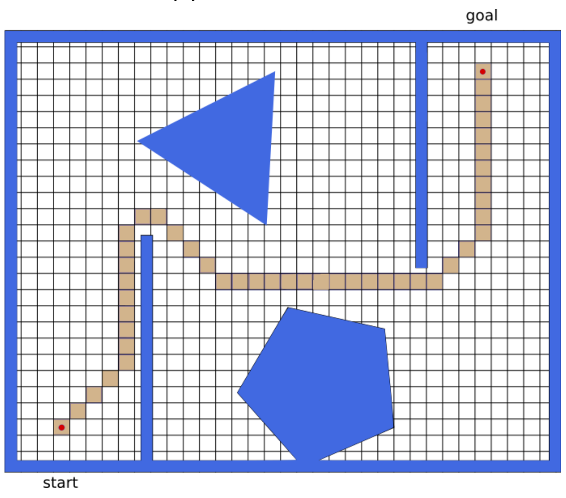
Depending on the resolution of cells, these algorithms might not be “complete”, as defined in Chapter 2. i. e., do not return a valid solution even if some exists. However, for sufficiently fine resolution, the algorithms are able to find such a solution, so these methods are called “resolution complete”.



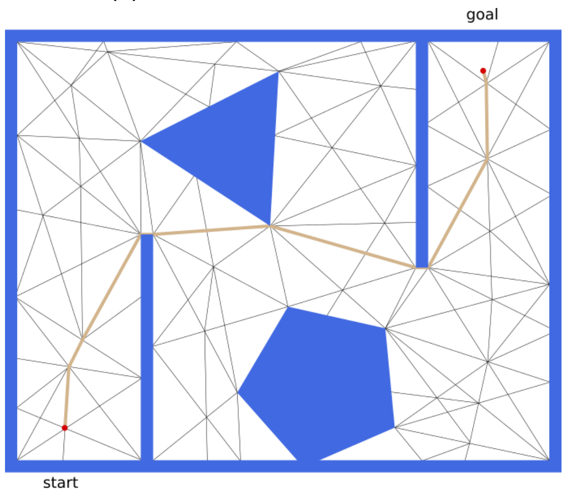
(a) : Visibility graph



(b) : Trapezoidal decomposition



(c) : Grid-based decomposition



(d) : Triangulation

Figure 3.1: Overview of selected basic motion planning methods

One of the major disadvantages of the aforementioned methods is their inefficiency in high dimensions, i. e., in motion planning for robots with a high number of degrees of freedom (DOF), e. g., robotic manipulators. Although some of them might be capable of such a task, execution would be slow. To overcome this issue, methods using "potential fields" have been proposed [24, 25]. Their main principle is simple and intuitive:

"The manipulator moves in a field of forces. The position to be reached is attractive pole for the end-effector, and obstacles are repulsive surfaces for the manipulator parts." [24] Therefore, a *potential function* must be defined, the direction of motion is then determined using the negated gradient of this function.

The main disadvantage of the aforementioned approach is the existence of local minima in environments with concavely shaped obstacles. Several methods have been proposed to overcome this issue: Barraquand et al. [25] present a method filling-up the attractor for low-dimensional spaces when the local minimum is reached (Best-First Planner) and a "random walk" method for high-dimensional spaces (Randomized Path Planner). Other authors test wall following [26] or using functions with one local minimum only — the so-called harmonic functions [27]. However, the process of finding of such a method is quite complicated.

Another problematic part of this approach is the design of a potential function that involves many heuristic parameters that need to be tuned for a specific environment [6].

3.2 Sampling-based motion planning

Sampling-based motion planners are designed to solve high-dimensional problems for robots with many DOF. The main idea is not to construct the entire configuration space $\mathcal{C}$, but instead to randomly discretize it and use a collision checker (a function or simply "black box"), which decides whether the sampled configuration belongs to $\mathcal{C}_{free}$. A similar principle is also used by the Best-First Planner and the Randomized Path Planner [25] to avoid the construction of complex potential functions that guarantee collision-free motion of the robot. This approach also allows planning regardless of the shabe of the robots and obstacles. Thus, the environment might contain non-convex polyhedra, 3D triangles etc. [6].

The problem with the completeness of the algorithm arises again with sampling-based algorithms. For a small number of samples or unevenly distributed samples, the algorithm might not find any valid solution. However, the sampling-based algorithms with random sampling, that are dense in infinity with probability one, are defined as "probabilistic complete". This means the probability that the algorithm will not find a solution, even if some exists, converges with increasing number of samples to zero in infinite time [28].

3.2.1 Asymptotically optimal methods

The sampling-based methods described in the following sections provide, in their elementary versions (sPRM/RRT/EST), a feasible solution to the motion planning problem, if any exists, for the umber of samples growing to infinity. But what if the path planning problem requires finding optimal path, i. e., the one with the shortest length among all other feasible paths? The optimality was already researched on a theoretical basis in [2]. The authors concluded that the problem is very difficult because it *"has both a combinatorial and an algebraic character,"* [2]. Practical approaches based on common sampling-based algorithms are presented by Karaman and Frazzoli [29], who also define the term "asymptotic optimality" of the motion planning algorithm.

For a complete definition of asymptotic optimality, the reader is advised to read [29]. Basically, the probability that an asymptotically optimal planner finds the optimal solution, if there is any robustly feasible solution, converges to one when the number of samples (or iterations of the algorithm) rises to infinity. The probabilistic completeness of such an algorithm is clearly a necessary condition, but it is not a sufficient condition.

3.2.2 Probabilistic Roadmaps

One of the planners utilizing the previously described principles was named “Probabilistic Roadmaps” (PRM) and was proposed by Kavraki et al. [30]. The planning process is divided into two phases: the *learning phase* and the *query phase*.

During the learning phase, a probabilistic roadmap is built by *“repeatedly generating random free configurations of the robot and connecting these with some simple but very fast (...) local planner,”* [30]. In this way, a graph is constructed, the vertices of which correspond to sampled configurations and its edges to collision-free path segments between them. Absence of collisions is just guaranteed by the local planner. In the subsequent query phase, graph searching methods are utilized to find the path between arbitrary initial and goal configurations. This is done in a similar way as for the roadmaps in Section 3.1. The roadmap is reusable, therefore, there can be more query phases with different start and goal positions. PRM therefore falls to the “multi-query planners” category.

Algorithm 1 Probabilistic roadmaps — learning phase [30]

```
Input: $W$ — environment
Input: $N_{max}$ — number of samples
Input: $k$ — maximum number of neighbours
Output: $R(V, E)$ — probabilistic roadmap
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: while $|V| \leq N_{max}$ do
 4:     $q \leftarrow$ random configuration in $W$ such that, isFree($q$) holds true
 5:     $N_q \leftarrow$ k-nearest neighbors of $q$
 6:     $V \leftarrow V \cup \{q\}$
 7:     for $\forall n \in N_q$ in order of increasing distance from $q$ do
 8:         if not SameComponent(q,n) and isFree(q,n) then
 9:             $E \leftarrow E \cup \{(q, n)\}$
10: return $R$
```

The original algorithm for roadmap construction is described in Algorithm 1. First, an empty roadmap is initialized. Then the following steps are repeated until the size of the roadmap equals the required number of samples. A random configuration q in the environment is sampled, which is then checked for collisions with obstacles using aforementioned techniques. If q is in free space, it is added to the roadmap and the set of k-nearest neighbors is found. The samples in this set are ordered with increasing distance from q. Edge connecting q with some neighbor n is created if and only if both q and n do not lie in the same connected component[1] and the path between them is free. The resulting structure is a tree or a collection of trees.

[1]This might be determined using the Union-Find structure.

One of the problems with the original algorithm is the fact that its probabilistic completeness has not yet been proved. Therefore, an alternative version, simplified PRM (sPRM), was introduced in [31]. The main difference between PRM and sPRM is the relaxation of the condition regarding the non-connected vertices in the same component. The final structure of this version of the algorithm can be described as a common graph with loops. However, because the length of the loops is non-negative, classic graph algorithms might still be used for searching in query phase.

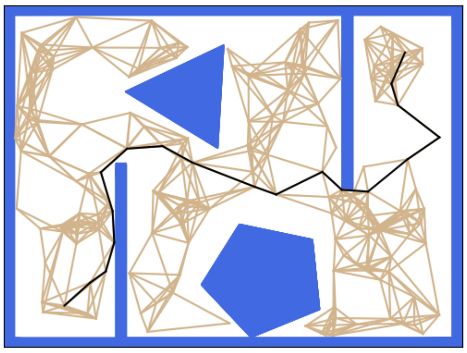
As is evident from Algorithm 1, there are 2 parameters of PRM/sPRM that need to be set. Namely the number of samples in the roadmap N and the number of nearest neighbors k. Figures 3.2 show how these parameters might change form of the roadmap as well as the length of the final path. In both cases, a direct ratio applies — the higher the number of samples the shorter the resulting path and the higher the number of nearest neighbors the shorter the resulting path. However, it should be noted that higher N and higher k lead to longer graph construction times.

Another possible modification is a change in the method of sampling. As suggested by Geraerts et al. [32], instead of fully random points, random sampling on a grid with resolution of increasing size might be used. Or as an upgrade one can use so-called Halton or Hammersley point sets with better coverage than the grid [33, 32]. Such a quasi-random algorithm was named Q-PRM. It has been shown to improve behavior of PRM in complex environments with narrow passages [33].

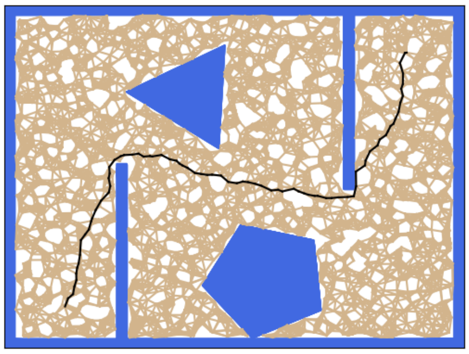
Advanced sampling methods, such as Gaussian sampling, also deal with narrow passages and obstacles. Two points are randomly sampled, between which the distance is selected according to the Gaussian distribution. If exactly one of them lies in $\mathcal{C}_{free}$, it is added to the set of samples. Obstacle-based techniques have a similar behavior, where a point sampled in $\mathcal{C}_{obs}$ moves in a random direction until it becomes free. However, as the authors of [32] conclude, these methods should target only particular areas of the workspace with narrow passages.

Planning with kinodynamic constraints

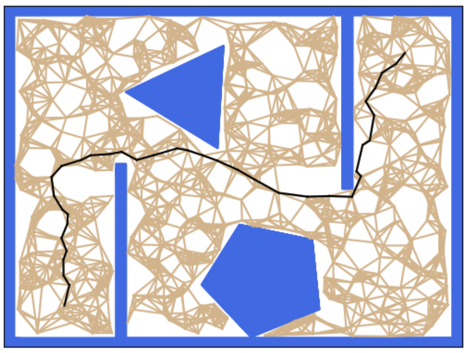
Although both PRM and sPRM algorithms were originally designed for classic Euclidean spaces with paths represented by straight lines, the algorithm might also be adapted for planning with constrained robots. Instead of straight lines, more complex path segments might be used to connect the sampled points in the learning phase. The main disadvantages, unlike other planners, are high computational demands and higher sensitivity to the initial number of sampled nodes. In such cases, a higher number of sampled nodes might lead to longer final paths.



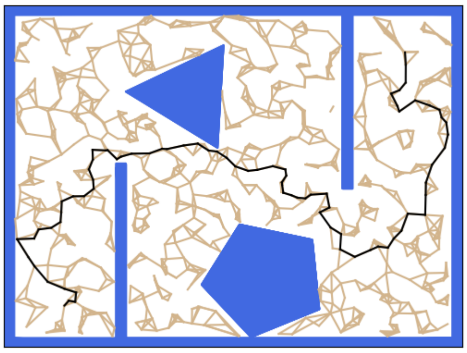
(a) : $N = 200$, $k = 10$



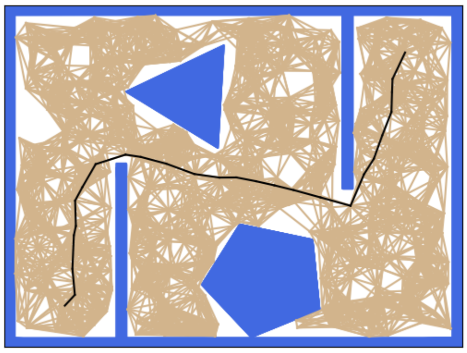
(b) : $N = 5000$, $k = 10$



(c) : $N = 1000$, $k = 10$



(d) : $N = 1000$, $k = 5$



(e) : $N = 1000$, $k = 20$

Figure 3.2: Comparison of influence of different settings for sPRM

Asymptotically optimal version — PRM*

The asymptotically optimal version of the PRM algorithm is denoted by "PRM*" [29]. It is based on sPRM method and, unlike its original version, specifies only the radius of searching for the nearest neighbors of the node, or the number k for its "k-nearest neighbors" variant. This optimal $k = k_{PRM*}$ number equals

$$k_{PRM*} > k^*_{PRM*} = e\left(1 + \frac{1}{d}\right)\log(n), \tag{3.1}$$

where d is the dimensionality of the problem and n is the number of sampled nodes. For simplicity $k_{PRM*} = 2e\log(n)$ always applies. Axample of PRM* algorithm solution is depicted in Figure 3.3.

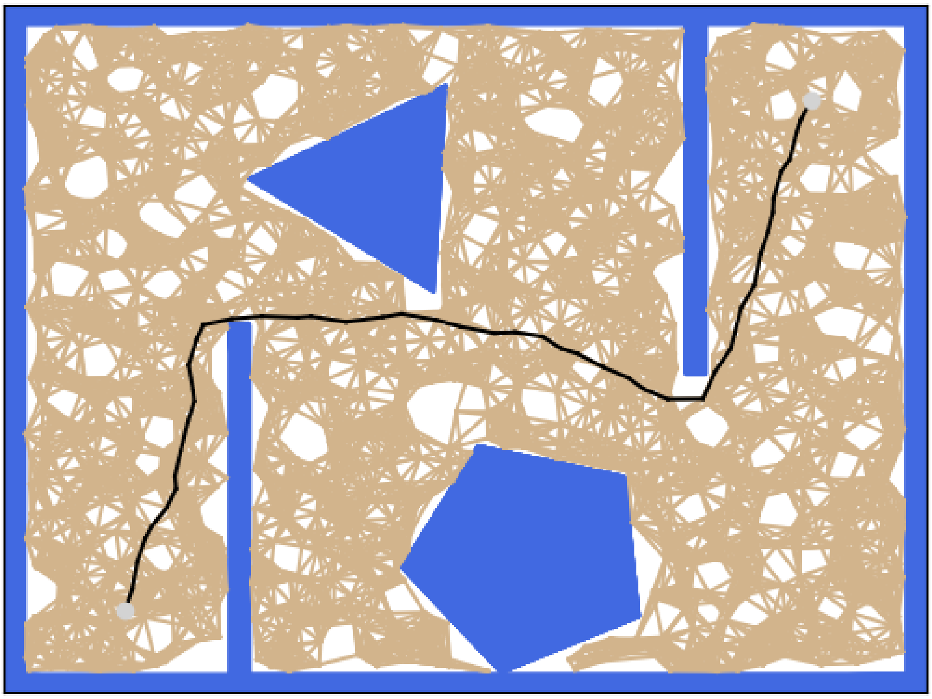


Figure 3.3: PRM* algorithm

3.2.3 Rapidly-exploring Random Trees

Unlike PRM, a planner based on a randomized data structure called Rapidly-exploring random tree (RRT) is a "single query" planner. As its name suggests, the tree in $\mathcal{C}_{free}$ grows, rooted in the initial configuration, towards the goal configuration. Therefore, it cannot be used for multiple queries — a new tree must grow for each. However, it is more appropriate for planning problems where the robot's motion is constrained, e. g., it can only move along curve-shaped paths or under other kinodynamic constraints. The RRT concept was first introduced by La Valle [34].

Algorithm 2 Rapidly exploring Random Trees, with goal biasing

```
Input: $W$ — environment
Input: $q_{init}$, $q_{goal}$ — initial and goal configurations
Input: $N_{max}$ — maximum number of iterations
Input: $d$ — expansion step
Input: $b$ — goal bias
Output: $T(V, E)$ — tree grown from $q_{init}$ towards $q_{goal}$
1: $V \leftarrow q_{init}$
2: $E \leftarrow \emptyset$
3: for $i = 0 \ldots N_{max}$ do
4:     if Random(0,1)$\leq b$ then
5:         $q_{rand} \leftarrow q_{goal}$
6:     else
7:         $q_{rand} \leftarrow$ random configuration in $W$
8:     $q_{near} \leftarrow$ nearest neighbor of $q_{rand}$ in $T$
9:     $q_{new} \leftarrow$ configuration in distance $d$ from $q_{near}$ towards $q_{rand}$
10:    if IsFree($q_{near}$, $q_{new}$) then
11:        $V \leftarrow V \cup \{q_{new}\}$
12:        $E \leftarrow E \cup \{(q_{near}, q_{new})\}$
13:        if distance of $q_{new}$ from $q_{goal}$ is lower than $d$ and isFree($q_{new}$, $q_{goal}$) then
14:            $E \leftarrow E \cup \{(q_{new}, q_{goal})\}$
15:            break
16: return $T$
```

The flow of the algorithm is as follows: the exploration tree T of robot configurations starts with one vertex, q_{init}. In each iteration, a random configuration q_{rand} is sampled (this position might even be blocked by an obstacle) and its nearest neighbor $q_{near} \in T$ is found. Then a new configuration q_{new} is created at a distance d from q_{near} towards q_{rand}, where d is a fixed parameter. If q_{new} and the path from q_{near} to q_{new} lie in the free space, q_{new} and its connection with q_{near} are added to T. Execution ends when a configuration sufficiently close to q_{goal} is found or when the maximum number of iterations is exceeded. Due to the probabilistic completeness of the RRT, there is no guarantee that the path does not exist at all, even if the maximum number of iterations is reached.

Algorithm 2 records the described procedure with further improvement of "goal biasing" [35]. Instead of constantly choosing a purely random configuration q_{rand}, the goal configuration q_{goal} is chosen as q_{rand} with some small probability b. In this way, the algorithm can yield sparser trees in less time because the tree is partially steered towards q_{goal}.

The importance of choosing appropriate parameters d and b is also highlighted in Figures 3.4. It should be clear from Figures 3.4a, 3.4b and 3.4c that a smaller value of d does not necessarily lead to shorter paths. On the contrary, it leads to longer computational times. The parameter should only be adjusted to pass through possible narrow passages. Figures 3.4c, 3.4d and 3.4e confirm previous statements about tree density depending on the goal bias b — the greater the bias b, the sparser the tree. However, as Figure 3.4e shows, high bias might prolong the path, or even no path might be found at all, especially in more complex environments with many obstacles in the line of sight between the start and the goal configurations. High bias also has a negative influence on runtime, as many samples lie in the obstacle region.

One of the other natural modifications is to grow simultaneously two exploring trees at the same time, one from q_{init} (T_{init}) and one from q_{goal} (T_{goal}). The trees are constructed alternately, either by using the same q_{rand} for both trees [36] or by generating a unique one for each tree [35]. The interconnection of trees might be quite problematic, the original algorithm in [36] ends when one of the configurations of T_{init} gets close enough to any configuration of T_{goal}. Alternatively, it is possible to continue the exploration, register candidates for connections and select the best after several iterations. The bidirectional approach should not only be faster, but should also deal with goals in appertures. A tree usually expands from such an apperture easier than it expands “into” the apperture.

Planning with kinodynamic constraints

The described method works well for a point robot in common Euclidean spaces. For constrained robots, the algorithm should be slightly modified: instead of creating q_{new} as a configuration at a distance d from q_{near}, all possible actions from q_{near} are executed and the one with the final position closest to q_{rand} is chosen. This action might be additionally integrated over a small time step Δt to obtain q_{new}. The rest of the algorithm is the same as for classic RRT with the exception of the local planner, which must reflect the constraints considered. An example of planning with kinodynamic constraints, respectively the resulting roadmap and the path of the robot, are depicted in Figure 3.5.

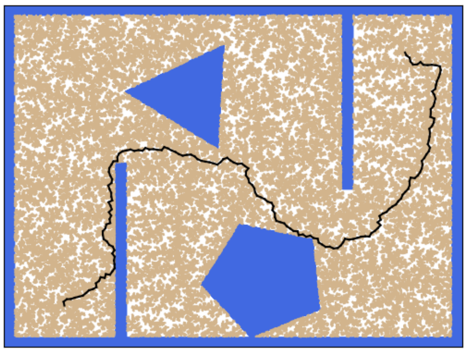
Asymptotically optimal version — RRT*

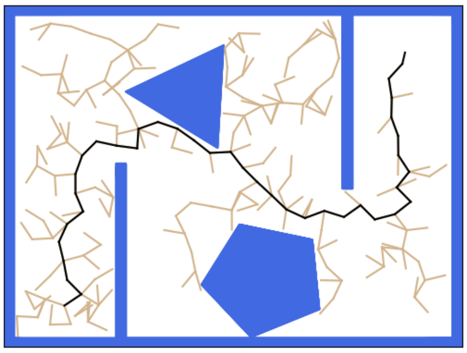
The authors of [29] also propose an algorithm called “Rapid-exploring Random Graph” (RRG). Unlike PRM*, this graph is built incrementally, i. e., nodes are sampled and added to the graph continuously. This is the same as with the RRT algorithm along with the sampling technique. Unnlike RRT, cycles are allowed in the graph (i. e., the roadmap does not form a tree) — after adding the sampled node q, the k-nearest neighbors of q are found and connected to q if there is a collision-free path between them. The constant $k = k_{RRG}$ is defined as

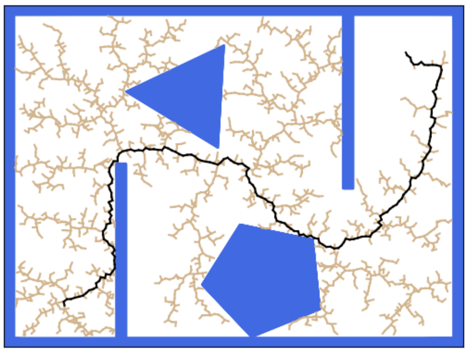
$$k_{RRG} > k^*_{RRG} = e\left(1 + \frac{1}{d}\right)\log(n), \tag{3.2}$$

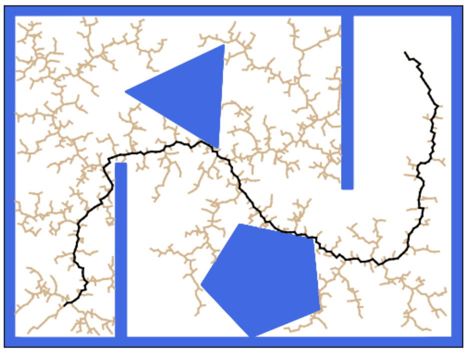
where d is the dimensionality of the problem and n is the actual number of nodes in the graph. Note that the number increases as the graph expands. As with k_{PRM*}, the term $k_{RRG} = 2e\log(n)$ always applies.

However, using graphs instead of trees means increased demands on the storage of datastructures. It would therefore be beneficial to simplify the resulting RRG to a tree. Such an algorithm is denoted by RRT* [29]. The RRT* tree is de facto RRG without cycles. The expansion phase is the same as for RRT, the parent of the newly sampled node is determined


(a) : $d = 0.2,\ b = 0$


(b) : $d = 5,\ b = 0$


(c) : $d = 1,\ b = 0$


(d) : $d = 1,\ b = 0.1$

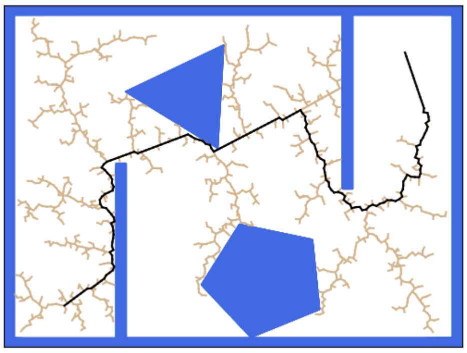

(e) : $d = 1,\ b = 0.95$

Figure 3.4: Comparison of the effect of different settings for RRT

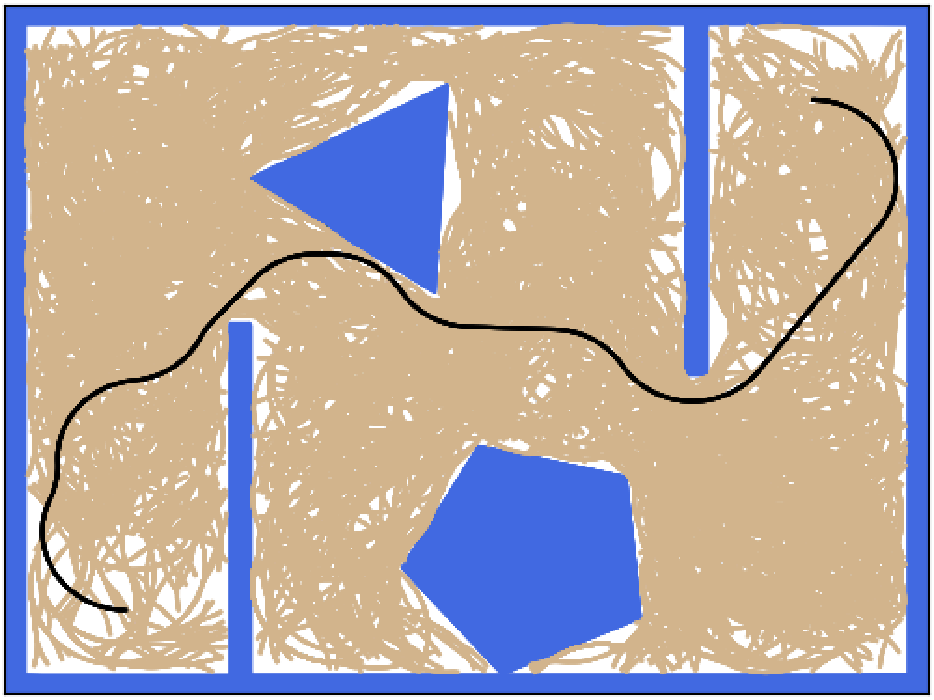


Figure 3.5: RRT with kinodynamic constraints

from its k-nearest neighbors, where $k = k_{RRT*} = k_{RRG}$, so the feasible path from the tree root trough the selected node is the shortest possible. Figure 3.6 shows the functionality of the RRT* algorithm.

3.2.4 Expansive Space Tree

Another motion planning algorithm, the "Expansive Space Tree" [37], is also based on the expansion of the tree $T(V,\ E)$, as in the case of RRT, and therefore also falls into the category of single-query motion planners. Unlike RRT, new configurations are sampled directly within a given distance from some configuration $q \in V$ and are added to the graph when a collision-free connection to q is available. The selection of q from V is random, even though nodes in less crowded areas (typically at graph boundaries) are more likely to be selected for expansion.

The detailed process of EST is described in Algorithm 3. The algorithm starts with a graph T with one vertex, q_{init}. In each iteration, a random configuration from T is picked with probability $1/w(q)$, where $w(q)$ is a function of density of T in neighborhood of q. In the original algorithm [37], $w(q)$ is defined as *"the number of sampled nodes in the tree that lie in N_k,"* where N_k is the set of k-nearest neighbors of q. The probability $1/w(q)$ is therefore higher in less explored areas and lower in sufficiently dense areas. Around this configuration, K random configurations with maximum distance d from q are sampled, and the function w is calculated for each new configuration. The configuration q_n is retained with a probability $1/w(q_n)$ and is added to the tree only when it lies in free space and can be connected with

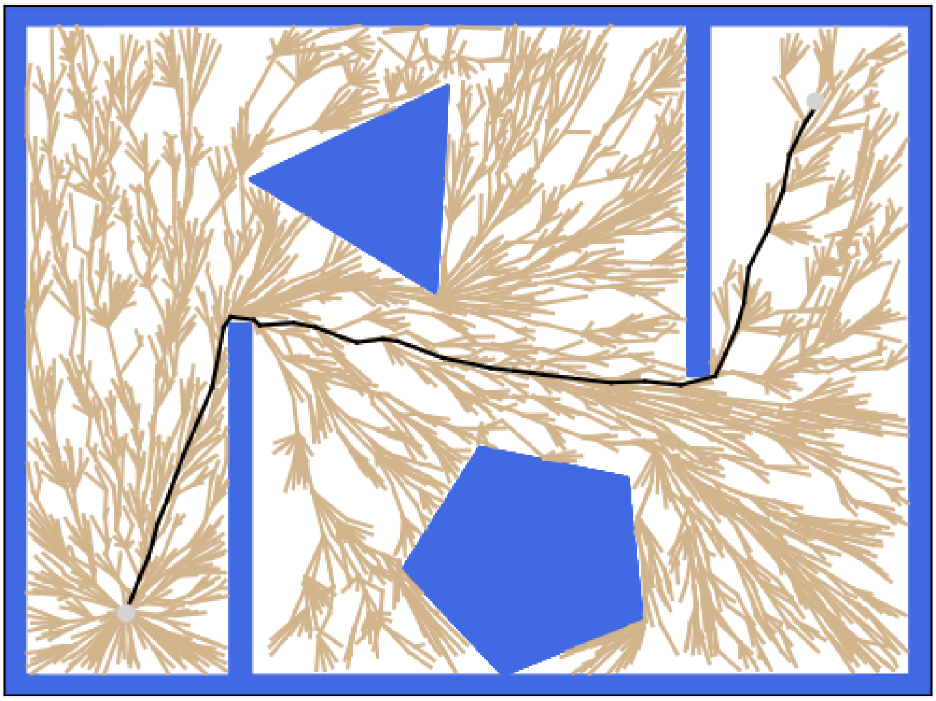


Figure 3.6: RRT* algorithm

Algorithm 3 Expansive Space Tree

Input: W — environment
Input: $q_{init},\ q_{goal}$ — initial and goal configurations
Input: N_{max} — maximum number of iterations
Input: d — maximum distance between samples
Input: K — number of configurations to sample in the neighborhood
Output: $T(V, E)$ — tree grown from q_{init} towards q_{goal}

```
1: $V \leftarrow q_{init}$
2: $E \leftarrow \emptyset$
3: for $i = 0 \ldots N_{max}$ do
4:     $q \leftarrow$ random configuration from $V$, picked with probability $1/w(q)$
5:     $N_k \leftarrow K$ random configurations in $W$ with maximum distance $d$ from $q$
6:     for $\forall q_n \in N_k$ do
7:         calculate $w(q_n)$, retain $q_n$ with probability $1/w(q_n)$
8:         if IsFree($q,\ q_n$) then
9:             $V \leftarrow V \cup \{q_n\}$
10:            $E \leftarrow E \cup \{(q,\ q_n)\}$
11:            if distance to $q_{goal} \leq d$and IsFree($q_n,\ q_{goal}$) then
12:                $E \leftarrow E \cup \{(q_n,\ q_{goal})\}$
13:                break
14: return $T$
```

parent configuration q. Execution ends when new configuration in T can be connected with the goal configuration q_{goal} or maximum number of iterations N_{max} is exceeded. As with RRT, achieving N_{max} does not guarantee the non-existence of any path from q_{init} to q_{goal} due to the probabilistic completeness of the algorithm.

The authors of [37] also propose a bidirectional approach. One tree is expanded from q_{init} and one tree is expanded from q_{goal}. The expansion phases of both trees alternate regularly. For tree incterconnections, the same rules might be applied as for RRT, although the trees in the original algorithm are joined by the first possible connection.

3.3 Planning for Dubins vehicle

All the planners mentioned until now were connecting robot configurations with straight line segments. It concerned 2D as well as 3D problems. In fact, most robots cannot move on such trajectories, or at least the movement is not optimal (e. g., the robot has to stop and adjust its direction). Of course, it is also possible to transform a planned path into a trajectory in the robot controller, although such a path may lose its optimality or even become infeasible. Therefore, it is plausible to take such limitations and robot's model into account in the planning phase and to adjust the shape of the used path segments.

In 2D, one such robot is a simple car. A simple car is an example of a non-holonomic vehicle. Namely, it has 3 degrees of freedom (x, y, θ), even though only its speed u and direction of the wheels (steering angle) ϕ can be changed. Note that at a constant steering angle, the car would follow a circular trajectory with perimeter ρ. The distance between the front and rear axles is L and the direction of the wheels ϕ is also limited to the range $(\phi_{min},\ \phi_{max})$ (e. g., wheels turned at 90° would block the car). The angles are typically symmetrical, $\phi_{min} = -\phi_{max}$. Also note that the maximum of the steering angle implies the minimum turning radius ρ_{min}. The model of a simple car is also depicted in Figure 3.7. The motion of a simple car can be described by 3 motion equations [6]:

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}u\cdot\cos\theta,\\ u\cdot\sin\theta,\\ \frac{u}{L}\tan\phi\end{pmatrix}. \tag{3.3}$$

Assume that $u \in \{0,\ 1\}$ (the car is either stopped or has a constant forward velocity) and there is a minimum turning radius ρ_{min}, which implies the maximum steering angle. Such a model is called "Dubins car" [6].

As Dubins [38] proved, the optimal trajectory for a Dubins car is a continuously differentiable curve consisting of a maximum of three parts, each part being either a straight line (S–segment) or an arc with radius ρ_{min}, oriented to the left (L–segment) or to the right (R–segment). All possible combinations, reffered to as "Dubins maneuvers" are therefore

$$LSL,\ RSR,\ LSR,\ RSL,\ RLR,\ LRL.$$

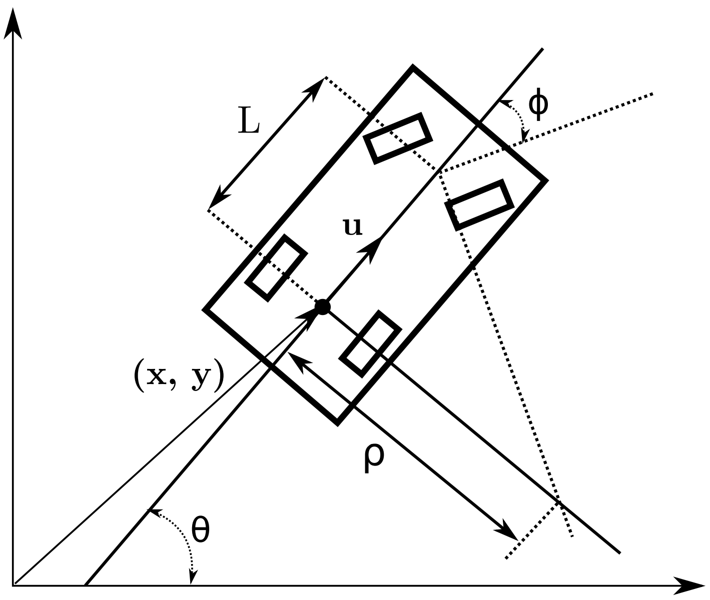


Figure 3.7: Model of a "simple car"

Each segment is parametrized. The boundary L or R curved segments are specified by the central angle of the corresponding circle segments, ϕ_s and ϕ_e, $\phi_s,\ \phi_e \in [0,\ 2\pi)$. The middle L or R curved segment is similarly specified by the central angle of the corresponding circle segment ϕ_c, $\phi_c \in (\pi,\ 2\pi)$ (must be greater than π, otherwise another kind of Dubins maneuver would be optimal [6]). The straight segment is characterized by its length L, $L \geq 0$. The optimal Dubins maneuver for a specific initial and goal configurations might be found after calculating all 6 possible Dubins maneuvers and choosing the shortest of them or by following precise guidelines determined by the relative position of the initial and goal configuration, as described in [39].

The are numerous libraries for computation of the optimal Dubins maneuver for given configurations, one of which is "opendubins"[2] also used to solve more complex Dubins vehicle problems (Generalized Dubins Interval Problem for solving of Dubins Traveling Salesman Problem with Neighborhoods) [40].

3.3.1 Dubins Airplane Model

All the principles described apply only to 2D. Even Dubins in [38] proved the optimality of three-segment paths for 2D and left the question of optimality for higher dimensions open. But with the rise of the unmanned aerial vehicles (UAVs), this had to be resolved — the Dubins car model was extended to 3D by adding a configuration variable for altitude and an additional constraint for maximum climb-rate. Such a model was named "Dubins Airplane" [41]. However, as Owen et al. [42] point out, the problem is much more complex,

[2] Available at `https://github.com/comrob/gdip`.

because for some specific altitude ranges there may be an infinite number of paths with a minimum distance.

The relations of the inertial velocity of an UAV and its airspeed v, heading angle θ and pitch angle (or “flight-path angle”) ψ are:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} v \cdot \cos\theta \cos\psi \\ v \cdot \sin\theta \cos\psi \\ -v \cdot \sin\psi \end{pmatrix}. \tag{3.4}$$

According to [42], the kinematic model of an UAV is consistent with commonly used aircraft guidance models as follows:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v \cdot \cos\theta \cos\psi^C \\ v \cdot \sin\theta \cos\psi^C \\ -v \cdot \sin\psi^C \\ \frac{g}{v}\tan\lambda, \end{pmatrix}. \tag{3.5}$$

where v is the airspeed, θ is the heading angle, ψ^C is the desired pitch angle, λ is the bank angle and g is the gravitational acceleration. Note that this model does not include dynamics of the UAV, or e. g., the windspeed.

One possible practical application, which does not take into account the climb-rate constraint, was introduced in [43]. Classic RRT with Dubins curves as segments is expanded from the projection of the initial configuration to the $x - y$ plane towards the projection of the goal angle into the same plane. The z coordinate is then additionally computed, so the robot climbs linearly along the entire 2D Dubins path.

A more advanced method of planning is introduced in [41] and described in more detail in [42]. Based on the altitude difference between the initial and the goal configurations, the path is categorized as either low-altitude, medium-altitude, or high-altitude. The low-altitude case is similar to the method introduced in [43] — first the 2D Dubins path is computed, the ascent of the UAV is linear over the entire path. Thanks to the classification, the pitch-angle constraint is not violated. In the high-altitude case, the difference in altitude is so great that it is necessary to add additional helices to the path to gain altitude. In the last case, medium-altitude, the altitude cannot be gained on a 2D Dubins path without violation of the pitch-constraint, but after inserting full helix, the altitude would be too high. It is therefore necessary to increase the length of the 2D path in another way, typically by inserting another curve segment. Although only the ascent of the UAV was considered in the previous contemplations, the situation for the descent is the same.

Another possible approach is to consider Dubins curves not only for the horizontal profile of the path, but also for the vertical profile. This method was presented in [44]. The horizontal Dubins path is computed first, followed by the vertical Dubins path, which is determined by the altitudes of the initial and the goal configurations and the length of the vertical path.

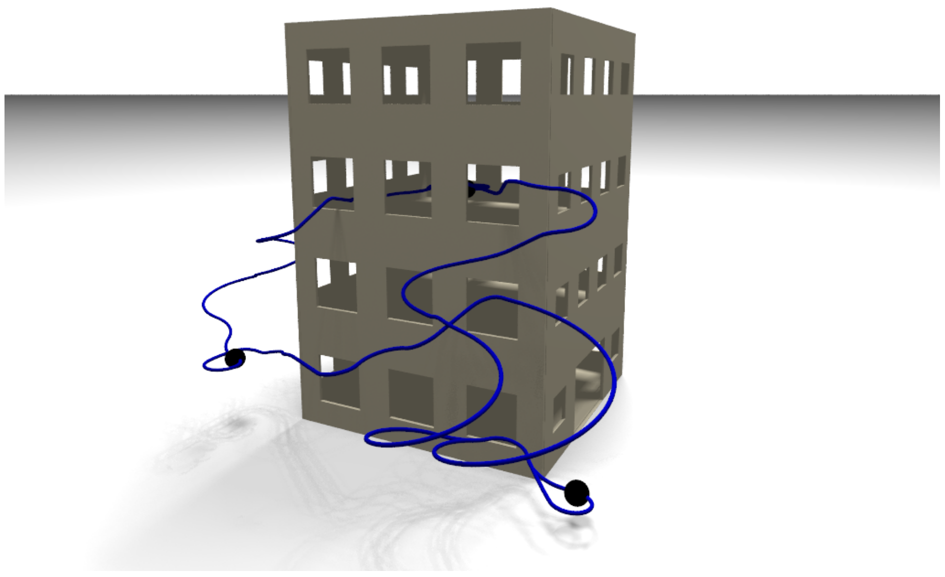


Figure 3.8: Solution of a problem for the Dubins airplane model

While the final path is not feasible, the turning radius for both profiles iteratively increases until the pitch limit is met. An example of a solution with these 3D Dubins curves is depicted in Figure 3.8.

3.4 Polynomial trajectories

The algorithms and methods discussed so far only concerned geometry planning without taking into account the time and dynamics of the robot (e. g., maximum thrust or rotation speed). Before deploying to a real robot, the resulting path must typically be transformed into a "trajectory" by some trajectory generator that converts it to a time domain and ideally takes these limitations into account. The on-board controller can only follow the trajectory in the time domain. The "trajectory" is defined as:

$$\sigma : \; [t_0, \; t_f] \rightarrow \mathcal{C}, \tag{3.6}$$

where $\mathcal{C}$ is the robot configuration space.

The final shape of the trajectory may match the shape of the original geometric path, but this approach might be inefficient. For waypoints connected by straight lines (either in 2D or 3D), this means that the robot must stop at each waypoint and adjust its rotation, which is inefficient and also sometimes infeasible (e. g., an airplane model). The proposed solution via Dubins paths might solve the feasibility problem, but the issue with efficiency in terms of motor thrust and time still remains.

3.4.1 Minimum snap trajectories

One of the trajectory generators introduced in [45] solves the problem of optimality by minimization of snap (second derivative of acceleration). The authors assume model of a quadrocopter described by Newton's equations:

$$m\ddot{\mathbf{r}} = -mg\mathbf{z}_W + u_1\mathbf{z}_B, \tag{3.7}$$

where m is the weight of quadrocopter, $\mathbf{r} = [x,\ y,\ z]^T$ is the position vector, g is the gravitational acceleration, $-\mathbf{z}_W$ is the direction of gravitational acceleration, u_1 are forces of rotors in direction of $\mathbf{z}_B$. Given the yaw angle ψ, they show that dynamics of such a quadrocopter is "differentially flat", i. e., *"the states and the inputs can be written as algebraic functions of four carefully selected flat outputs and their derivatives,"* [45]. The four flat outputs are $x,\ y,\ z,\ \psi$. Based on these results, they propose an algorithm for generation trajectories in the area of flat outputs and also a controller for following the generated trajectories.

This trajectory generator has been successfully deployed in [46] in combination with RRT* not only for 3D but also for 2D problems. This geometric planner is used to find a collision-free path, which is then pruned to a minimum set of waypoints. These points are connected with polynomial segments into a smooth minimum snap trajectory. This decoupled approach is compared to the RRT* algorithm, which uses polynomial segments directly for expansion. The main advantage of the second approach is the asymptotic convergence to the globally optimal solution, on the contrary, the authors emphasize the importance of a suitable choice of segment time, which must be known a priori in this case. Despite the loss of asymptotic optimality, the decoupled approach generates higher quality trajectories in less time [46].

The work of Richter et al. has been extended multiple times, e. g., in [47] for a local replanner based on Informed RRT*, or in [48].

3.4.2 Minimum jerk trajectories

An alternative to the minimum snap trajectories are the minimum jerk[3] trajectories introduced in [49]. The authors assume a similar model of quadrocopter as in [45], with the total thrust scalar $f \in \mathbb{R}$ as the control input. The quadrocopter state space has 9 dimensions because the angular rate commands are assumed to be tracked without error. Furthermore the trajectory generation is solved separately for each axis — their recombination is necessary only to check

[3]The jerk is the first derivative of acceleration.

the feasibility of dynamics. This also means that the same algorithm might be used for both 2D and 3D trajectory generation. An example of a minimum jerk trajectory is illustrated in Figure 3.9.

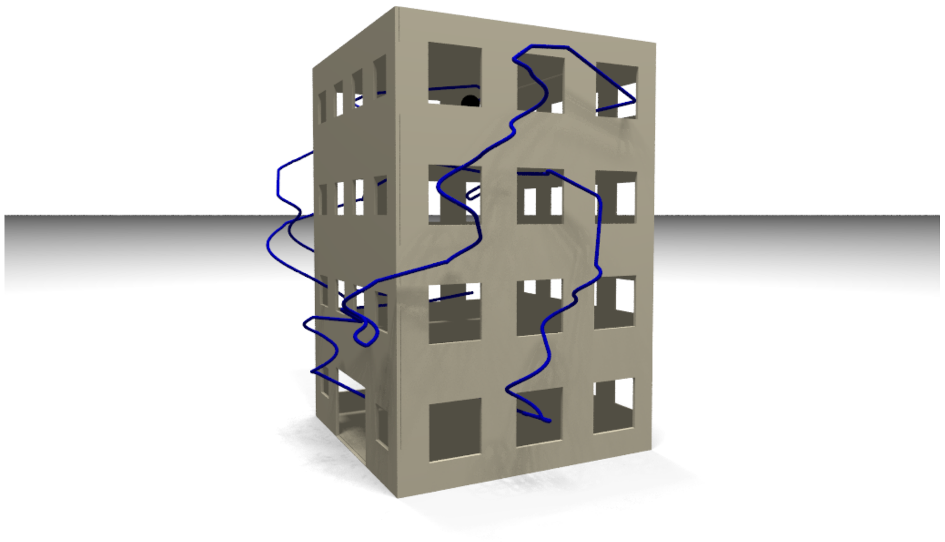


Figure 3.9: Minimum jerk trajectories as a solution of the single-goal trajectory planning

Let the state of quadrocopter for axis k be given as $s_k = (p_k,\ v_k,\ a_k)$, then the optimal trajectory is defined[4] as a polynomial of fifth order:

$$s^*(t) = \begin{bmatrix} \frac{\alpha}{120}t^5 + \frac{\beta}{24}t^4 + \frac{\gamma}{6}t^3 + \frac{a_0}{2}t^2 + v_0 t + p_0 \\ \frac{\alpha}{24}t^4 + \frac{\beta}{6}t^3 + \frac{\gamma}{2}t^2 + a_0 t + v_0 \\ \frac{\alpha}{6}t^3 + \frac{\beta}{2}t^2 + \gamma t + a_0 \end{bmatrix}, \tag{3.8}$$

where $s_0 = (p_0,\ v_0,\ a_0)$ is initial condition and $\alpha,\ \beta,\ \gamma$ are variables. These variables are then determined from the function of components of the final state. The final state might be defined either fully or partially (e. g., without specifying the final velocity).

The authors of [49] also propose an algorithm for checking the feasibility of the trajectory in terms of dynamics, i. e., compliance with the minimum and the maximum thrust of quadrocopter's motors and with maximum rotation speed.

The latter trajectory generator was implemented e. g., in [50] as a local planner. For each iteration of planning, multiple polynomial motion primitives are sampled with varying yaw angles, final altitudes and positions, so that they evenly cover the field of view of the

[4]For precise derivation, please see [49]

deployed camera. The global planner works independently on a simple voxel occupancy grid and is based on pure breadth-first search.

Another planner was presented in [51]. Similarly to [46] the authors utilize the RRT* algorithm, more precisely its extension called “Kinodynamic RRT*”. Its principle is the same as in the second approach of [46], i. e., they use polynomial primitives directly for tree expansion. However, in order to decrease number of unreasonable states, the sampling is guided by an artificial potential field. First, the position is randomly sampled in 3D and the velocity and acceleration are confined within the cone. This cone is determined by the direction and size of the attractive force of the potential field.

Chapter 4

Related works for multi-goal motion planning

As mentioned in the previous chapters, the solution of the multi-goal path planning problem typically consists of finding of paths between each pair of goals (for this category of problems also called “cities”) and finding the best sequence of their visits. This general approach is called “decoupled”. The first part of the task, finding paths, is solved by the single-goal planners introduced in the previous chapter. This chapter discusses the suitability of some sampling-based algorithms for multi-goal planning, as well as the second part of the multi-goal planning problem. Two possible approaches to finding the correct sequence of nodes to visit are described — solving the task as a Traveling Salesman Problem, or solving it as an Orienteering Problem.

4.1 Sampling-based planners for multi-goal planning

The single-goal sampling-based planners can be divided, as described in Section 3.2, into multi-query *(roadmap constructing)* planners and single-query *(tree expanding)* planners. The only representative of the first group that was introduced is the PRM algorithm. Due to its multi-query nature, the PRM/sPRM algorithm is more suitable for multi-goal path planning than single-query planners. To find paths between each pair of given cities, only query phases need to be repeated and only one roadmap constructed. An example of such paths found using sPRM is depicted in Figure 4.1. The Travelling Salesman Problem is formulated using the lenghts of found paths, and after solving it, i. e., after obtaining the optimal order of nodes to visit, the corresponding paths directly form the final tour.

The multi-goal path planning with RRT is not straightforward. Of course, it is possible to grow independent trees for each start–goal pair, but such an approach would be very computationally and time consuming. A solution to this problem has been proposed in [52], where multiple RRT trees are grown from all target locations, alternately in a random manner similar to [35]. When two trees approach each other, they simply merge and grow afterwards as one tree. An example of the roadmap and final paths between goals is depicted in Figure 4.2. The transformation to the final tour is then the same as for the sPRM algorithm.

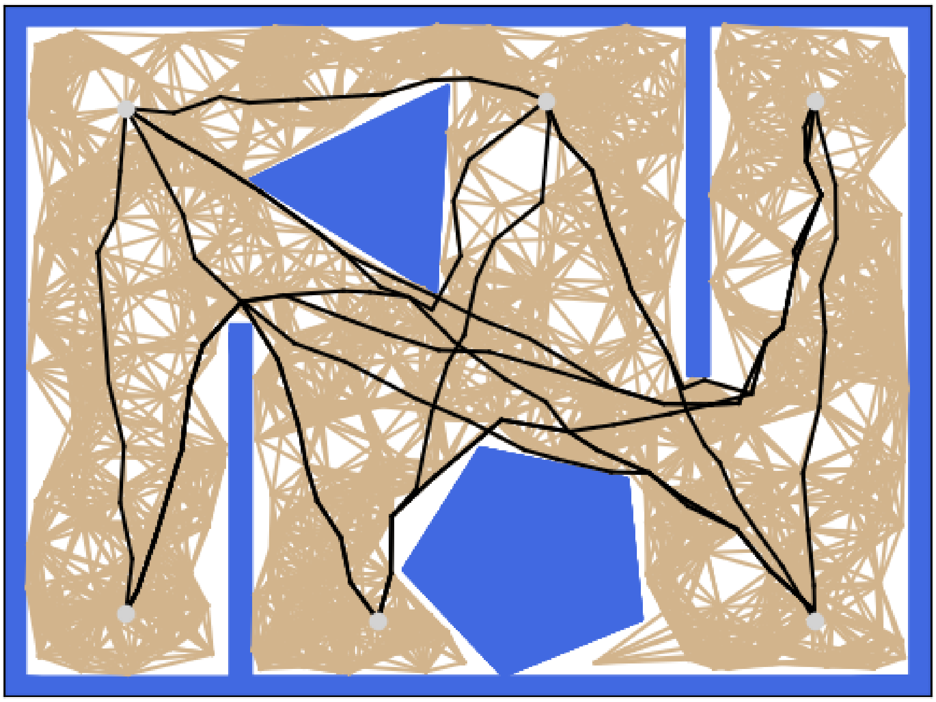


Figure 4.1: Path between multiple cities found using sPRM

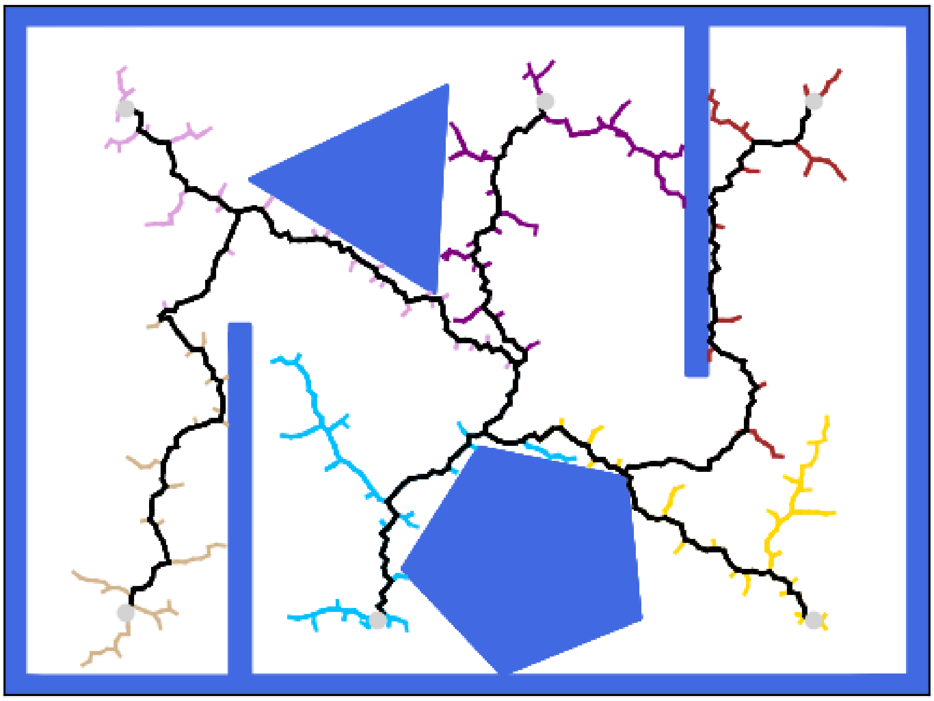


Figure 4.2: Paths found using Multi-T-RRT

As far as the EST algorithm is concerned, to the best of the author's knowledge, no multi-goal planner has been proposed so far. Therefore, the only solution is to plan a path for each start–goal pair.

4.2 Traveling Salesman Problem

According to [53], the TSP is *"probably the most widely studied combinatorial optimisation problem and (...) has also become a standard testbed for new algorithmic ideas."* It is basically an extension of the classic NP-complete problem of finding the Hamiltonian circuit (the circuit visiting each vertex exactly once) on the undirected graph G. In addition, in the assignment of TSP, the edges are weighted (i. e., a cost function $c: \ E(G) \rightarrow \mathbb{R}^+$ is given) and the main challenge is to find the Hamiltonian circuit with the minimum sum of cost.

Its motivation c is based on the problem of the salesman, who is obliged to visit cities indexed from 1 to n, each of them exactly once, having to start and end in the "base" city 1, so that the distance traveled is the smallest possible [54]. This is equivalent to a linear program

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{j=1, i\neq j}^{n} c_{i,j} x_{i,j} \\
\text{s. t.} \quad & \sum_{i=1, i\neq j}^{n} x_{i,j} = 1, \qquad j = 1, \dots, n \\
& \sum_{j=1, i\neq j}^{n} x_{i,j} = 1, \qquad i = 1, \dots, n \\
& s_i + c_{i,j} \leq s_j + M \cdot (1 - x_{i,j}) \qquad i = 1, \dots, n, \ j = 2, \dots, n \\
& x_{i,j} \in \{0, 1\}, s_i \in \mathbb{R}
\end{aligned} \tag{4.1}$$

where c is the distance matrix, $x_{i,j} = 1$ when city i immediately precedes city j in the solution ($x_{i,j} = 0$ otherwise), M is a large integer (so-called "big M") and s is a real number used to eliminate subtours.

The most common version of the TSP is called "symmetric TSP", i. e., the case, where the path from A to B costs the same as the path from B to A. Moreover, when cities correspond to points in metric space and the costs of the edges between them correspond to metric distances, the problem is marked with "metric TSP" [53]. For this special case of metric TSP there are several heuristics, they will be described later. Euclidean metrics are often used — then this particular problem is called "Euclidean TSP". An example of a solution of such problem is illustrated in Figure 4.3.

However, when the distance from A to B differs from the distance from B to A, the problem is classified as "asymmetric TSP" (ATSP). The assignment of ATSP might be transformed into symmetric TSP and therefore solved by the same algorithms. The disadvantage of such a conversion is a significant increase of cities, 2 – 3 times [53].

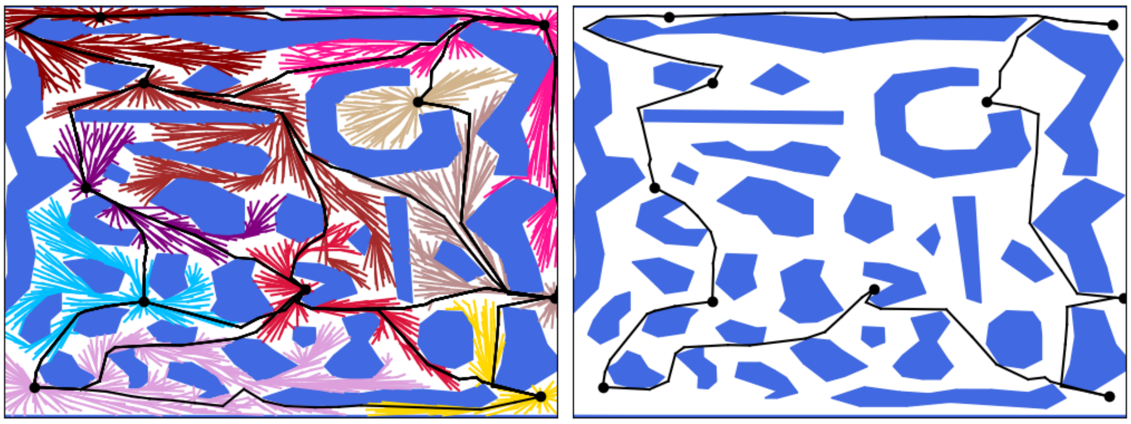

(a) : City-to-city paths **(b) :** Final TSP solution

Figure 4.3: Result of TSP on paths found with SFF* planner

The algorithms solving the TSP have been developed for decades, starting with general cutting plane methods in integer programming applied to the solution of TSP for 49 cities in the USA [55], more advanced and efficient "branch & cut" methods [56], the "branch & bound" algorithm [57], and ending with high-performance exact TSP solvers for large instances such as Concorde [58].

For large or oppositely very small instances, it is useless and often inefficient to find the exact solution to the problem, in which case it is advisable to use heuristics. One might easily think of a basic greedy algorithm, where the tour starts in any city, continues to the closest univisited one, and so on, until all cities connect to the Hamiltonian circuit [59]. However, such an algorithm might not find a feasible tour, even when one exists.

If the problem instance falls into the category of metric TSPs, two classic heuristic algorithms might be applied to solve the problem: the double-tree 2-approximation heuristics based on doubling the edges of the graph's minimum spanning tree [60], or Christofides' $\frac{3}{2}$-approximation algorithm combining the minimum spanning tree of the graph with minimum weight matching of nodes of odd degree in the tree [61].

Another possible solution is to utilize the Variable Neighborhood Search (VNS) method, a general metaheuristic for combinatorial optimization problems, including TSP [62]. Like other heuristics, VNS is an iterative algorithm, each iteration consists of two steps: "shake" and "local search". During the shake step, the initial solution x changes to a random one x' in its neighborhood. In the local search step, solution x' is improved to some solution x'' using general local search methods. If solution x'' is better than incumbent solution x, solution x is replaced by x'' and the iteration continues until a certain set limit is met (e. g., the maximum number of iterations).

One of the most powerful heuristics for the solution of TSP is Lil-Kernighan heuristic (LKH) based on local search methods [63]. Unlike classic k-OPT local search methods for the solving combinatorial problems with fixed k, the parameter k is chosen adaptively to maximize the profit of node swapping in each iteration. It can also natively solve the ATSP problem without any need of conversion to its symmetric variant. To date, the LKH has been improved several times and has evolved into the highly efficient Lil-Kernighan-Helsgaun heuristic [64], which is available as a free library for non-commercial usage[1].

4.2.1 Relaxation for multi-goal path planning

The authors of [65] propose a "lazy" approach to the multi-goal path planning problem. Instead of finding paths between each pair of cities and solving the TSP in the end, the path lengths between the cities are estimated using the Euclidean distance (i. e., all obstacles are ignored) and the initial TSP is solved. The resulting "subpaths" are then planned using either RRT or RRT* algorithm, the initial distance estimates are replaced by the actual path length and the TSP solver is restarted with this modified table. This process is repeated until the solution is stable, i. e., the solution of the final TSP consists of already expanded paths. This method is further denoted by "Lazy TSP".

4.3 Traveling Salesman Problem for Dubins vehicles

The Traveling Salesman Problem for tasks with straight paths and Euclidean metric (ETSP) has already been discussed in the previous section (see Section 4.2). A modification of this problem for Dubins vehicles is denoted by Dubins Traveling Salesman Problem (DTSP) [66]. Even though the problem of TSP for Dubins vehicles might be solved in almost the same manner — first to solve the corresponding ETSP and convert straight paths to Dubins maneuvers (so-called Dubins Shortest Path Problem [38]) — such a solution might even be infeasible or suboptimal, e. g., in environments with obstacles [66].

It is usually also unnecessary to visit nodes with a particular heading. Therefore, the heading for each node can be adjusted to make the resulting tour shorter. In such cases, the previous decoupled approach can be used with a slight midification of solving the Dubins Touring Problem instead of classical DSPP, as presented in [67]. Another possible heuristic called "Alternating algorithm" was introduced in [68]. The odd-numbered edges of initial ETSP solution are retained, the remaining are substituted with Dubins maneuvers, so the connections of nodes are smooth.

Instead of decoupled approach, DTSP might be addressed by direct approach, as described in [66]. For each node, k heading angles are sampled (uniformly or by informed sampling [67]), each node can then be considered as a cluster of k points in graph G. In the next step all $k^2 n \cdot (n-1)$ Dubins maneuvers between configurations corresponding to pairs of points in distinct clusters are computed. Note that the problem is asymmetric, therefore two maneuvers

[1]Available at `http://webhotel4.ruc.dk/~keld/research/LKH-3/`.

must be computed for each pair. The final tour must lead through each cluster and visit only one point of each cluster. This modification of TSP is known as Generalized Asymmetric TSP (GATSP) and can be transformed to ATSP using "Noon-Bean transformation" [69].

Consider the transformation of the described graph $G(V,\ E)$ to $G'(V',\ E')$. The vertices of the original graph remain the same, i. e., $V' = V$. First, all vertices corresponding to the same cluster form a circle of length zero in G'. Then, for each edge $e = (p_i^n,\ p_j^m) \in E$ of clusters i and j, a new edge $e' = (q_i^n,\ q_j^{m+1}) \in E'$ is created — the edge still links the corresponding pair of clusters, but not the same pair of points. The cost of each existing edge $e \in E$ is moreover increased by M, where $\sum_{(i,j)\in E} c_{i,j} < M < +\inf$ and $c_{i,j}$ are the costs corresponding to the edge $e = (i, j)$. The cost of each edge between a pair of unconnected points from distinct clusters is $2M$. After resolving the ATSP, the result must be converted to the original assignment in reverse.

4.3.1 Dubins Orienteering Problem

In the TSP, all cities have to be visited during the tour. It is not necessary to visit all cities in the orienteering problem (OP). Instead, cities are ranked with a certain score or "reward" collected by the agent when visiting a particular city. In addition, such an agent has a limited "travel budget" (i. e., the maximum total cost of his tour). The goal of the orienteering problem is to maximize the total collected reward during the agent's tour. Because TSP and OP are combinatorial optimization problems, algorithms and methods for solving them are very similar or even the same [70].

Orienteering problem for curvature-constrained vehicles, Dubins Orienteering Problem (DOP), was introduced in [71]. This problem combines the classic Euclidean OP with the DTSP to solve the OP for an UAV. The proposed method utilizes VNS metaheuristic [62] mentioned earlier in this chapter for direct solution of the problem. The presented principles might also be applied to the Physical Orienteering Problem (POP) [72] for complex environments with obstacles.

Chapter 5

Space Filling Forest

In [4], a novel sampling-based approach to multi-goal path planning was proposed. Multiple trees expand from all goal configurations until they meet each other. Nodes for tree expansion are randomly chosen from an open list of active frontier nodes. Each node is initially part of this list, but is removed from it after k unsuccessful expansion attempts. In each iteration, new node is sampled in free space at a distance d from the selected node for expansion. The local planner verifies colissionlessness of the path between them. In addition, there must be no node with a distance to the new node less than d. In this way, the trees cannot grow "into themselves" and expand to unexplored areas. When two trees, T_i and T_j approach each other so that a node from T_i is at a distance $\leq d$ from a node from T_j and these nodes can be connected by a collision-free path, a path is established connecting the goals i and j going through these nodes. An example of the resulting roadmap and final paths is depicted in Figure 5.1.

One of the greatest advantages of SFF, thanks to its expansion techniques, unlike RRT and PRM, is its independence on the so-called "Voronoi bias". La Valle [34] describes this bias as an advantage of RRT, as it allows rapid expansion into unexplored areas — with respect to the Voronoi diagram of the RRT tree in the environment, larger Voronoi cells belong to the boundary cells of the RRT tree. Therefore, due to uniform sampling and nearest neighbor selection, expansion is more likely to target such a cell. However, this bias also steers the tree into open spaces and reduces the probability of passing through narrow passages. A similar problem has already been described for PRM with uniform sampling in Section 3.2.2.

This algorithm was further improved by Janoš et al. [5]. The open list has been replaced by a set of priority queues — each tree has one priority queue for every other tree. Each node of tree T is part of each priority queue belonging to T, nodes in the prority queues are sorted according to proximity to the target goals. In each iteration, one priority queue is randomly selected, and its top element is expanded with probability b ("goal bias"). With probability $1 - b$ an expanded node is selected from this queue regardless to order in the queue. In addition, nodes are after k unsuccessful expansions removed from all priority queues and added to the closed list. When the priority queues are empty and the targets are still not

Algorithm 4 Space Filling Forest Star — general overview

Input: q_{init}, q_{goal} — initial and goal configurations
Input: N_{max} — maximum number of iterations
Output: $T(V, E)$ — tree grown from q_{init} towards q_{goal}

```
 1: $V \leftarrow q_{init}$
 2: $E \leftarrow \emptyset$
 3: $O \leftarrow q_{init}$                                          ▷ open set
 4: $C \leftarrow \emptyset$                                         ▷ close set
 5: for $i = 1 \ldots N_{max}$ do
 6:     if $O \neq \emptyset$ then
 7:         if RANDOM(0,1) $\leq b$ then
 8:             $e \leftarrow$ node from $O$ closest to $q_{goal}$
 9:         else
10:             $e \leftarrow$ random node from $O$
11:     else
12:         $e \leftarrow$ random node from $C$
13:     $q_{new} \leftarrow$ EXPAND($e$,$O$,$C$)                      ▷ Algorithm 5
14:     if $q_{new} = \emptyset$ then
15:         continue
16:     $V \leftarrow V \cup q_{new}$
17:     $E \leftarrow E \cup \{(e, q_{new})\}$
18:     if distance of $q_{new}$ to $q_{goal} \leq d$ and ISFREE($q_{new}$, $q_{goal}$) then
19:         $E \leftarrow E \cup \{(q_{new}, q_{goal})\}$
20:         break
21: return $T$
```

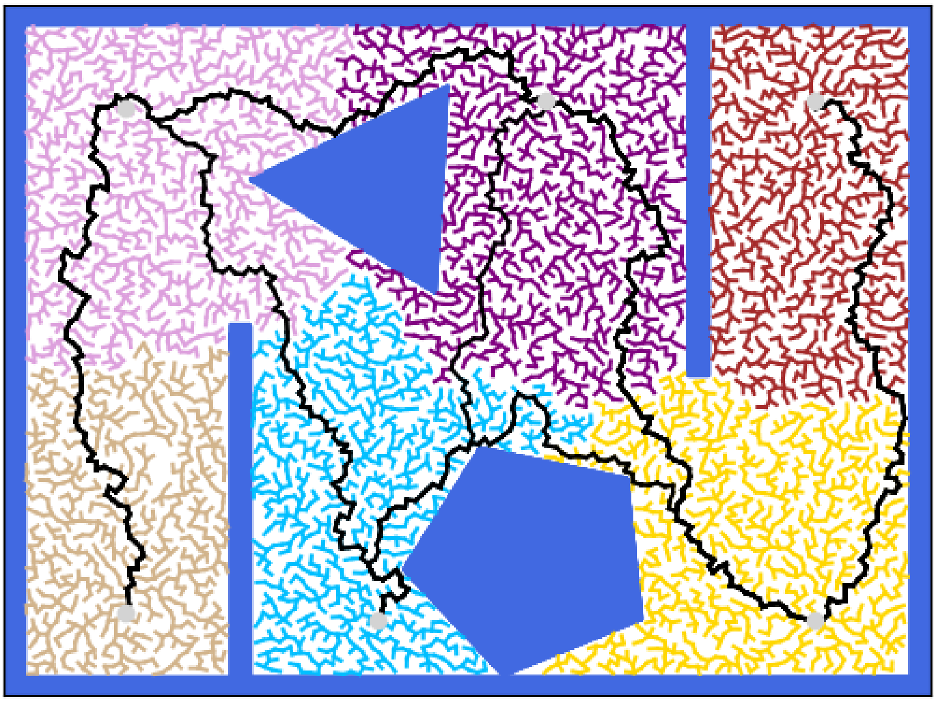


Figure 5.1: SFF for multi-goal planning

connected, further attempts are performed to expand the nodes from the closed list until the iteration limit is reached. Candidate pairs are also recorded for connection of two adjacent trees, and the one pair providing the shortest path is not selected until the full expansion of trees. This method will be further denoted by NR-SFF*.

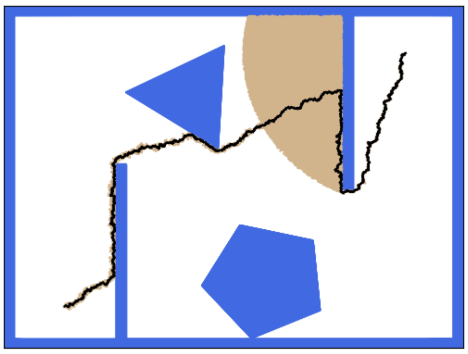
Although this method is designed for multi-goal path planning, it can be easily modified for single-goal planning. Instead of multiple trees, only one expands towards the goal. For bidirectional approach, two trees might be grown (from q_{init} and from q_{goal}), in a similar way as for RRT and EST.

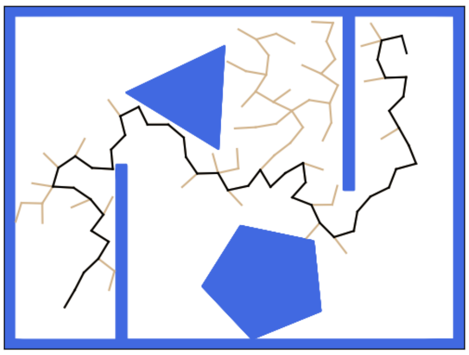
The effect of settings d and b on the resulting path and the NR-SFF* roadmap is shown in Figures 5.2. In general, the higher the "goal bias", the less spacious the final roadmap is — Figures 5.2d, 5.2e and 5.2c. This behavior is very advantegeous in high-dimensional environments for its shorter expansion time. The effect on the length of the resulting path is rather opposite. Additionally, this problem can be solved by increasing the sampling distance (Figure 5.2b) and applying any kind of path smoothing.

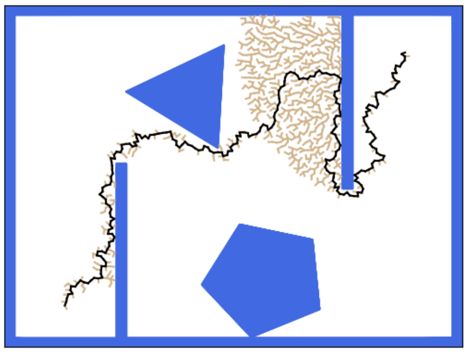
An alternative to path smoothing is to utilize rewiring techniques known from the asymptotically optimal planner RRT* (Section 3.2.3). Instead of joining the new sample q_{new} to the expanded node as in the NR-SFF* algorithm, k_{RRG}-nearest neighbors X are found. The sample q_{new} is connected to the neighbor q_{neigh} so that the length of the path from the root of the expanded tree to q_{new} is as short as possible, i. e., the paths through other retrieved neighbors would be longer. In addition, when the path from the root of the expanded tree to some neighbor q_{neigh} in X through q_{new} is shorter than the existing connection of q_{neigh} with the root, the existing connection of q_{neigh} with its parent node is discarded and q_{neigh} is reconnected to the tree via q_{new}.

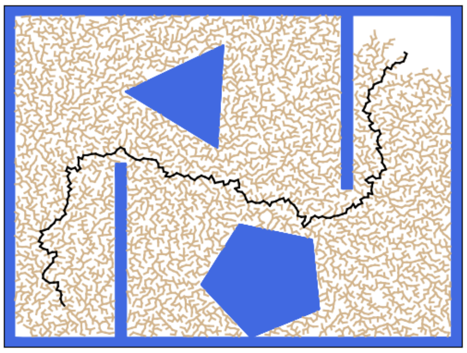
This algorithm was named "Space Filling Forest Star" (SFF*) and is recorded in Algorithm 4. The expansion phase of selected node is then described in Algorithm 5. The flow of both methods has already been described in the previous paragraphs. In addition, it should be noted that the function `cost(q)` stands for the length of the path from the root of the expanded tree to q and the function `cost(p,q)` stands for the length of the direct connection of nodes p and q, as described in the previous paragraph.

The outputs of SFF*, the roadmaps and the resulting paths, are depicted in Figure 5.3. As shown in Figure 5.3a, without any goal bias the final path is very close to the optimal path, but it took significantly longer time to generate the roadmap. This roadmap is more spacious than the roadmap in Figure 5.3b, although the final path is longer. Again, one of the solutions is to increase the sampling distance e. g., as in Figure 5.3c.


(a) : $d = 0.2,\ b = 0.95$


(b) : $d = 5,\ b = 0.95$


(c) : $d = 1,\ b = 0.95$


(d) : $d = 1,\ b = 0.1$

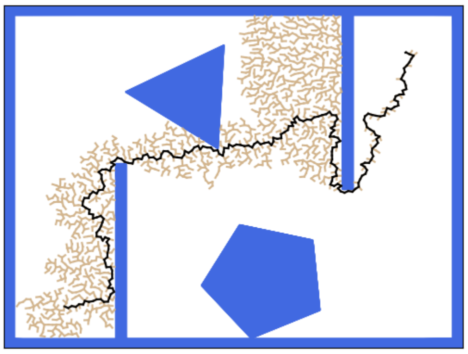
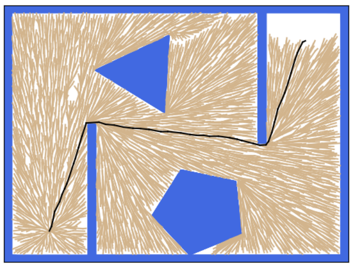
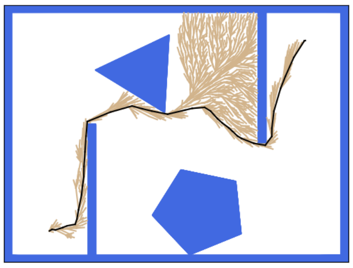

(e) : $d = 1,\ b = 0.5$

Figure 5.2: Comparison of effect of different settings for NR-SFF*


(a) : $d = 1,\ b = 0$


(b) : $d = 1,\ b = 0.95$

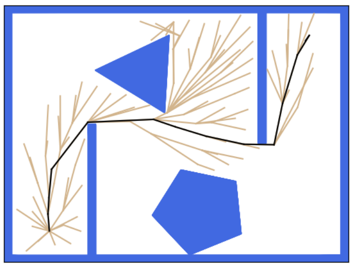

(c) : $d = 5,\ b = 0.95$

Figure 5.3: Tree grown by SFF* with different parameters

Algorithm 5 Space Filling Forest Star — expansion

Parameter: W — environment
Parameter: d — distance between samples
Parameter: b — “goal” bias
Parameter: k — minimum trials before removal of a node from open list
Input: e — node to expand
Input: O — open list
Input: C — closed list
Output: q_{new} — expanded node

```
1: for $i = 1 \dots k$ do
2:     $q \leftarrow$ random condiguration in $W$, in distance $d$ from $e$
3:     if isFree(e,q)and closest node to $q$ in $T$ is $e$ then
4:         $q_{new} = q$
5:         break
6: if $q_{new} = \emptyset$ then
7:     $O \leftarrow O \setminus e$
8:     $C \leftarrow C \cup e$
9:     return $\emptyset$
10: $O \leftarrow O \cup q_{new}$
11: $X \leftarrow$ nearest neighbors of $q_{new}$
12: for $\forall q_{neigh} \in X$ do
13:     if isFree($q_{neigh}$,$q_{new}$) and cost($q_{new}$)>cost($q_{neigh}$) + cost($q_{neigh}$,$q_{new}$) then
14:         set $q_{neigh}$ as parent of $q_{new}$ and update its cost
15:     if isFree($q_{new}$,$q_{neigh}$) and cost($q_{neigh}$)>cost($q_{new}$) + cost($q_{new}$,$q_{neigh}$) then
16:         set $q_{new}$ as parent of $q_{neigh}$ and update its cost
17: return $q_{new}$
```

Chapter 6

Implementation of the algorithms

The presented papers [4] and [5] completely cover the Space Filling Forests for multi-goal planning in 2D and 3D Euclidean space. However, neither of them analyze or even consider utilization of SFF-based algorithms for single-goal planning as well as the single-goal and multi-goal planning for Dubins vehicles and planning on polynomial trajectories, i. e., trajectory planning with consideration of vehicle's dynamics. This gap is filled in following sections.

The algorithms were implemented both in Open Motion Planning Library [73, 74] for the single goal planning, which is described in the first section, and in custom planning library for the multi-goal planning and planning in complex configuration spaces, which is described in the second section.

6.1 Open Motion Planning Library

The OMPL software project contains implementations of many algorithms for sampling-based single-goal motion planning, including the algorithms presented in Section 3.2 — RRT, PRM, EST with its variants — and others. All of these planners might be moreover used in various workspaces, including 2D and 3D Euclidean space and 2D "space" for planning for Dubins car.

The OMPL is distributed alongside a wrapper named OMPL.app that provides an user interface, including graphical interface, for the API of OMPL. Apart from that, it also contains the FCL library [75] for collision checking, as the pure OMPL does not embed any [74]. Additionally, the OMPL.app offers a benchmarking module [76], which enables easy testing of multiple planners under the same conditions. The results of benchmarks can be plotted via "PlannerArena" web interface[1] providing fast comparison of tested planners.

The implementations of NR-SFF* and SFF* algorithms are based on the existing RRT and RRT* implementations of OMPL, and fully corresponds to the algorithms described in Algorithm 4 and Algorithm 5. Thanks to the modularity of OMPL, most of the functions are provided by its API, except for some specific methods for sampling, e. g., sampling in

[1]Interface with simple examples available at `http://plannerarena.org/`.

the specific distance from a given point. No additional modifications were made also for generation of Dubins maneuvers and corresponding planning with Dubins car, everything was already part of the API. Two examples of final paths generated by OMPL implementation of SFF* are visualized in Figure 6.1.

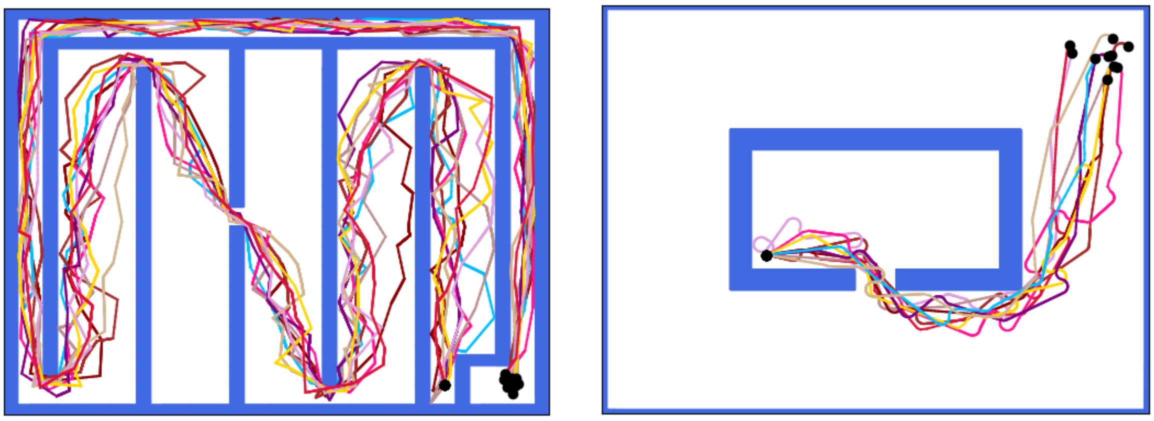

(a) : Ten runs of SFF* in Maze problem **(b) :** Ten runs of Dubins SFF* in Bugtrap problem

Figure 6.1: Examples of problems solved in OMPL

Due to the strict guidelines of authors of the OMPL on new code the SFF* implementation have not been published yet. The publication would probably capture more attention of motion planning community towards Space Filling Forests and is therefore planned in near future.

6.2 Custom library of motion planning algorithms

Although the OMPL would be the preferred variant for implementation of all proposed algorithms, it does not support multi-goal planning, and even for the single-goal planning its API does not cover some of the required methods, e. g., generation of 3D Dubins maneuvers. Even though such methods could have been implemented into the API and used thanks to its modularity also with other planners for benchmarking, credibility of such comparison is questionable. The correct behaviour of the planners is not guaranteed. On account of these reasons, a custom library of single-goal and multi-goal planners was designed.

This library is based on the original library used in [5] and publicly available on GitHub[2]. The original library was completely rebuilt, optimized and extended with other planners and operational spaces. The main language is still C++, used standard is C++17, and the source code is compiled with g++ compiler[3]. The code was written to be in compliance with requirements on object oriented programming (OOP) and is therefore divided into separate

[2]`https://github.com/ctu-mrs/space_filling_forest_star`
[3]Available at `https://gcc.gnu.org/`.

hierarchical classes encapsulating main code of the library. Additionally, to reduce copying of the code for similar datatypes (typically structures specific for particular problem dimension), the C++ class and function advanced templating principles were applied.

The library now allows planning with SFF, NR-SFF* and SFF* algorithms with extensions explained later in this chapter, sPRM and PRM*, RRT and RRT* (for single-goal planning only), Multi-T-RRT [52] (for multi-goal planning only) and Lazy TSP with RRT or RRT* as low-level planners (for multi-goal planning only). All the single-goal planners are able to plan in 2D and 3D Euclidean spaces (these are covered by OMPL and would not be further tested), to plan for Dubins car (covered by OMPL too) and Dubins airplane model, as well as to plan polynomial trajectories in 2D and 3D. All multi-goal planners are able to plan in 2D and 3D Euclidean spaces; SFF, NR-SFF*, SFF*, sPRM, PRM* and Lazy TSP can additionally plan for Dubins car and Dubins airplane models.

6.2.1 Input and output of the library

The type of used planner along with other settings must be specified in the input configuration file. Its structure must comply with the YAML language[4]. For decoding of the file, the yaml-cpp library[5] was integrated. Precise requirements of the configuration file and list of all possible modifications can be found in the Appendix B. Minimal specification must in all cases include type of the planner, maximum number of iterations, robot model file, sampling/expansion limits (or at least enabling *autorange* settings for problems with a map of obstacles), at least one city and a goal for single-goal planning or at least two cities for the multi-goal planning, distance of smallest step for the local planner, sampling distance and maximum distance between two trees. Some output file should be also specified, as only the progress of solution is printed out to the standard output file.

The model files (either models of input environment or robot model, or output tree and roadmap file) comply either with OBJ standard[6], or are in a custom MAP format — files with ".tri" extension.

In the OBJ file, each row is prefixed with a character specifying type of the row. The library recognizes 4 types of rows: "o" for object row (header with object's name, for start of description of a new object), "v" for vertex row (coordinates of the vertex follow), "l" for link row (two numbers specifying order numbers of vertices in the link, i. e., graph edge) and "f" for facet row (three numbers specifying order numbers of vertices in the facet). This type of file is well suited for visualization in third-party software, e. g., in Blender[7], into which it can be easily imported.

[4]Specification and libraries available at `https://yaml.org/`.

[5]Available at `https://yaml-cpp.docsforge.com/`.

[6]Full description, history and actual standard available at `https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml`.

[7]Available at `https://www.blender.org/`.

The MAP file typically begins with a header with file type (Map, Roadmap, Frontiers, etc.) and dimension of the output (2D, 2DDubins, 2DPolynom or 3D). In 2D files, a point is defined by 2 coordinates, in 2DDubins and 2DPolynom files it is defined by 3 coordinates, and in 3D files it is defined by 6 coordinates. The coordinates are delimited by a space. On each row of the file, an object is specified. The object might be either a single point, a line defined by 2 points, or a triangle defined by 3 points. The points are delimited by a space as well. The type of the object is therefore dynamically deduced from the length of the row. This type of file is ideal for postprocessing or visualization in custom scripts, e. g., in Python. One of the possible implementations of script in Python language using the Matplotlib[8] library for plotting of the library's output files is available as a part of the planning library.

6.2.2 Sampling

As all of the proposed and implemented planners are *sampling-based*, a module for sampling is an important part of the library. Although the problem of sampling itself is quite specific for particular planner, most of them share common basis — uniform sampling.

First of all it is important to notice that computer generated random numbers are not truly random, usually some linear congruential generator is used for their generation. For basic generators and generators with poorly chosen parameters the results might be periodic and some regularities in samples might appear [6, 77]. One of the solutions is to use Mersenne Twister algorithm [6, 32, 78]. For this library the Mersenne twister algorithm from the standard C++ library was selected.

For each of the available problem dimensions, two methods had to be specified: a method for sampling a point in the space for RRT-based and PRM algorithms and a method for sampling a point in a given distance from its parent for all SFF-based algorithms. In both cases, limits of the environment must be satisfied.

Sampling in space

The first method is quite straightforward for all dimensions. The position of point in both 2D and 3D is sampled uniformly — a random number per each axis is generated — in the given ranges of the environment.

The yaw angle for configurations of Dubins car in 2D and Dubins airplane in 3D is sampled uniformly in range $[-\pi, \pi)$ and the pitch angle for Dubins airplane in 3D is sampled uniformly in the range specified by the configuration file of the problem. However, the sampling of rotation in 3D space is different: as Kuffner [15] emphasizes, a naïve approach to sampling using Euler angles might lead to non-uniform distribution of samples. To overcome this issue, a special methods must be followed, or unit quaternions instead of Euler angles used. For

[8]Available at `https://matplotlib.org/`.

this library, the latter case was applied — the rotation is given by an unit quaternion, which is randomly sampled by the method described in [15].

The velocity for planning on polynomial trajectories has normal distribution with mean equal to average velocity (computed from the sampling distance and segment time) and standard deviation equal to half of the average velocity. This measure should limit sampling of unreasonable states slowing down the movement, or oppositely of states being infeasible due to unachievable speed. The acceleration is then sampled uniformly in given ranges — minimum and maximum thrust. The sampling in cone similar to [51] was not chosen, because the proposed SFF* algorithm aims more on planning in narrow passages where the search cannot be limited only on the direction towards the goal.

Sampling in a distance from parent

The sampling in a given distance d differs for each of the dimensions. For Euclidean 2D spaces, the polar method was implemented — a random angle is sampled and the point in the distance d and this direction is selected if it lies in the bounds of the environment, otherwise the procedure is repeated. When planning for the Dubins car, a temporary point P is sampled first. Its position is determined the same way as for the Euclidean 2D space, yaw angle is uniformly sampled in the interval $[-\pi, \pi)$. In the next step, a Dubins maneuver is planned to P, and the resulting point R is interpolated on this maneuver so that the length of the segment from P to R is equal to d.

The principle of procedures for all 3D spaces is similar — a temporary point P is sampled first, then the resulting point R is interpolated on the computed segment.

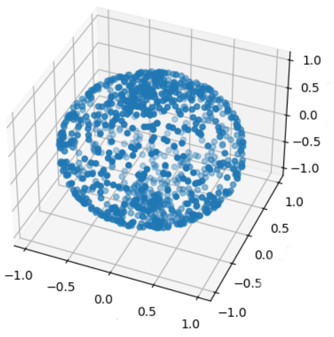
Finding position of P in the distance d in 3D leads to the uniform sampling of points on 2-sphere [79]. Three possible sampling methods are summarized in Algorithm 6. The most straightforward way would be to use the polar method with 2 random angles sampled uniformly (procedure `Polar-wrong`). However, the distribution of points on the surface would not be uniform — most of the points would be concentrated near poles, as the area on the sphere corresponding to the cumulative distribution function [79] grows according to the *cosine* function. This situation is depicted in Figure 6.2a. To avoid it, the random angle θ must be sampled according to *arccos* distribution function (procedure `Polar-right`). The angles θ and ϕ are interchangeable, switching of these angles would lead to non-uniform sampling as in Figure 6.2b. Another possibility is to use so called "Muller" method, named after its author [80]. A random point in an unit cube is sampled, this point is projected on a unit sphere and scaled to distance d. The result is depicted in Figure 6.2c. For the planning library, this method was selected.

The rotation of P in Euclidean 3D space is represented by an unit quaternion, as described in [15]. This quaternion is sampled uniformly with the method given in [15]. For the planning for the Dubins airplane, the yaw angle of P is sampled uniformly in range $[-\pi, \pi)$ and the

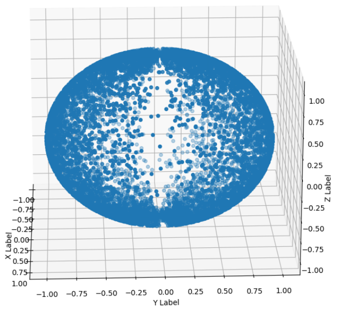
Algorithm 6 Methods for sampling on 2-sphere

Input: r — radius of the sphere

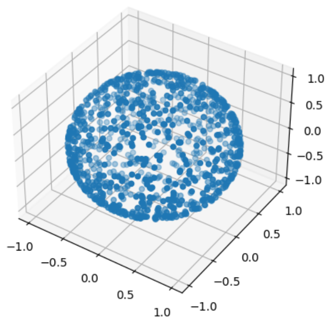
```
1: procedure POLAR-WRONG
2:     $\theta \leftarrow$ random angle in range $[-\pi;\ \pi)$
3:     $\phi \leftarrow$ random angle in range $[-\pi;\ \pi)$
4:     $x = r \cdot \sin\theta \cdot \cos\phi$
5:     $y = r \cdot \sin\theta \cdot \sin\phi$
6:     $z = r \cdot \cos\theta$
7:     return $Point3D(x,\ y,\ z)$
8: procedure POLAR-RIGHT
9:     $u \leftarrow$ random number in range $[0,\ 1)$
10:    $\theta = \arccos(2u - 1)$
11:    $\phi \leftarrow$ random angle in the interval $[-\pi;\ \pi)$
12:    $x = r \cdot \sin\theta \cdot \cos\phi$
13:    $y = r \cdot \sin\theta \cdot \sin\phi$
14:    $z = r \cdot \cos\theta$
15:    return $Point3D(x,\ y,\ z)$
16: procedure MULLER
17:    $u \leftarrow$ random number in range $[0,\ 1)$
18:    $v \leftarrow$ random number in range $[0,\ 1)$
19:    $w \leftarrow$ random number in range $[0,\ 1)$
20:    $n = \sqrt{u^2 + v^2 + w^2}$
21:    $(x,\ y,\ z) = r \cdot (u,\ v,\ w)\,/n$
22:    return $Point3D(x,\ y,\ z)$
```



(a) : Wrong method 1 — high density around poles



(b) : Wrong method 2 — switched distribution of angles



(c) : Correct approach for uniform sampling

Figure 6.2: Sampling on sphere — wrong and correct approaches

pitch angle is sampled uniformly in the range given by the configuration file. The rotation of the robot for planning on polynomial trajectories is embedded into the velocity and acceleration vectors.

The velocity and acceleration for planning on polynomial trajectories is determined the same way as for the sampling in space (Section 6.2.2), i. e., the velocity is sampled according to normal distribution and the acceleration is sampled uniformly.

6.2.3 Collision checking

One of the most important and computationally expensive parts of the sampling-based algorithms is the collision detection.

Robots and obstacles might be usually decomposed into simpler primitives, even when they are not convex. As Jiménez et al. [81] describe, many algorithms for detection of interferences of such primitives exist. One of the most basic is e. g., direct computation of distance between two sets of points [6].

The collision detection might be significantly speeded up by utilization of hierarchical volume representations. This way the object is transformed into a tree, so that each vertex of the tree represents a bounding volume that contains some subset of the object and allows faster computation of the interference than with the whole object itself. An union of children objects' subsets must be moreover equal to the subset represented by their parent. The root of such tree represents bounding volume of the whole object.

Each interference query then proceeds from the root of such tree and, in case the bounding volumes of examined objects overlap, continues recursively downto bottom-most leaves. The bounding volumes might be represented by spheres, axis-aligned bounding boxes, or oriented bounding boxes, which perform best (at least in dense, cluttered environments) [81].

Implementation-wise it is desirable to utilize existing libraries for collision detection, e. g., RAPID library [82], CGAL [83] or FCL [75]. For this library of motion planning algorithms, the RAPID library[9] was selected.

6.2.4 Local planner

Not only the sampled nodes, but also the transitions between them must be collision-free. This task is solved by so called "local planner".

One of the approaches is the *incremental planning*, where small steps are incrementally taken from the initial sample towards a goal sample, and for each step possible collisions of robot's configuration with surrounding environment are checked. The *binary approach* applies methods of binary search on the transition, i. e., the middle configuration is checked first, then the motion is divided into two halves etc. Another possible planner, *line planner*,

[9]Available per request at `http://gamma.cs.unc.edu/OBB/`.

combines both approaches: first the initial and goal configurations are checked and the rest is examined in binary manner [32].

In this custom library of planning algorithms, classic incremental local planner was implemented.

6.2.5 Nearest neighbors

All of the following algorithms search typically in one of their computational steps for nearest neighbors of some point in space. It is of course possible to use exhaustive iteration through all samples with complexity $\mathcal{O}(n)$, favourable is to use some specialized datastructures, like "k-d trees" [84], where the average time complexity of search for M neighbors in n samples is $\mathcal{O}(M \dot{\log} n)$.

Furthermore the results do not have to be precise (sampling-based algorithms are still some kind of heuristics), therefore some randomized approaches might be used. One of them is implemented in the library FLANN [85] used also in this library of planning algorithms[10]. The implemented algorithm uses multiple random k-d trees to obtain best performance of search for nearest neighbors.

6.2.6 Generation of Dubins maneuvers

The 2D Dubins maneuvers are generated using the opendubins library[11], which was already introduced in the Section 3.3. This library offers methods both for finding the optimal maneuver from point A to point B as well as for interpolation on this maneuver. These are the only required methods. Moreover the library is written in C++, so the process of embedding was possible without any significant modifications of the code.

On the contrary, direct integration of the library for generation of 3D Dubins maneuvers was not possible. The selected library[12] is written in Julia language. To embed it into the project, the code was rewritten to C++ and added to the library for generation of 2D Dubins maneuvers. The custom merged version of libraries is available at `https://github.com/jarajanos/gdip`. One of the advantages of this solution is the utilization of the part for generation of 2D maneuvers, as the computation of 3D maneuvers requires multiple computations of 2D maneuvers — precise principle was already described in Section 3.3.1.

6.2.7 Generation of polynomial segments

The code for the generation of polynomial segments is based on [49], i. e., the aim of the planning is to minimize jerk of the vehicle rather than the length of final trajectory. From

[10]The FLANN library is available at `https://github.com/flann-lib/flann`.
[11]Available at `https://github.com/comrob/gdip`.
[12]Available at `https://github.com/comrob/Dubins3D.jl`.

code available on the official GitHub repository[13] a static library was created[14], which is more suitable for deployment on this library. Moreover the generation of trajectories in 2D was added into this library, in which only 2 axis instead of 3 axis are considered.

An input to this trajectory generator consists of three vectors (representing position, velocity, acceleration) for both start and goal configurations, a vector of gravitational acceleration and of time duration of the trajectory. This time is computed from the Euclidean distance of positions of both configurations and from the average velocity. An output forms three real constants (α, β, γ) for each axis, defining the polynomial shape of the trajectory. The principle was described in Section 3.4. The generator library also contains methods for interpolation of configurations on the trajectory and checking of feasibility in terms of vehicle's dynamics.

6.2.8 Integration of TSP solvers

The aim of the multi-goal path planning problem is to search for the shortest tour between multiple goals — the Traveling Salesman Problem and its solution is therefore its indivisible part. As some of the planners need multiple solutions of TSP throughout the algorithm flow and others need it at least in the end when the cities are connected with found paths, the planning library integrates an interface for calling of external TSP solvers. In actual version, LKH and Concorde solvers are supported — both were already introduced in Section 4.2. Both might be automatically downloaded and installed to a default location via Makefile commands, or path to an external location might be specified in the configuration file.

The LKH solver is downloaded and installed from its official site[15]. As an input the standard TSPLIB file is accepted, type of the solved problem might be either TSP or ATSP. The standard Euclidean problems are symmetric, therefore they are naturally well-suited for TSP, while the problems with Dubins vehicles and even mutliple possible paths between single nodes (due to multiple heading angles) fall into category of Generalized Asymmetric Traveling Salesman Problems (GATSP). The GATSP is then converted to ATSP with Noon-Bean transformation, which is implemented in this library as well. The processing of results is straightforward, as LKH saves into the result file only the order of cities in the tour. When solving ATSP, the order has to be back-converted to GATSP, getting not only the order of cities, but also the optimal heading angle for each city.

The Concorde solver is quite old but still efficient TSP solver. It is automatically downloaded from the official site[16] as well and, as it contains deprecated calls to external API, it is automatically patched and installed. Contrary to LKH, the Concorde needs commercial

[13] Available at `https://github.com/markwmuller/RapidQuadrocopterTrajectories`.
[14] Available at `https://github.com/jarajanos/RapidQuadrocopterTrajectories`.
[15] More information at `http://webhotel4.ruc.dk/~keld/research/LKH-3`.
[16] More information at `http://www.math.uwaterloo.ca/tsp/concorde/`.

CPLEX optimizer API[17] installed on the target machine, location of its libraries has to be specified in the Makefile of the planning library. Another difference is the type of accepted input, as it only solves (symmetric) TSP and not ATSP. When solving ATSP, this problem has to be converted to TSP — algorithm for the conversion is part of the planning library as well. The processing of the result is done in a similar way as for the LKH.

6.2.9 Implementation of planners

Solver class

In order to unify the implemented planners mentioned in the introduction of Section 6.2 as much as possible, all of them inherit from "Solver" class. This class embeds common structures used for planning like description of the problem, random generator for sampling of nodes or integrated TSP solver. It also owns all instances of expanded trees.

Apart from these structures, the Solver class implements the local planner for all supported dimensions. For Dubins maneuvers it is also tested, that all points lie in given bounds of the environment and for polynomial trajectories also their feasibility in terms of dynamics.

Other methods procure finding of all connected trees, getting all paths from them and also combining of these paths together to link unconnected pairs of cities via other cities. The Floyd-Warshall algorithm is utilized for this purpose. Common parts of paths are additionally removed to shorten the total length.

The Solver class also implements common methods for export of cities, expanded trees, paths and final TSP tour in OBJ or MAP formats as well as saving of parameters of achieved solutions or TSP table for external TSP solvers in commonly used TSPLIB format [86].

SpaceForest class

The most fundamental part of the library is the family of Space Filling Forest algorithms implemented in the "SpaceForest" class. The class covers both SFF and newer NR-SFF* switchable by the value of priority bias (setting bias to 0 enables SFF, any other bias enables NR-SFF*) as well as their improved version SFF* which is enabled via "optimize" setting in the configuration file.

Contrary to the original version of library presented in [5] the code was significantly optimized, especially the implementation of NR-SFF* and SFF*. Thanks to a change of order of some methods described later in the following paragraphs, the runtime of SFF algorithms was reduced.

The SFF and NR-SFF* versions differ mostly in the structural representation of frontiers, i. e., nodes for expansion. The old SFF uses common vector of nodes for all trees, whereas for NR-SFF* each of n cities uses $(n-1)$ priority queues for each of their neighboring city.

[17] Available at `https://www.ibm.com/analytics/cplex-optimizer`.

Each priority queue keeps its own order of nodes[18] in open list closest to the particular target city. In each iteration a random queue is selected for a random tree, and with probability p its top node is expanded, or with probability $(1 - p)$ a random node from the queue is expanded. For the representation of priority queue a custom implementation of binary heap is utilized, supporting not only standard push and pop operations, but also access and popping of random elements in the heap.

After selection of a node for expansion a random configuration is sampled in given distance and it is checked, that it lies in the bounds of the environment. In following step, the growth back into the tree is tested by searching for nearest neighbors, using the aforementioned FLANN library. Next, the feasibility of the trajectory is checked by the local planner. The order of last two steps is crucial for performance of the library, as the local planner is significantly more demanding, due to mutiple collision checks.

Another important step is linking of expanding trees. When two trees approach each other, i. e., the new node is close enough to a node from another tree, a reference to both nodes is saved, without any feasibility check of path between these nodes. This check is done after complete expansion, not until the links are sorted by the length of resulting path from shortest to longest.

The last step of the expansion process before addition of the new node to the tree is RRT-rewiring, for SFF* only. Its principle was already described in Chapter 5.

In case when one of the checks fails, the new configuration is discarded and next expansion is attempted, up to k-times, where k is given maximum number of misses. After k misses, the expanded node is removed from all priority queues (or from the frontier structure) and inserted into the closed list.

The end of the expansion phase depends on the “connect-only” setting. If this option is enabled, the expansion is finished after connection of all trees — this is checked by finding weakly connected components of the created graph. If the “connect-only” option is disabled, the open list for expansion (i. e., all priority queues) must be empty and the trees must be connected. If they are not connected despite the emptiness of open list, the expansion continues from the closed list. This option is by default disabled.

The situation is different for multi-goal planning for Dubins vehicles. The decoupled approach was selected as a solution of the DTSP problem, therefore multiple yaw angles are considered for each of the given cities. In order to maximize the number of possible paths for each angle, multiple paths are allowed between the city node and its children. The 2D table of distances had to be therefore expanded to four dimensions (instead of city-city pairs, city-city-angle-angle quadruples have to be considered). Moreover, as the DTSP is not symmetric, i. e., path from A to B is not equal to path from B to A, paths back and forth would have to been checked normally, which could have potentionally eliminated some

[18] Only references to nodes are stored in the queues.

solutions in narrow passages. Instead of this approach, the points of one tree are inverted when getting paths from connected trees, and therefore the same paths are retained although they are flown in opposite direction.

For the polynomial trajectories the base code is modified as well. When checking for nearest neigbors in the expanded or in any other neighboring tree, the 9D distances are considered (not the length of the trajectory). Also the collisions are in this case checked only on the straight-line connection between considered nodes[19]. The reason is that for close nodes, the generated trajectory is long and expensive and therefore the tree would be too dense.

RapidExpTree class

The module "RapidExpTree" is primarily designed to comply with the Multi-T-RRT algorithm presented in Section 3.2.3. When planning only for one goal, the behavior of the implementation corresponds to the standard RRT algorithm. Even though the asymptotically optimal version and priority bias are not defined for Multi-T-RRT, both are available for RRT, i. e., when planning for single goal.

At the beginning a tree is set up for each city node. During every iteration a random tree t_1 is selected (apart from the tree for goal node) and is expanded in RRT manner — a random configuration in space r is sampled (or goal node selected, for single goal with priority bias), closest configuration in the expanded tree n is found (using FLANN) and a new configuration c is interpolated on the path from r to n in given distance. This path is either a straight line, a Dubins maneuver or a polynom. The path from r to c is checked for collisions (using local planner of Solver class) and added to the expanded tree (in case of RRT*, the parent and children rewiring is done as well). If a node of other tree t_2 happens to be located in neighborhood of c, the trees are connected via c and further expanded as one tree. The merging of trees is done via Union-Find structure.

The iteration process ends when all trees are connected together. The resulting paths are extracted only between directly connected trees, rest of them is found with Floyd-Warshall algorithm of Solver class.

ProbRoadMaps class

The "ProbRoadMaps" class is dedicated to sPRM planner. The setting of priority bias as well as settings of sampling and tree distances are ignored, the most important settings are number of iterations corresponding to the maximum number of sampled configurations and number of PRM connections for each configuration. The latter setting may be omitted by choosing PRM* version, for which the number of connections is calculated automatically.

First, the random configurations are sampled in the free configuration space. Standard uniform sampling is applied, with no advanced techniques of sampling. Each configuration is

[19]For the subsequent nodes collisions are still checked on whole trajectories

then connected up to n nearest neighbors (using FLANN), where n is the selected number of connections, after checking feasibility of such connection with the local planner. The paths between cities are found in the created graph using the Dijkstra algorithm, based on the custom implementation of the binary heap.

LazyTSP class

As the name suggests, this module integrates the Lazy TSP planner. Contrary to the previous planners Lazy TSP is purely multi-goal planner and its functionality is tightly bounded with the TSP solver. The TSP solver is therefore mandatory and must be specified in the configuration file. The algorithm's basis planner is standard RRT or its asymptotically optimal version RRT* (switchable via "optimize" option in configuration file), both are implemented without priority bias.

In each iteration, first the TSP is solved with the table of distances between cities. In the first iteration, it is initialized with Euclidean distances between cities. The TSP is solved by generation of temporary TSPLIB file, calling external solver and processing the generated result file. The name of the temporary file is unique and is dependent on the ID of the program's instance, to avoid rewrites or other mismatches. This is important especially for testing when multiple instances of the solver are run in parallel. For each edge from the result single-goal RRT or RRT* is run and distance in the table is updated accordingly. The process is repeated until two calls of TSP solver return the same solution, i. e., the path is the shortest possible. The algorithm also (unsuccessfully) ends, when the limit of iteration is exceeded[20].

LazySpaceForest class

The module "LazySpaceForest" implements a modification of classic SFF for 3D problems with Dubins maneuvers denoted by "Lazy SFF". The problem is solved in decoupled way — at first it is solved as standard 3D problem, and the resulting straight-line paths are converted to Dubins maneuvers. Main purpose of this implementation is to show that this approach is infeasible especially for environments with narrow passages. The configuration is the same as for the SpaceForest module.

First, the problem is converted from 3D Dubins problem to Euclidean 3D problem, which is then solved by standard SFF, NR-SFF* or SFF* planner. From the resulting path waypoints, a Dubins Touring Problem (DTP) is constructed, in which the optimal heading angles are determined. According to the "dubins-resolution" setting of the configuration file, multiple heading angles are sampled for each waypoint, and for each pair of heading angles of two subsequent waypoints, a Dubins maneuver is constructed. In the created graph, the shortest path is determined using the Dijkstra algorithm with custom binary heap.

[20] Contrary to other planners, the limit applies to each call of RRT.

Chapter 7

Benchmarking of the proposed planners

The implementations of NR-SFF* and SFF* in OMPL as well as in the custom library have been extensively tested in almost all dimensions that are not covered by article [5], both in single-goal and multi-goal planning. Specifically, the OMPL implementation was tested in single-goal planning for Euclidean 2D and 3D planning and planning for Dubins car. The implementation in the custom library was tested in single-goal planning for Dubins airplane and planning on polynomial trajectories in 2D and 3D and in multi-goal planning for Dubins vehicles in 2D and 3D. The main purpose of these tests is to compare NR-SFF* and SFF* with existing planners, rather than looking for for optimal settings of SFF-based algorithms — this is in certain extent already covered by [4] and [5].

7.1 Testing in Open Motion Planning Library

The OMPL implementation was compared with all available geometric planners that fall into the same category of planneers as NR-SFF* and SFF* (e. g., are not bidirectional) and have been properly implemented for particular dimension[1]. Precisely, the following planners were used for comparison: BKPIECE, LBKPIECE, KPIECE, EST, FMT, PRM, PRM*, RRT, RRT*, TRRT, SBL, SPARS, SPARS2, SST, STRIDE and for some dimensions also LBTRRT and RRTXStatic.

An integrated OMPL benchmarking module was used for the benchmarks, each configuration (map, start and goal, planner) was run 1 000 times, with variable timeouts according to the difficulty of the specific problem. Some planners usually ended up much earlier, some tried to improve the solution as long as possible (typically PRM based algorithms or asymptotically optimal planners like RRT*).

The tests ran on a DELL G5 15 notebook with Intel Core i7-10750H with 6 cores, each with a core frequency up to 5 GHz, and 12 MB of L3 cache. HyperThreading feature has been disabled to prevent incorrect correlations at runtime for specific planners. The type of

[1]Some of the tested planners might have ended erroneously,e. g., with SIGTERM flag and therefore were excluded from further tests.

the integrated RAM was DDR4 with a frequency 2933 MHz, and its size was 16 GB.

The test results were converted to a database with an integrated converter. Even though this database might be processed by the Planner Arena interface introduced in Chapter 6, the output plots are confusing as Planner Arena does not distinguish between approximate (incorrect, nearest to the goal) and correct solutions and timed-out runs are sometimes wrongly marked as correct solutions. Also, formatting or further processing of output plots is not possible. The output databases with the results of benchmarks were therefore processed using Python and available libraries for data processing — NumPy and Pandas. The graphs were plotted with Matplotlib and Seaborn libraries.

7.2 Testing of the custom library

The benchmarking of algorithms implemented in the custom library was performed differently. For single-goal planning, only RRT, RRT* and PRM* were compared, for multi-goal planning PRM*, Lazy-SFF and Lazy-TSP with RRT and RRT* basis planners were tested.

The configuration files were generated automatically using a Python script, and the library was then launched by another Python script 1 000 times per each configuration. This was done in parallel, typically on 32 threads, with a 30 minute timeout. Unlike the OMPL, each run is limited by a fixed number of iterations, typically 100 000.

The benchmarks were executed on the MetaCentrum grid. MetaCentrum is a Czech virtual organization for sharing the computing resources of its members to Czech academic community[2]. Each group of tests[3] was executed on a different cluster, so it is not possible to specify exactly hardware resources used for the computation, or the results (in particular computation times) should only be interpreted within the scope of each test, not globally. It should also be noted that the total walltime of the tests performed ranges in the thousands of days of computation.

The results of each test were written into single CSV parameter file, then all files were processed using Python and available libraries for data processing — NumPy and Pandas — in the same way as the OMPL benchmarks.

7.3 Maps for problems in 2D space

The 2D testing environments were chosen to correspond as much as possible with benchmarks in [5] and highlight the main advantages of SFF-based algorithms (especially independence from Voronoi bias). The maps for single-goal planning are depicted in Figure 7.1 with start

[2]More information at `https://metavo.metacentrum.cz/`.

[3]By "group of tests" is meant all tests with the same problem dimension and different environments and planners.

and goal points in the default configurations for single-goal planning. The problems were also solved in the opposite direction, starting from the goal point.

For the multi-goal path planning, the same environments as in [5] were chosen, they are depicted in Figure 7.2 along with the default configuration for a setup with 20 cities.

The Bugtrap map shown in Figure 7.1a contains one narrow passage (trap entry), which might be difficult for some planners to overcome. The depicted configuration is called “in” as the robot is trying to get into the bugtrap, the reversed configuration is similarly called “out”. Unlike Bugtrap, the Dense map (Figure 7.1b) is a prototype of a common map with multiple obstacles, and wide corridors. Priority bias or Voronoi bias might be misleading. These “fragilities” of planners should get exhibited in two newly designed maps: Spiral and Maze. The Spiral map (Figure 7.1c), like the Bugtrap map is tested in two configurations, “in” and “out”. Maze map configuration shown in Figure 7.1d is called “corridor”, the inverted configuration is denoted by “walls”.

In all cases the robot has a circular shape, its size was adjusted to fit into narrow passages with sufficient space to find a valid solution even for Dubins car or with polynomial trajectories.

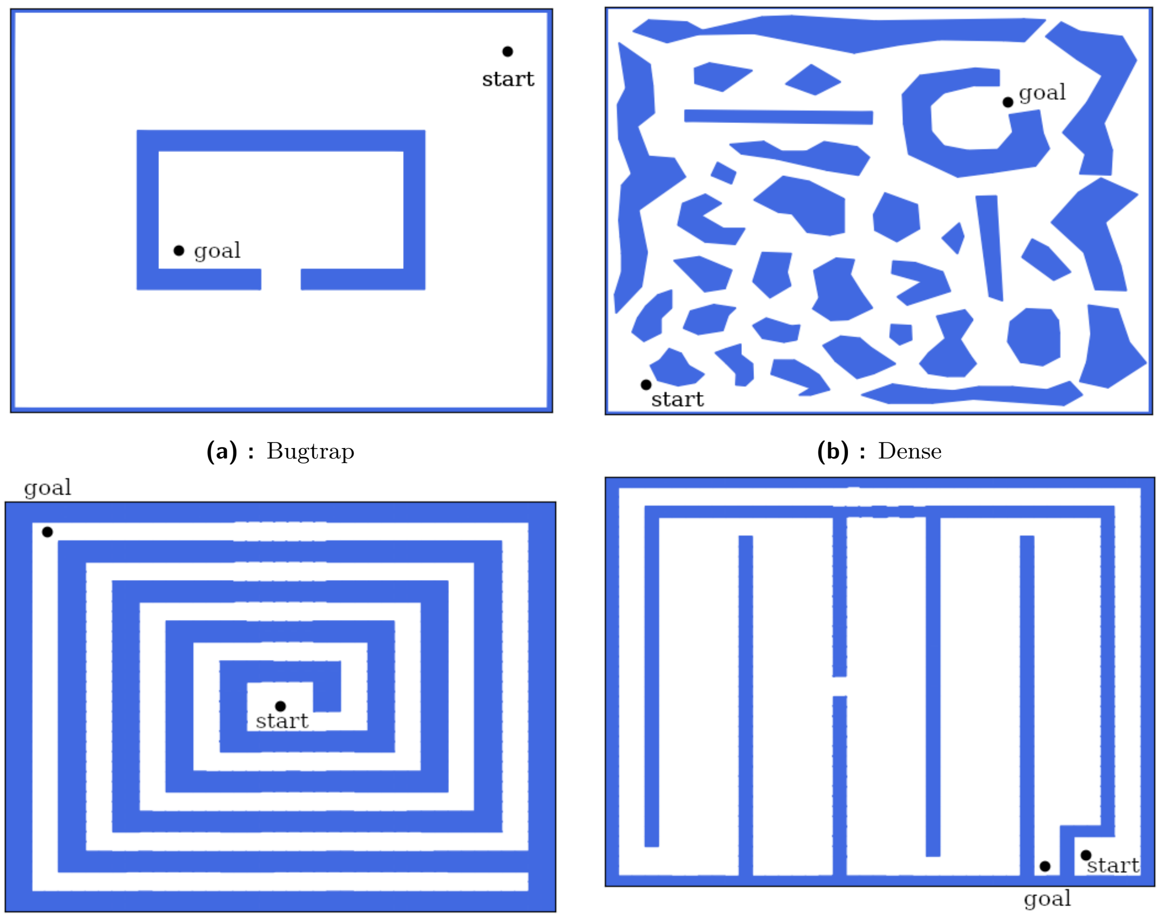
7.4 Maps for problems in 3D space

The selection of 3D environments for benchmarks of implementations in the custom library fully corresponds to the benchmarks executed in [5]. These are depicted in Figures 7.3a, 7.3b and 7.3c along with the start and goal points for single goal planning in 3D.

The environments should also have been used for benchmarking of planners for 3D Euclidean space in the OMPL. However, the migration of the maps was not successful due to unknown issues with the FCL collision library, which did not work properly and the final paths were not collision-free. Therefore the native OMPL.app environments were used for testing, they are depicted in Figures 7.3d, 7.3e and 7.3f. Although it would also be possible to use the OMPL environments for benchmarks of the planners in the custom library, it is impossible to use them for multi-goal path planning. Also, there may not be a solution to the problems of single-goal path planning problems with constrained robot, or it would be difficult to find it in reasonable amount of time. Each of the environments contains some narrow passages and obstacles that might be problematic for planners with priority bias or Voronoi bias, as well as in particular 2D maps.

The Building map shown in Figure 7.3a is inspired by the task of MBZIRC 2020 international competition[4]. The Dense environment (Figure 7.3b) is the transformation of the 2D map of the same name into 3D with the random height and altitude position of each segment. The Triangles map (Figure 7.3c) was inspired by the twodimensional Triangles map presented in [5]. Unlike the Dense 3D map, the height of all segments is the same, and equals to the

[4]More information and actual challenge might be found at `https://www.mbzirc.com/`.



(a) : Bugtrap **(b) :** Dense

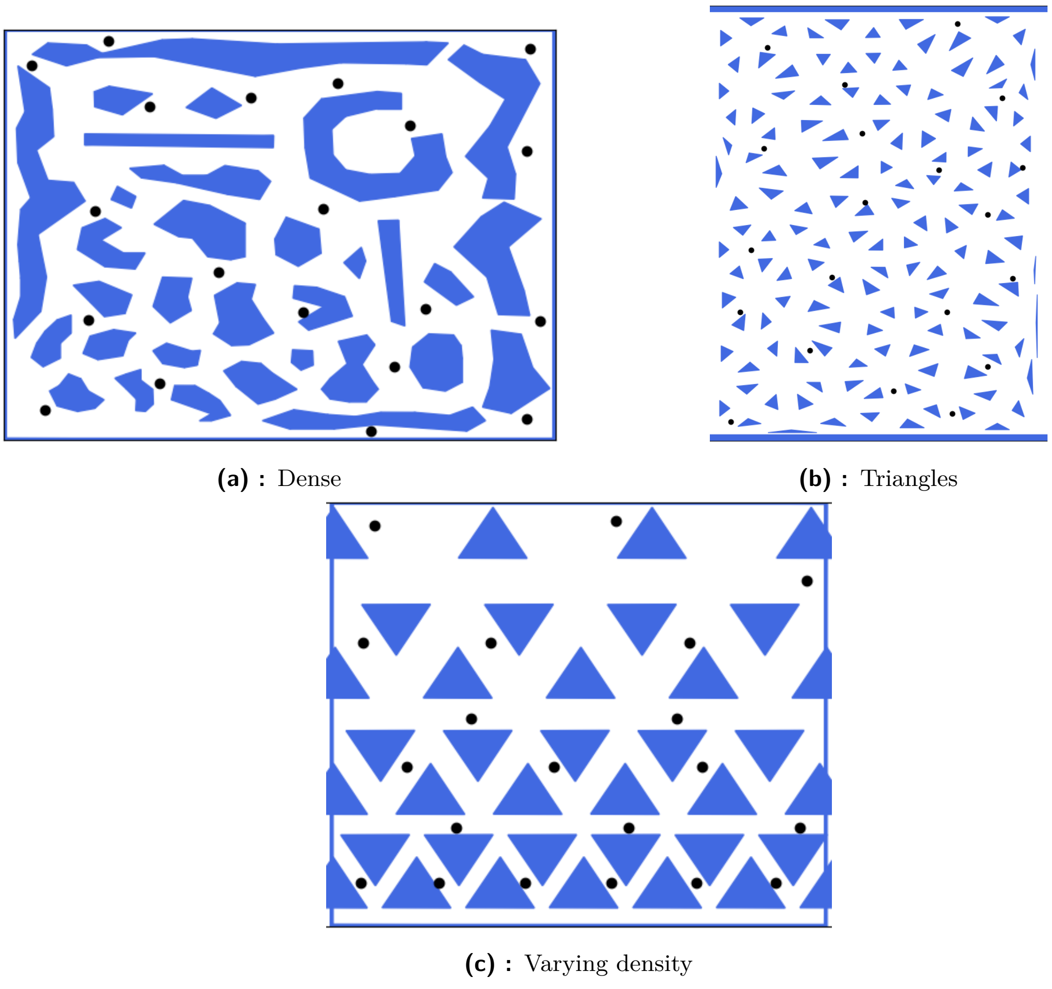
(c) : Spiral **(d) :** Maze

Figure 7.1: Visualization of maps for 2D single-goal problems

height of the configuration space.

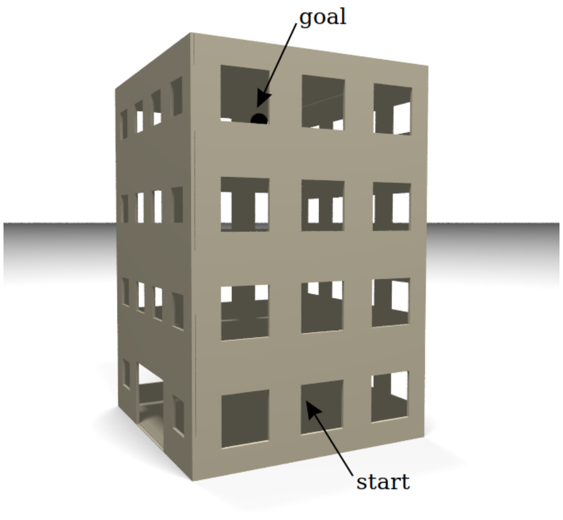
The shape of the robot deployed in the previously introduced environments is a cyllinder, its size has been adjusted individually for each map to fit in narrow passages with enough space, so it was possible to find a valid solution even for Dubins airplane or with polynomial trajectories.

The Bugtrap map depicted in Figure 7.3d along with the robots used, is one of the most famous benchmarking maps. For this configuration, the challenge is to get the robot out of the "hollow case" through a narrow hole. In this case, the rotation of the robot also plays a significant role. On the Easy (Figure 7.3e) and Twistycool (Figure 7.3f) maps the main challenge is similar, as the "complexly-shaped" robots (depicted in the referenced figures) are supposed to get through a rectangular window. The window for the Twistycool map is smaller and the problem is therefore more difficult.

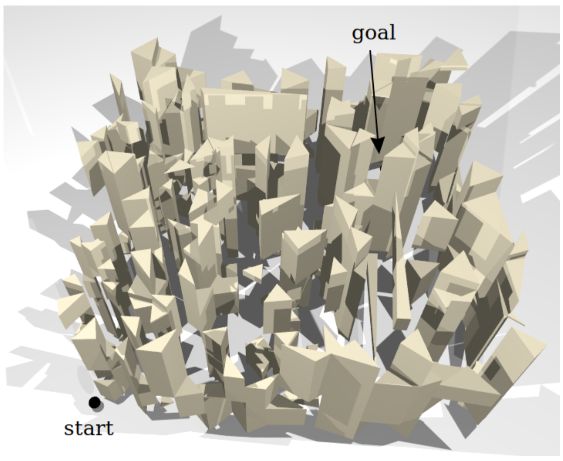

(a) : Dense **(b) :** Triangles

(c) : Varying density

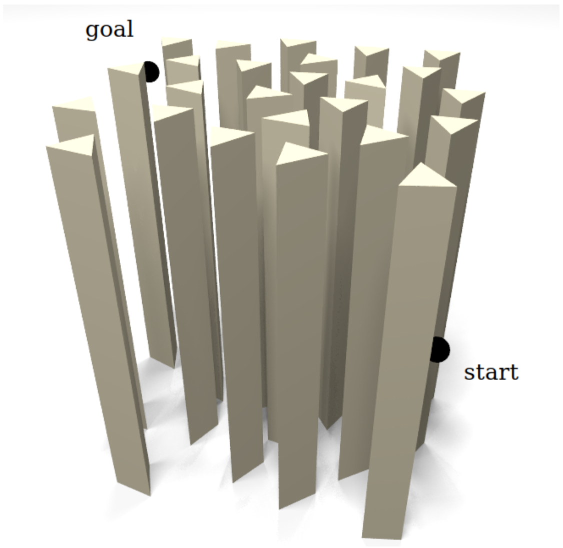
Figure 7.2: Visualization of maps for 2D multi-goal problems



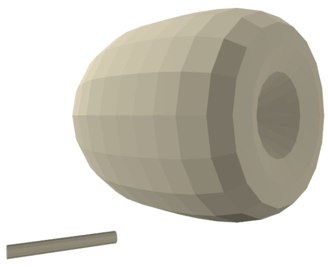
(a) : Building



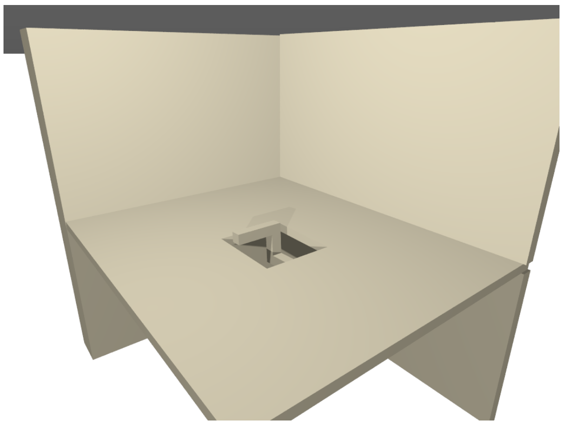
(b) : Dense



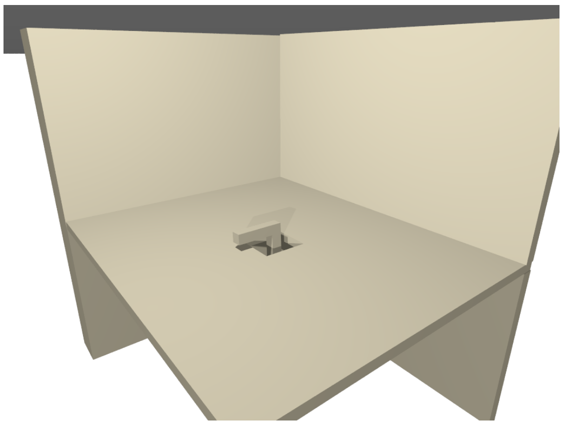
(c) : Triangles



(d) : Bugtrap



(e) : Easy



(f) : Twistycool

Figure 7.3: Visualization of maps for 3D problems

Chapter 8

Space Filling Forest for single-goal motion planning

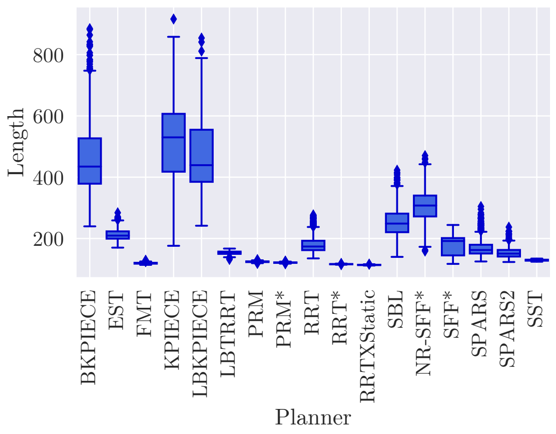
Both implementations of SFF algorithms, in the OMPL and in the custom library, were compared with other available planners, with conditions specified in Chapter 7. As the single-goal motion planning was not analyzed in [4] or in [5], all possible combinations of dimensions and kinodynamic constraints were tested, i. e., standard unconstrained planning in the Euclidean space, planning for the Dubins vehicles and planning with polynomial trajectories, both in 2D and 3D.

In the following sections, complete results of the benchmarks are analysed, but only significant graphs were selected for demonstration of the described features. The complete results in a form of graphs are available in the appendix (Appendix C). In all cases, three parameters were watched: the success rate, time of run and the length of the resulting path.

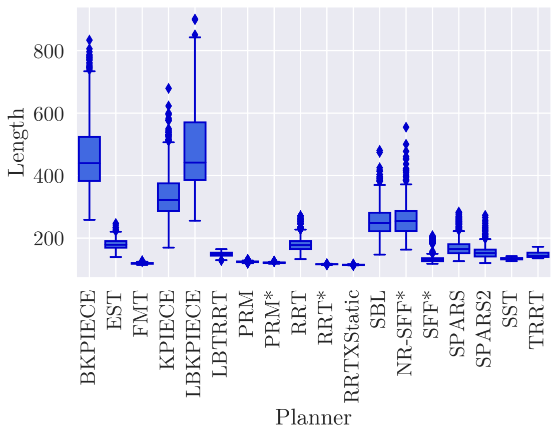
8.1 Planning in 2D Euclidean space

Planning in the 2D Euclidean space was tested on the OMPL implementation, as described in Chapter 7. The tested planners were left in their default configurations, only the “range” parameter was unified for all tree-expanding algorithms. Each test was repeated 1 000 times for each planner.

The results for Bugtrap configurations are very tight for most of the planners. The shortest solutions found asymptotically optimal RRT* and RRTXStatic planners, followed by PRM, PRM* and FMT algorithms. However, apart from the FMT planner, all of them ran in all cases for the maximum allowed time. The performance of the SFF* planner was mediocre, and poor concerning the NR-SFF* planner. Additionally, there is a significant difference between “in” and “out” configurations, as depicted in Figure 8.1 — the results for the “in” configuration are worse. This is probably caused by the priority queue utilized in NR-SFF* and SFF*. However, compared to classic EST, SFF* found shorter paths in shorter time in vast majority of cases. The performance of NR-SFF* is comparable to SBL. The worst results achieved the KPIECE-based algorithms, STRIDE and TRRT were not even able to solve the problem.



(a) : Lengths of the paths for Bugtrap "in"

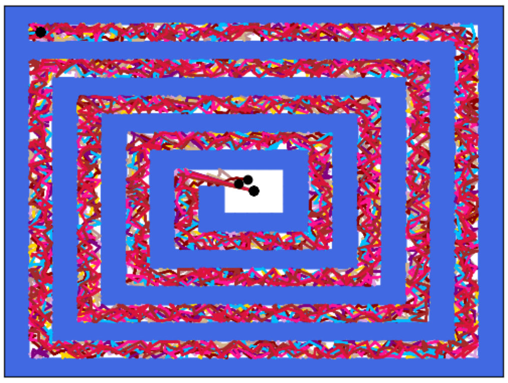


(b) : Lengths of the paths for Bugtrap "out"

Figure 8.1: Comparison of lengths of paths for Bugtrap configurations

The results for both Dense configurations are almost the same as in the Bugtrap problem. The main difference is only in the success rate and the computational times of the EST algorithm, which were the worst.

The outcome of the benchmark of Spiral configurations is different. Some planners, including EST and RRT*, have not found a solution in almost all runs, the success rates of other planners like SPARS or SST are low. The order of planners as per the lengths of their paths is then similar to the previous environments, except for RRT. The lengths of paths found using RRT are comparable to NR-SFF* and are significantly worse than the paths found using SFF*. The quality of paths found using SFF* compared to the paths found using KPIECE planner is depicted in Figure 8.2.


(a) : Paths found by KPIECE planner

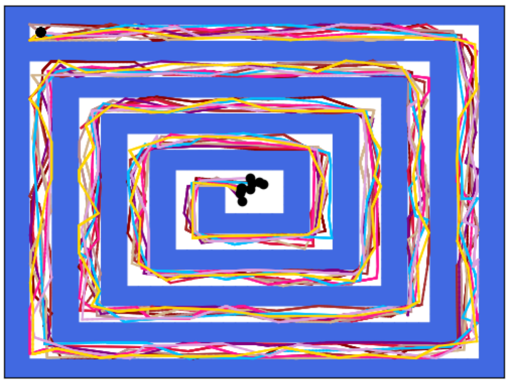
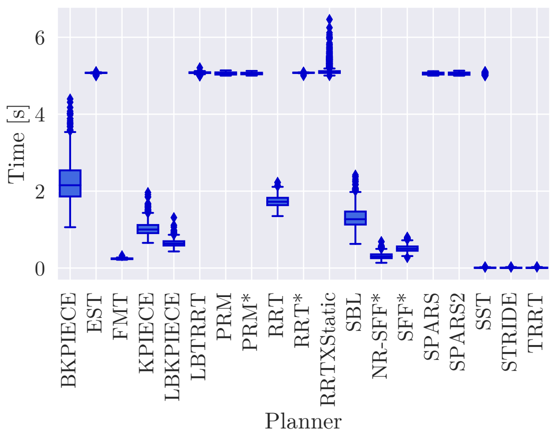
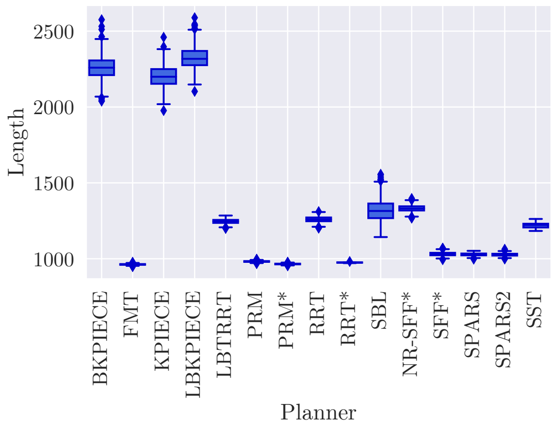

(b) : Paths found by SFF* planner

Figure 8.2: Comparison of quality of paths in Spiral map

However, considering also the times of computations, only the FMT planner achieved better results than SFF*. Such a good performance of SFF-based algorithms was reached thanks to independence from Voronoi bias, which steers other planners to wrong direction. In this way, they waste the time and consequently some of them could not even reach the given goal.

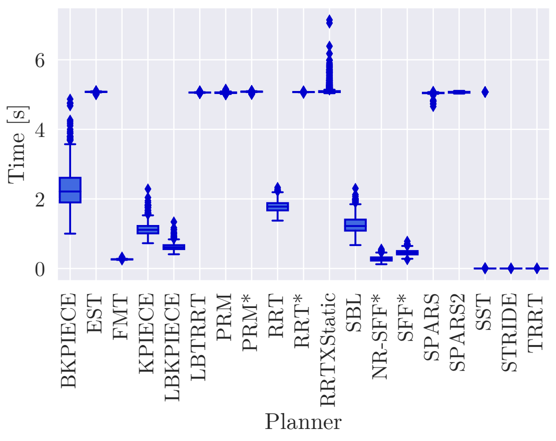


(a) : Computational times for Spiral "in"

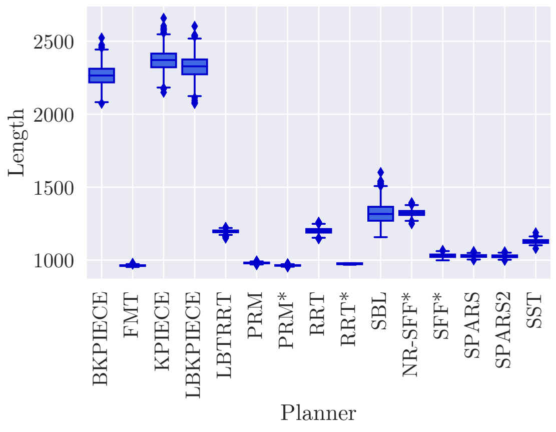


(b) : Lengths of the paths for Spiral "in"

Figure 8.3: Computational times and lengths of paths of Spiral "in"



(a) : Computational times for Spiral "out"



(b) : Lengths of the paths for Spiral "out"

Figure 8.4: Computational times and lengths of paths of Spiral "out"

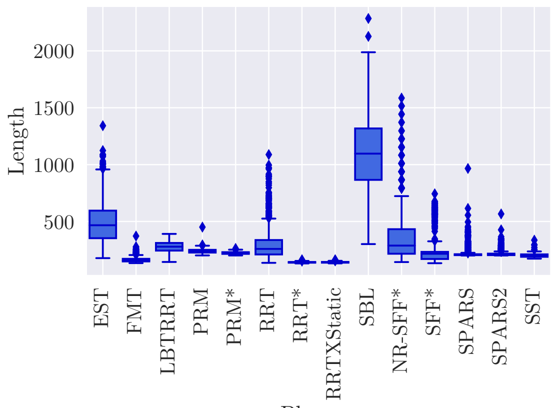
The benchmarks in the Maze map ended up the same way as the Spiral configurations, the only difference was a higher success rate for some planners like RRT* (RRTXStatic or EST did not even solve this problem). This is caused by the fact, that the Maze map does not have so narrow corridors, especially in the second section. In this way, the Voronoi bias does not have such a negative impact.

8.2 Planning in 3D Euclidean space

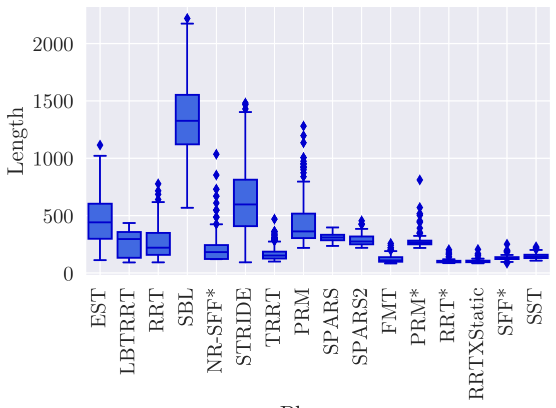
Unlike the benchmarks of the OMPL in 2D Euclidean space, which were done on custom maps and with custom settings, the tests in 3D Euclidean space were performed in 3D environments, which are part of OMPL.app, as described in Chapter 7. The parameters of benchmarks were also left in default, except from the number of runs, which was set to 500 for the Easy and Twistycool environments, and to 100 for the 3D Bugtrap environment. The longer timeouts are reason for this — 300 seconds for the Bugtrap, and 20 seconds for other two problems.

In the 3D Bugtrap problem all planners failed, i. e., no solution was found. In order to find at least some solution, the "range" settings of tree-expanding algorithms was adjusted. However, even after the adjustments, no solutions were found. This is clearly one of many problems discovered in the OMPL.

Solving the second problem, Easy, was more successful. Like the problems solved in 2D Euclidean space, the success rate of the SST planner was low and the STRIDE and TRRT algorithms did not find almost any solution. The worst results delivered BKPIECE and LBKPIECE planners, their paths were in average almost 2 000 times worse than the best average. The best average length of paths was achieved by RRT* and RRTXStatic planners, followed by FMT. The SFF* algorithm found paths comparable to paths found using PRM, but in much shorter average time. It also defeated the classic RRT planner and EST planner, which was even worse than NR-SFF*. The filtered results are depicted in Figure 8.5a.



(a) : Lengths of the paths found for the Easy problem



(b) : Lengths of the paths found for the Twistycool problem

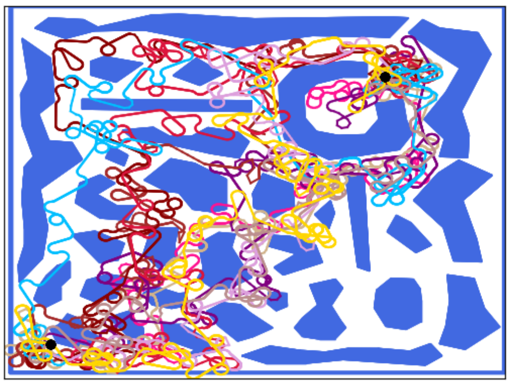
Figure 8.5: Lengths of the paths found in 3D Euclidean test problems

The results of the more difficult Twistycool problem are quite different. Most significant are the changes of success rates, where the majority of planners solved the problem in less than half of the runs. This concerns both SFF-based algorithms, together with EST, PRM, RRT* and others. The shortest paths were found using RRT* and RRTXStatic planners, followed by FMT and SFF*. Even results of the NR-SFF* algorithm are better than average, as it defeated EST, RRT or even PRM*. Such great results were achieved probably thanks to the independence from Voronoi bias. The filtered results are depicted in Figure 8.5b.

8.3 Planning for Dubins car

The last category of planners implemented in the OMPL are algorithms for planning for Dubins car. The tests were performed in the same maps as benchmarks for the 2D Euclidean space. The tested planners were left in their default configurations, only the "range" parameter was unified for all tree-expanding algorithms, and the "turningradius", representing the turning radius of the Dubins car, was adjusted.

However, issues with some planners were discovered during the tests. Some of the resulting paths, or Dubins maneuvers, collided with the obstacles, as shown in Figure 8.6. In cases of LBKPIECE, BKPIECE and SBL, only few curves go through the walls, for paths of SPARS, SPARS2 and RRTXStatic planners are the violations more evident, while the results of the PRM and PRM* planners should not even be taken in account.


(a) : Overlapping Dubins maneuvers — results of LBKPIECE planner

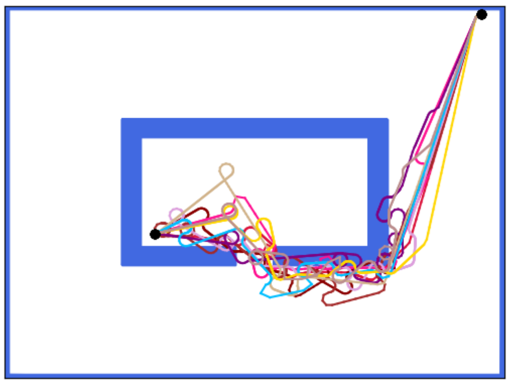
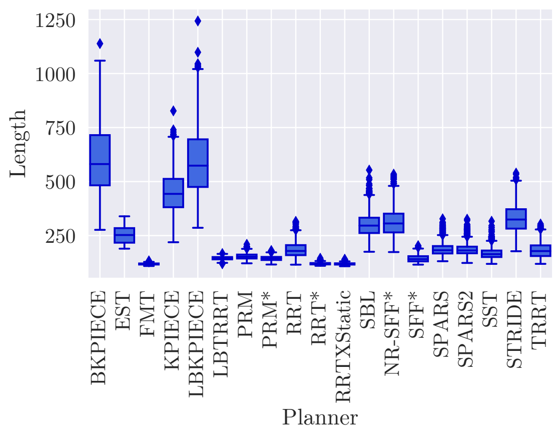

(b) : Final paths going through the walls of an obstacle — results of PRM planner

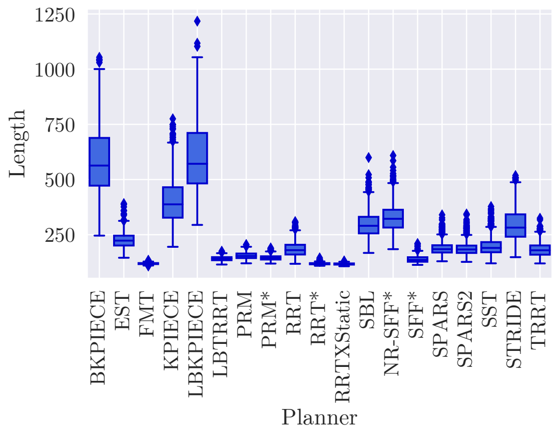
Figure 8.6: Issues with Dubins maneuvers in OMPL

The results for both Bugtrap configurations are different from the results for 2D Euclidean space in terms of the success rates and the computational times, but almost identical in terms of the path lengths. The lowest success rate when "getting into the bugtrap" has EST, followed by SFF* and NR-SFF*, which found a solution in around half of the runs. Other succeeded in almost all runs. For the problem of "getting out", only SFF-based algorithms did not succeeed in cca. 1/4 of runs for NR-SFF*, or 1/3 of runs for SFF*. The shortest solutions were in both cases found using RRT*, FMT and SFF* planners, the longest using LBKPIECE and BKPIECE — despite the fact that their paths collide with obstacles.

The Dense configurations brought similar results for all three watched parameters. The



(a) : Lengths of the paths for Bugtrap “in”



(b) : Lengths of the paths for Bugtrap “out”

Figure 8.7: Comparison of lengths of paths for Bugtrap configurations when planning for Dubins car

SFF* algorithm found one of the shortest solutions and defeated RRT, EST and PRM, but the success rate is low — cca. 1/2 of runs for one configuration, and even less than 1/10 of runs for the second configuration. Lower success rate has only EST with almost no successful runs.

The Spiral problem was solved only by the problematic planners colliding with obstacles, and in one configuration also by FMT.

The situation for the Maze problem is a bit better, unfortunately both problems were not solved by any of the SFF-based algorithms. On the other hand, the success rates of RRT-based algorithms are close to zero as well. Surprisingly, the KPIECE planner delivered quite good results.

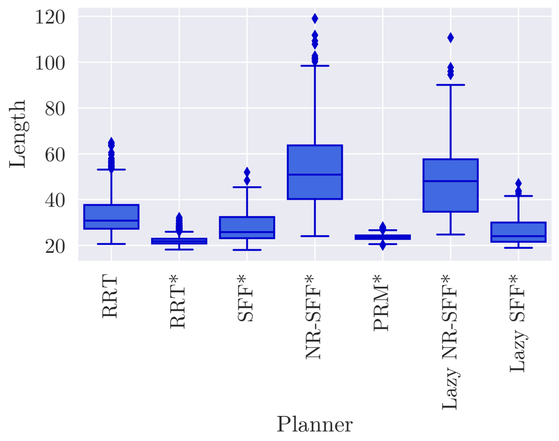
8.4 Planning for Dubins airplane

The planning for Dubins airplane is the first category of planning tested in the custom library of planning algorithms and also the only category of single-goal planning, where Lazy SFF planners were deployed.

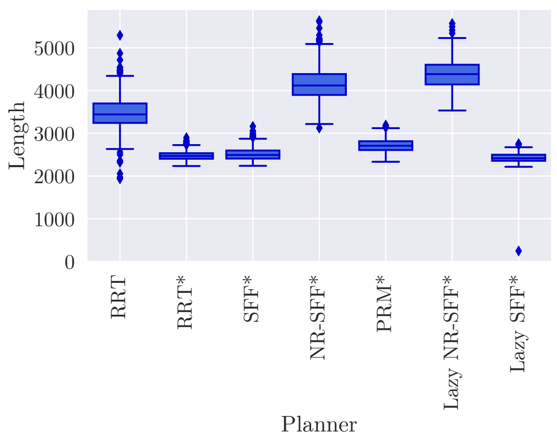
For the benchmarks, the 3D maps Building, Dense and Triangles introduced in Chapter 7 were used. Maximum number of iterations of SFF-based and RRT-based algorithms was 100 000, number of sampled nodes for PRM was 1 000. Apart from the radius of Dubins curves, also sampling ranges of pitch angles were adjusted for all planners.

With an exception of Lazy SFF algorithms, all planners solved every problem almost in all runs. The worst computational times were achieved by the Lazy SFF algorithms, due to the complicated and time demanding solving of Dubins Touring Problem. Slightly worse times has also PRM*, other times of runs are similar to each other.

The shortest paths for Dubins airplane found RRT* planner, closely followed by PRM. Lengths of the paths of SFF* and Lazy SFF* as well as NR-SFF* and Lazy NR-SFF* are almost identical and are definitely not worth the significantly longer computational time. They are also depicted in Figure 8.8a.



(a) : Lengths of the paths for Building problem



(b) : Lengths of the paths for Dense problem

Figure 8.8: Lengths of the paths of selected problems of planning for Dubins airplane

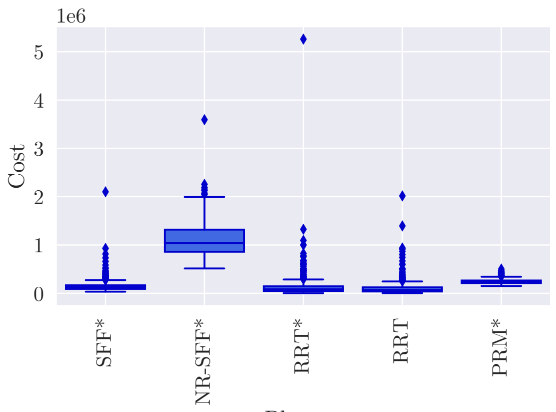
The lengths of paths in the Dense problem are better for SFF*, as their lengths are comparable to the lengths of paths found using RRT*. This is also depicted in Figure 8.8b. The results for the Triangle problem are almost the same — at least the order of planners.

8.5 Planning on polynomial trajectories in 2D

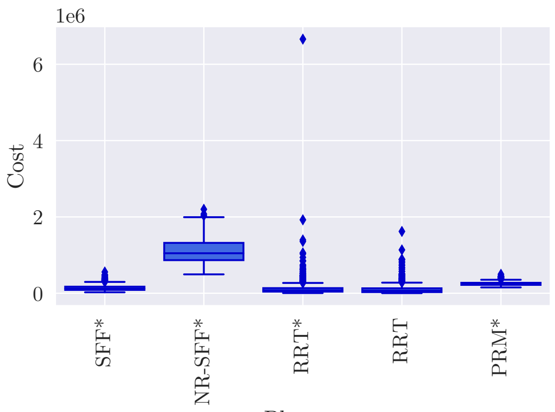
The benchmarking of the proposed trajectory planner with polynomial 2D trajectories was performed in the custom planning library, in the same environments as other 2D problems. Maximum number of iterations of SFF-based and RRT-based algorithms was 100 000, number of sampled nodes for PRM was 1 000. Unlike the previous dimensions, dynamic parameters were also set: the minimum and the maximum thrust equaled to one expansion step and the maximum rotation speed is 1 rad/s. The duration of a segment was 4 seconds. The gravity was not set for the 2D case. Beware, that the cost of solution was minimized rather than its length.

For both Bugtrap configurations, the results are practically the same. The problems were solved by all tested planners, the least successful was PRM* with cca. 2/3 of successful runs, the most successful were NR-SFF* and SFF*, which solved the problems in almost all runs. The lowest average cost of trajectories achieved RRT and RRT*, followed by SFF*. An example of such trajectory together with the expanded tree is depicted in Figure 8.10. The worst average cost delivered NR-SFF*. The times of computations are similar for the

RRT-based planners and SFF*, for the "in" problem the times for both SFF-based algorithms are worse again. Quite good cost of found trajectories has also PRM*, although its running times were the longest.



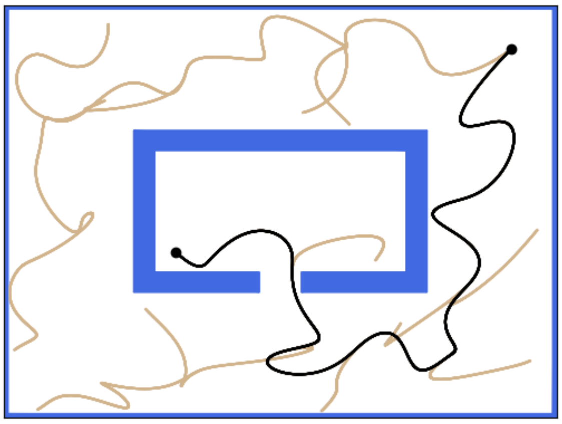
(a) : Costs of the trajectories for the Bugtrap "in" problem



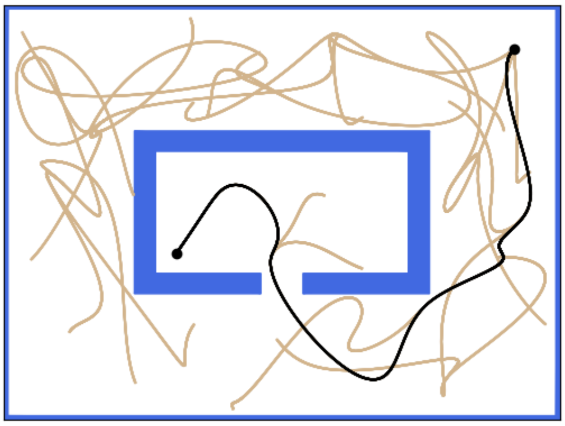
(b) : Costs of the trajectories for the Bugtrap "out" problem

Figure 8.9: Comparison of costs for both Bugtrap problems

The results for both Dense problems are the same as for the Bugtrap problem, apart from the PRM* planner, which did not solve the problem in any of the runs.



(a) : Expanded tree and final trajectory found by SFF



(b) : Expanded tree and final trajectory found by SFF*

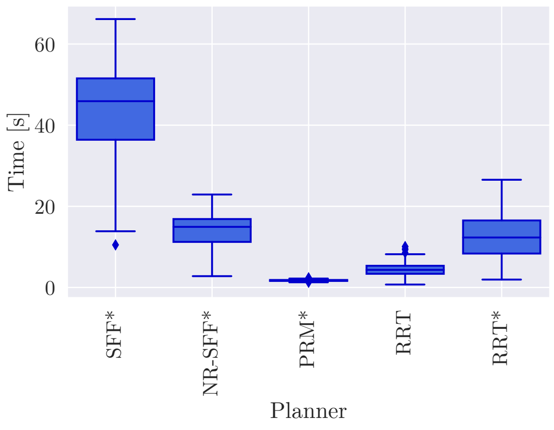
Figure 8.10: Expanded trees and trajectories in Bugtrap problem

For both Spiral and Maze problems was found no solution, as both are too difficult to be solved using polynomial trajectories. The only exception is the Maze "corridor" problem, which was in a few cases solved by the RRT planner. According to smaller tests, increase of number of iterations did not lead to any solution as well, at least for the SFF-based algorithms. This might be caused by the priority bias, or by the size and the dimensionality of the sampling space.

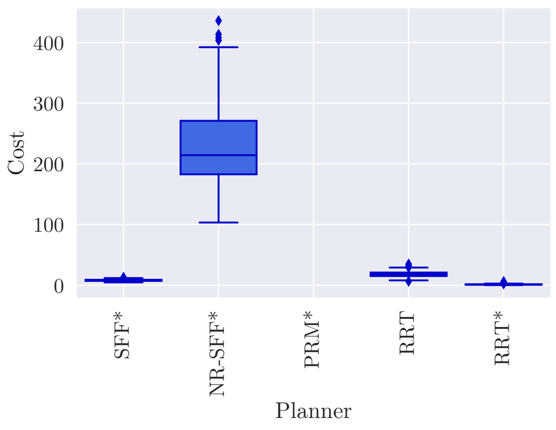
8.6 Planning on polynomial trajectories in 3D

The 3D trajectories were tested in almost the same way as the 2D trajectories, of course with an exception of used environments. The maximum and minimum thrust equaled to one expansion step, the maximum rotation speed was 1 rad/s. The duration of a segment was 4 seconds. The size of gravity was set to triple of the expansion step, it acted in the negative direction of z-axis. The cost of the solution was minimized rather than its length.

The 3D trajectory problems are quite problematic for the PRM* algorithm. For the Building problem, it did not find any solution. Oppositely, the RRT-based algorithms found a valid path in every run. The SFF-based algorithms were successful only in half of the cases. Regarding the cost of solutions, the best results achieved RRT* followed by SFF*. However, the computational times of SFF* are the worst, they are three times longer than the times of RRT*. The results are depicted in Figure 8.11. An example of an expanding tree with the final polynomial trajectory is shown in Figure 8.12.



(a) : Times of computations



(b) : Costs of trajectories

Figure 8.11: Results for the Building problem

The Dense problems were solved only by the RRT-based algorithms, and in the case of RRT* only in few percent of cases — usually the computation timed out. The environment is probably too spacious, so the planners with Voronoi bias are in advantage, as they are able to expand faster.

The results for the Triangles problem are better for all planners. All runs were successful for both RRT-based planners, which solved all runs, the SFF-based algorithms were successful in half of the cases, and even PRM* found a path in cca. 1/5 of all runs. RRT* won in the cost of solutions followed by SFF* and PRM*. However, it should be noted that the computational times of SFF* are the worst and are in order of minutes.

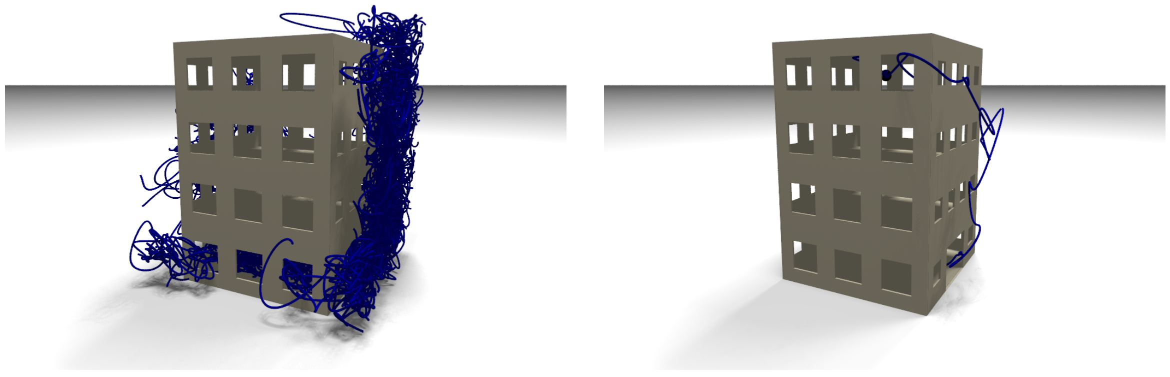

(a) : Expanding SFF* tree (b) : Final trajectory

Figure 8.12: Expanding tree and final polynomial trajectory by SFF* in the Building environment

8.7 Discussion

It is apparent from the benchmarks, that SFF-based algorithms are for single-goal planning definitely not the best, but they are an interesting alternative to classic planners for some environments.

Such environments were Spiral and Maze in 2D Euclidean spaces, where other planners failed (computed for longer time or found longer paths) due to the effect of Voronoi bias, which steered their expansion to a wrong direction. For other environments, the proposed algorithms found mediocre paths in mediocre times, but they still defeated EST and KPIECE-based planners. Unfortunately, the priority queue had in some cases negative impact on the results of SFF*.

Similar results were achieved in 3D Euclidean space, where SFF* delivered better results in more difficult Twistycool environment and outperformed standard planners like RRT or EST. This was possible thanks to the independence from Voronoi bias.

When planning for Dubins car, NR-SFF* and SFF* found in some environments very good paths, but achieved very low success rates in comparison to other planners. Here, Voronoi bias probably helped other planners as they were able to expand faster.

In the case of planning for Dubins airplane, the best average of path lengths achieved RRT*, but it is closely followed by SFF*. Unfortunately the computational times were the worst for the SFF-based planners, probably due to a higher number of computations of path segments, i. e., 3D Dubins maneuvers.

For 2D polynomial trajectories, RRT* excelled again, but the SFF* planner was able to achieve the highest success rate of all planners. For the 3D case of polynomial trajectories, identical results as for the Dubins airplane were achieved, probably due to the higher number of generations of trajectory segments as well.

Chapter 9

Space Filling Forest for multi-goal motion planning

As already mentioned in Chapter 7, the proposed algorithms for multi-goal planning were benchmarked with their main competitors in the custom library of planning algorithms. Only the planning with Dubins vehicles, i. e., the Dubins car and Dubins airplane, was tested because the multi-goal planning in Euclidean spaces was compared with other planners in [5].

9.1 Planning for Dubins car

The algorithms for planning with the Dubins car were tested in the environments "Dense", "Triangles" and "Varying density", each in configurations with 5, 10 and 20 cities. The novel SFF* and NR-SFF* algorithms were compared with the tree-growing Lazy TSP and the roadmap-constructing PRM* planner.

All planners used the same radius of the Dubins maneuvers, both SFF-based planners were limited by 100 000 iterations, each RRT expansion of the Lazy TSP algorithm was limited by 100 000 iterations and used the same sampling distance as SFF-based algorithms. The number of nodes sampled using the PRM* planner was 1 000. Three main parameters were watched, likewise to single-goal planning: Time, the length of the TSP tour and the TSP succcess rate, i. e., a percentage of runs in which one graph was formed by all paths connecting the cities and the LKH TSP solver found a valid tour.

An example of a final TSP tour for the Dubins car found using the SFF* planner in "Dense" map with 5 cities is depicted in Figure 9.1.

The complete results of the benchmarks are recorded in Table 9.1. Please note that the values for the TSP length and the computational times are in format "mean | std", where *mean* is the mean of valid values, and *std* is the standard deviation from the mean value. The values for the computational time include both solved and unsolved cases.

According to the measured values, it might appear that the best planner is PRM* in all tested environments. It found the shortest paths with lowest dispersion among runs, additionally in the shortest time. However, its success rates in "Dense" and "Varying density" are the lowest. This is probably caused by the choice of the number of sampled nodes —

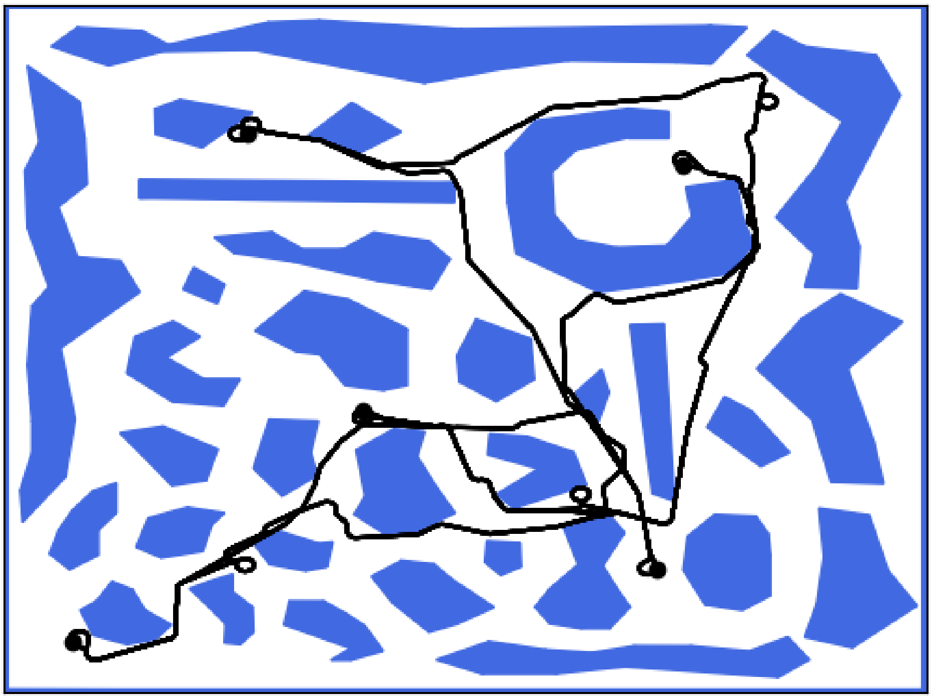


Figure 9.1: Final TSP tour after planning with Dubins car in "Dense" map using SFF*; visible points of inflection are caused by selected small turning radius

higher number of sampled nodes would increase the success rate, but also extend the path length due to the higher number of Dubins maneuvers included. The highest success rate and comparable length of the tour were achieved by the Lazy TSP planner, but its time of run increases exponentially with the number of cities, which is apparent from the results for 20 cities.

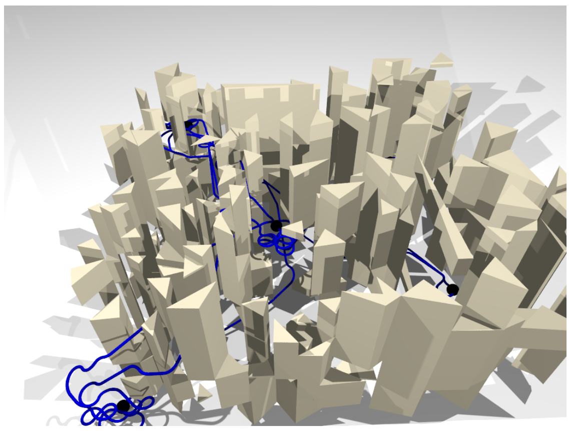
Both SFF-based planners provided longer paths in mediocre time and their success rates in "Triangles" and "Varying density" environments were poor. The results for the "Dense" map with 10 and 20 cities are surprising, as other planners failed in almost all runs. This is probably caused by some hard-to-connect city in a narrow passage. This result confirms the assumption that SFF-based planners are suitable for problems with narrow passages, even for the planning with the Dubins car, thanks to independence from Voronoi bias.

9.2 Planning for Dubins airplane

The planners for the Dubins airplane model were tested in 3D environments "Dense", "Building" and "Triangles", each in configurations with 5, 10 and 20 cities. The same planners as for the Dubins car problem were benchmarked, also with Lazy NR-SFF* a Lazy SFF* planners using the decoupled solution of the DTSP problem, i. e., finding path in 3D Euclidean space first and then solving DTP with resulting path nodes.

All planners used the same initial radius of Dubins maneuver in the vertical level, as well as the same ranges of pitch angles. All SFF-based algorithms were limited by 100 000 iterations, each RRT expansion of the Lazy TSP algorithm was limited by 100 000 iterations and used the same sampling distance as SFF-based algorithms. The number of nodes sampled using the PRM* planner was 1 000. Number of sampled angles for the DTP problem solved by Lazy SFF planners was 2, i. e., only the opposite angles of each node were considered. The same parameters as for the Dubins car problem (Time, the length of the TSP tour and the TSP success rate) were watched.

Two examples of a final TSP tour for Dubins airplane found using the SFF* planner in "Building" and "Dense" maps are illustrated in Figure 9.2.


(a) : Final TSP tour in "Dense" map with 5 cities

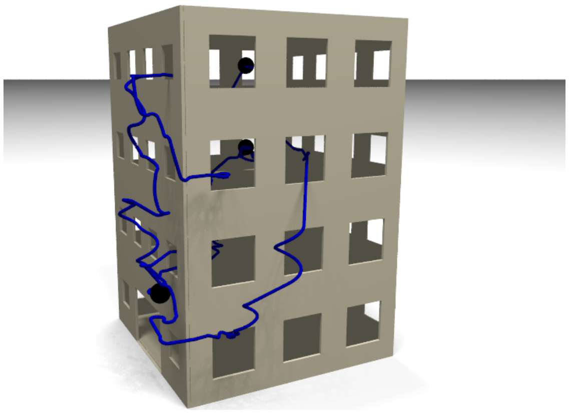

(b) : Final TSP tour in "Building" map with 5 cities

Figure 9.2: Final TSP tours after planning with Dubins airplane using SFF*

The complete results of the benchmark are recorded in Table 9.2. They are almost the same as for the 2D case, as the PRM* planner was able to find the shortest paths in the shortest time, but with the lower success rate. This is valid especially in the "Triangles" map with 10 and 20 cities. The Lazy TSP planner found paths comparable in length, but its demands on the computational time are relatively higher for configurations with higher number of cities than for the 2D case, and increases exponentially as well.

The Lazy SFF algorithms failed for every configuration. This was caused by the fact that the difference between paths in Euclidean space and trajectories for the Dubins airplane is so significant that such a transformation is impossible in environments with obstacles. In addition, the solution of the DTP with the resulting path nodes apparently increased the times of computations, so the duration of computation for the Lazy SFF planners is higher than for the standard SFF algorithms. Therefore, there is no use in using the lazy approach. This fact highlights the need of planning directly with the Dubins maneuvers, as both NR-SFF* and SFF* do.

The proposed SFF* algorithm produced paths comparable in length with the Lazy TSP approach in mediocre times, with an exception of the “Building” map where the computations were even faster than with the PRM* planner. The computational time was even smaller for the NR-SFF* planner with no rewiring, but at the cost of longer tours.

9.3 Discussion

For both dimensionalities of problems, the PRM* planner found the shortest paths, but with one of the lowest success rates. This is due to the relatively low number of sampled points. If this number were higher, the results would be probably different — the success rates would be higher at the cost of longer paths with more Dubins maneuvers.

The second best planner, Lazy TSP, found on the one hand tours of comparable in length to PRM*, but on the other hand its run time increased exponentially, which is especially evident in 3D. However, it should be noted, that Lazy TSP is a “lazy” variant of classic approach to multi-goal motion planning. In the case of classic approach, where paths would be searched for all city–city pairs, the time would be much longer.

The Lazy SFF approaches totally failed, because they failed to convert the classic 3D path into 3D Dubins maneuver in environments with obstacles. From these failures, it is apparent that the only valid option is to expand directly using Dubins maneuvers.

On the one hand, SFF-based methods for multi-goal path planning found only average paths in mediocre times, on the other hand, linked some hard-to-connect cities, where other planners failed. Perhaps with different settings of Dubins vehicles and in environments with more narrow passages, the results would be better.

Table 9.1: Results of multi-goal path planning for Dubins car model

Num. cities:		**5**			**10**			**20**	
	Succ.	**TSP length**	**Time**	**Succ.**	**TSP length**	**Time**	**Succ.**	**TSP length**	**Time**
	-	$\times 10^3$	seconds	-	$\times 10^3$	seconds	-	$\times 10^3$	seconds
Dense									
SFF*	*99 %*	13.32 | 1.70	27.48 | 0.90	*90 %*	22.07 | 1.82	30.90 | 0.71	*88 %*	30.85 | 2.12	33.59 | 0.70
NR-SFF*	*99 %*	21.10 | 2.96	16.88 | 0.29	*89 %*	34.16 | 2.97	19.19 | 0.53	*87 %*	46.02 | 3.71	21.52 | 0.56
PRM*	*93 %*	**9.13** | **0.65**	1.93 | 0.19	*1 %*	**14.77** | **1.35**	1.96 | 0.15	*2 %*	**20.37** | **2.28**	2.12 | 0.12
Lazy TSP	*98 %*	10.41 | 0.73	32.86 | 8.84	*1 %*	16.37 | 1.06	62.01 | 33.77	*0 %*	nan | nan	96.75 | 57.87
Triangles									
SFF*	*78 %*	2.90 | 0.56	1.79 | 2.21	*50 %*	3.96 | 0.55	1.53 | 1.91	*59 %*	5.94 | 0.71	1.64 | 2.10
NR-SFF*	*79 %*	3.99 | 0.90	1.60 | 2.07	*54 %*	5.07 | 0.86	1.43 | 1.89	*60 %*	6.96 | 0.89	1.49 | 1.99
PRM*	*91 %*	**2.17** | **0.08**	0.81 | 0.07	*91 %*	**2.86** | **0.10**	0.82 | 0.07	*91 %*	**4.13** | **0.25**	0.88 | 0.09
Lazy TSP	*100 %*	2.58 | 0.20	14.71 | 2.69	*100 %*	3.41 | 0.21	26.95 | 6.65	*100 %*	4.86 | 0.39	31.09 | 10.68
Varying density									
SFF*	*46 %*	13.21 | 2.19	8.61 | 4.96	*46 %*	15.84 | 1.89	11.57 | 5.72	*39 %*	24.60 | 2.41	13.84 | 5.12
NR-SFF*	*46 %*	19.93 | 3.69	6.27 | 4.71	*45 %*	23.76 | 3.53	9.19 | 5.39	*39 %*	35.43 | 4.11	11.58 | 4.80
PRM*	*35 %*	**10.86** | **0.65**	2.32 | 0.11	*27 %*	**12.36** | **0.80**	2.38 | 0.14	*18 %*	**18.56** | **1.41**	2.51 | 0.15
Lazy TSP	*100 %*	12.41 | 0.85	7.87 | 2.02	*100 %*	14.39 | 0.89	24.96 | 5.14	*100 %*	22.68 | 1.76	88.69 | 14.59

Table 9.2: Results of multi-goal path planning for Dubins airplane model

Num. cities:		5			10			20	
	Succ.	**TSP length**	**Time**	**Succ.**	**TSP length**	**Time**	**Succ.**	**TSP length**	**Time**
	-	$\times 10^3$	seconds	-	$\times 10^3$	seconds	-	$\times 10^3$	seconds
Dense									
SFF*	*100 %*	17.52 | 1.84	37.75 | 0.85	*98 %*	19.67 | 1.73	37.39 | 3.27	*95 %*	26.34 | 2.00	39.05 | 4.74
NR-SFF*	*99 %*	36.22 | 4.53	20.02 | 0.99	*97 %*	36.43 | 3.61	20.06 | 3.02	*96 %*	44.73 | 3.94	21.61 | 3.96
PRM*	*99 %*	**7.99 | 0.25**	6.25 | 0.26	*98 %*	**12.35 | 0.43**	6.34 | 0.26	*98 %*	**18.93 | 1.13**	6.71 | 0.38
Lazy TSP	*100 %*	10.14 | 0.64	19.43 | 4.56	*100 %*	16.43 | 0.81	87.38 | 17.04	*100 %*	25.62 | 1.74	171.56 | 23.50
L. NR-SFF*	*0 %*	nan | nan	0.45 | 0.24	*0 %*	nan | nan	2.69 | 1.62	*0 %*	nan | nan	12.94 | 15.93
Lazy SFF*	*0 %*	9.21 | 0.00	0.85 | 0.30	*0 %*	nan | nan	3.52 | 2.06	*0 %*	nan | nan	13.64 | 8.01
Building									
SFF*	*87 %*	0.13 | 0.03	2.04 | 0.60	*84 %*	0.19 | 0.03	2.47 | 1.00	*71 %*	0.35 | 0.08	2.68 | 1.12
NR-SFF*	*87 %*	0.20 | 0.05	1.34 | 0.65	*83 %*	0.26 | 0.05	1.74 | 0.90	*72 %*	0.45 | 0.10	2.07 | 0.96
PRM*	*100 %*	**0.08 | 0.00**	3.53 | 0.42	*100 %*	**0.12 | 0.00**	3.55 | 0.36	*99 %*	**0.18 | 0.01**	3.72 | 0.26
Lazy TSP	*100 %*	0.13 | 0.02	1.51 | 0.42	*100 %*	0.15 | 0.01	13.61 | 1.69	*100 %*	0.43 | 0.03	4.40 | 0.64
L. NR-SFF*	*0 %*	nan | nan	4.90 | 5.37	*0 %*	nan | nan	8.80 | 15.65	*0 %*	nan | nan	11.74 | 16.55
Lazy SFF*	*0 %*	nan | nan	5.36 | 6.42	*0 %*	nan | nan	8.73 | 9.59	*0 %*	nan | nan	11.25 | 14.31
Triangles									
SFF*	*59 %*	0.11 | 0.02	10.71 | 7.10	*57 %*	0.16 | 0.02	12.72 | 6.23	*55 %*	0.25 | 0.03	13.37 | 5.60
NR-SFF*	*62 %*	0.19 | 0.05	9.17 | 6.68	*58 %*	0.24 | 0.05	11.49 | 6.16	*55 %*	0.35 | 0.05	12.20 | 5.53
PRM*	*56 %*	**0.07 | 0.01**	2.58 | 0.12	*39 %*	**0.10 | 0.01**	2.69 | 0.15	*35 %*	**0.15 | 0.01**	2.90 | 0.19
Lazy TSP	*100 %*	0.08 | 0.01	7.10 | 2.98	*99 %*	0.11 | 0.01	27.65 | 5.64	*99 %*	0.20 | 0.02	116.55 | 12.95
L. NR-SFF*	*0 %*	0.12 | 0.00	7.25 | 18.66	*0 %*	nan | nan	30.02 | 43.02	*0 %*	nan | nan	110.37 | 95.99
Lazy SFF*	*0 %*	0.08 | 0.01	7.56 | 16.55	*0 %*	nan | nan	36.70 | 59.38	*0 %*	nan | nan	104.24 | 89.23

Chapter 10
Conclusion

The main objective of this work was to study and extend the existing sampling-based motion planning methods from the family of "Space Filling Forest", NR-SFF* and SFF*. These have been defined in [5] — one of the co-authors of the paper is the author of this thesis. Two challenges were defined for this work. First, derive methods for single-goal motion planning in 2D and 3D, for Dubins car and for Dubins Airplane, as well as for planning with state-of-the-art polynomial trajectories in 2D and 3D. Second, propose a novel multi-goal motion planning method for Dubins car and Dubins airplane, where instead of planning a path for each pair of cities, multiple trees are grown in SFF manner and the paths are extracted from them at once.

In Chapter 2, the problem of motion planning was defined as well as the necessary terminology used throughout the whole work was described. The next Chapter 3 gave reader an overview of single-goal planners. Basic algorithms as well as newer and more complex sampling-based methods were presented here. Specifically, Probabilistic Roadmaps, Rapid Exploring Random Trees and Expansive Space Trees with their asymptotically optimal variants were studied, and their behavior with respect to various parameters was analyzed. Subsequently, Dubins car and Dubins airplane models were described together with approaches to planning with them. Minimum snap and minimum jerk polynomial trajectories were also studied together with related works.

The multi-goal path planning was discussed in Chapter 4. Methods for multi-goal planning derived from single-goal sampling-based planners were presented, and two combinatorial problems were introduced, the Traveling Salesman Problem and the Orienteering Problem along with variants for Dubins vehicles. An overview of the principles and methods used for their solution was given and selected solvers were referenced.

In Chapter 5 a family of motion planning algorithms called Space Filling Forest was introduced and their behavior with various settings was analyzed.

The next Chapter 6 briefly introduced the implementation of the proposed SFF-based algorithms in OMPL, followed by a detailed description of the created custom library of planning algorithms. All SFF-based algorithms are implemented in this library, along with a selection of RRT- and PRM-based algorithms. All their implementation details and supporting algorithms used were discussed. An overview of implemented sampling methods, collision checking, nearest neighbors search and generation of paths for Dubins vehicles was provided, as well as polynomial trajectories.

The resources, maps and models used for bechmarking of the proposed algorithms were described in Chapter 7 and finally, in Chapters 8 and 9 the results of the benchmarks were studied. First, single-goal planning for Euclidean space, for Dubins vehicles and on polynomial trajectories was tested, always in 2D and 3D.

In Euclidean spaces, the influence of the Voronoi bias was evident. Although SFF* was not the best planner, it excelled in environments like Spiral or Maze with the best length-to-time ratio. It outperformed classic planners like RRT and EST, in some cases also sPRM.

For problems with Dubins car, the success rate of SFF-based planners was lower than of the others. Also, for problems with Dubins airplane, SFF* may not be the best option, but still worked better than standard PRM* or RRT. The most suitable planner for planning for Dubins vehicles was RRT*, with Voronoi bias allowing faster tree expansion. RRT* also had the best results for problems with polynomial trajectories where SFF-based algorithms failed in most runs and their computational times were bad, especially in the case of 3D. It still outperformed the PRM* planner.

Second, novel multi-goal planning for Dubins vehicles was benchmarked. In almost all environments, the shortest tours were found with paths from the PRM* planner, but this approach also had the lowest success rate. For the Lazy TSP planner, computational time grew exponentially with the number of cities. The SFF-based planner found mediocre tours in mediocre times, but managed to link hard-to-connect cities. Decoupled Lazy SFF planners failed.

Any consequent research of SFF might focus on adjustments of planners parameters, especially when planning with polynomial trajectories, or on searching for other applications where SFF-based algorithms might be deployed. Also the multi-goal planning might get further extended by planning on polynomial trajectories.

In this work, new SFF-based methods for single-goal path and trajectory planning were presented along with a novel approach to multi-goal path planning for Dubins vehicles. Although the proposed algorithms have not achieved excellent results, they are an interesting alternative to classic sampling-based planners, especially in environments where the Voronoi bias of classic planners might lead to worse results.

Appendix A

Bibliography

[1] J. H. Reif and J. A. Storer, “Shortest paths in euclidean space with polyhedral obstacles.,” 1985.

[2] J. Canny and J. Reif, “New lower bound techniques for robot motion planning problems,” in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pp. 49–60, 1987.

[3] J. H. Reif, “Complexity of the mover’s problem and generalizations,” in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pp. 421–427, 1979.

[4] V. Vonásek and R. Pěnička, “Space-filling forest for multi-goal path planning,” in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1587–1590, 2019.

[5] J. Janoš, V. Vonásek, and R. Pěnička, “Multi-goal path planning using multiple random trees,” *IEEE Robotics and Automation Letters*, vol. 6, p. 4201–4208, Apr 2021.

[6] S. M. LaValle, *Planning Algorithms.* USA: Cambridge University Press, 2006.

[7] M. Ignat’ev, F. Kulakov, and A. Pokrovskij, *Robot-manipulator Control Algorithms.* Joint Publications Research Service, 1984.

[8] N. J. Nilsson, “A mobile automaton: An application of artificial intelligence techniques,” in *IJCAI*, 1969.

[9] H. Chang and T.-Y. Li, “Assembly maintainability study with motion planning,” in *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 1012–1019 vol.1, 1995.

[10] V. Vonásek and B. Kozlíková, “Tunnel detection in protein structures using sampling-based motion planning,” in *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, pp. 185–192, July 2017.

[11] A. Gasparetto, P. Boscariol, A. Lanzutti, and R. Vidoni, *Path Planning and Trajectory Planning Algorithms: A General Overview*, pp. 3–27. Cham: Springer International Publishing, 2015.

[12] J. Latombe, *Robot Motion Planning.* The Springer International Series in Engineering and Computer Science, Springer US, 2012.

[13] S. Spitz and A. Requicha, "Multiple-goals path planning for coordinate measuring machines," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 3, pp. 2322–2327 vol.3, 2000.

[14] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[15] J. Kuffner, "Effective sampling and distance metrics for 3D rigid body path planning," in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2004, pp. 3993 – 3998 Vol.4, 04 2004.

[16] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques.* New York, NY, USA: Association for Computing Machinery, 1991.

[17] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Commun. ACM*, vol. 22, p. 560–570, oct 1979.

[18] C. O'Dúnlaing, M. Sharir, and C. Yap, "Retraction: A new approach to motion-planning," *Journal of Algorithms*, vol. 6, pp. 207–220, 01 1983.

[19] O. Takahashi and R. Schilling, "Motion planning in a plane using generalized Voronoi diagrams," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 2, pp. 143–150, 1989.

[20] P. F. Rowat, *Representing spatial experience and solving spatial problems in a simulated robot environment.* PhD thesis, University of British Columbia, 1979.

[21] D. G. Kirkpatrick, "Efficient computation of continuous skeletons," in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pp. 18–27, 1979.

[22] M. Kallmann, "Path planning in triangulations," in *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games*, 01 2005.

[23] J. Kümpel, J. Sparbert, B. Tibken, and E. P. Hofer, "Quadtree decomposition of configuration space for robot motion planning," in *1999 European Control Conference (ECC)*, pp. 277–282, 1999.

[24] O. Khatib, “Real-time obstacle avoidance for manipulators and mobile robots,” in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 500–505, 1985.

[25] J. Barraquand and J.-C. Latombe, “Robot motion planning: A distributed representation approach,” *International Journal of Robotic Research - IJRR*, vol. 10, pp. 628–649, 12 1991.

[26] X. Yun and K.-C. Tan, “A wall-following method for escaping local minima in potential field based motion planning,” in *1997 8th International Conference on Advanced Robotics. Proceedings. ICAR’97*, pp. 421–426, 1997.

[27] S. Masoud and A. Masoud, “Motion planning in the presence of directional and regional avoidance constraints using nonlinear, anisotropic, harmonic potential fields: a physical metaphor,” *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 32, no. 6, pp. 705–723, 2002.

[28] J. Barraquand, L. Kavraki, J.-C. Latombe, R. Motwani, T.-Y. Li, and P. Raghavan, “A random sampling scheme for path planning,” *The International Journal of Robotics Research*, vol. 16, no. 6, pp. 759–774, 1997.

[29] S. Karaman and E. Frazzoli, “Sampling-based algorithms for optimal motion planning,” *CoRR*, vol. abs/1105.1186, 2011.

[30] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, “Probabilistic roadmaps for path planning in high-dimensional configuration spaces,” *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[31] L. Kavraki, M. Kolountzakis, and J.-C. Latombe, “Analysis of probabilistic roadmaps for path planning,” *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 166–171, 1998.

[32] R. Geraerts and M. Overmars, “Sampling techniques for probabilistic roadmap planners,” tech. rep., Institute of information and computing sciences, Utrecht university, 01 2004.

[33] M. Branicky, S. LaValle, K. Olson, and L. Yang, “Quasi-randomized path planning,” in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 2, pp. 1481–1487 vol.2, 2001.

[34] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” tech. rep., Iowa State University, 1998.

[35] S. M. LaValle and J. James J. Kuffner, “Randomized kinodynamic planning,” *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[36] S. LaValle and J. Kuffner, "Randomized kinodynamic planning," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 1, pp. 473–479 vol.1, 1999.

[37] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Proceedings of International Conference on Robotics and Automation*, vol. 3, pp. 2719–2726 vol.3, 1997.

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

[39] P. Souères and J.-D. Boissonnat, *Optimal trajectories for nonholonomic mobile robots*, pp. 93–170. Laboratoire d'Analye et d'Architecture des Systèmes, 04 2006.

[40] P. Váňa and J. Faigl, "Optimal solution of the generalized Dubins interval problem finding the shortest curvature-constrained path through a set of regions," *Autonomous Robots*, vol. 44, no. 7, pp. 1359–1376, 2020.

[41] H. Chitsaz and S. M. LaValle, "Time-optimal paths for a Dubins airplane," *2007 46th IEEE Conference on Decision and Control*, pp. 2379–2384, 2007.

[42] M. Owen, R. Beard, and T. McLain, *Implementing Dubins Airplane Paths on Fixed-Wing UAVs**, pp. 1677–1701. Springer, Dordrecht, 08 2015.

[43] Y. Lin and S. Saripalli, "Path planning using 3D Dubins curve for unmanned aerial vehicles," in *2014 International Conference on Unmanned Aircraft Systems, ICUAS 2014 - Conference Proceedings*, pp. 296–304, 05 2014.

[44] P. Váňa, A. Alves Neto, J. Faigl, and D. G. Macharet, "Minimal 3D Dubins path with bounded curvature and pitch angle," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8497–8503, 2020.

[45] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, pp. 2520–2525, 2011.

[46] C. Richter, A. Bry, and N. Roy, *Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments*, pp. 649–666. Cham: Springer International Publishing, 2016.

[47] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran, "Continuous-time trajectory optimization for online UAV replanning," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5332–5339, 2016.

[48] M. Burri, H. Oleynikova, M. W. Achtelik, and R. Siegwart, “Real-time visual-inertial mapping, re-localization and planning onboard MAVs in unknown environments,” in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1872–1878, 2015.

[49] M. W. Mueller, M. Hehn, and R. D’Andrea, “A computationally efficient motion primitive for quadrocopter trajectory generation,” *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[50] M. Ryll, J. Ware, J. Carter, and N. Roy, “Efficient trajectory planning for high speed flight in unknown environments,” in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 732–738, 2019.

[51] Z. Tang, B. Chen, R. Lan, and S. Li, “Vector field guided RRT* based on motion primitives for quadrotor kinodynamic planning,” *J. Intell. Robotic Syst.*, vol. 100, pp. 1325–1339, 2020.

[52] D. Devaurs, T. Siméon, and J. Cortés, “A multi-tree extension of the transition-based RRT: Application to ordering-and-pathfinding problems in continuous cost spaces,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2991–2996, 2014.

[53] H. H. Hoos and T. Stützle, “8 - travelling salesman problems,” in *Stochastic Local Search* (H. H. Hoos and T. Stützle, eds.), The Morgan Kaufmann Series in Artificial Intelligence, pp. 357–416, San Francisco: Morgan Kaufmann, 2005.

[54] C. E. Miller, A. W. Tucker, and R. A. Zemlin, “Integer programming formulation of traveling salesman problems,” *J. ACM*, vol. 7, p. 326–329, oct 1960.

[55] G. Dantzig, R. Fulkerson, and S. Johnson, “Solution of a large-scale traveling-salesman problem,” *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[56] M. Grötschel and O. Holland, “Solution of large-scale traveling salesman problems,” *Math. Program.*, vol. 51, pp. 141–202, 07 1991.

[57] M. Held and R. M. Karp, “The traveling-salesman problem and minimum spanning trees: Part ii,” *Mathematical Programming*, vol. 1, pp. 6–25, 1971.

[58] D. Applegate, W. J. Cook, V. Chvátal, and R. Bixby, “Concorde TSP solver.”

[59] B. Alsalibi, M. Babaeianjelodar, and I. Venkat, “A comparative study between the nearest neighbor and genetic algorithms: A revisit to the traveling salesman problem,” *International Journal of Computer Science and Electronics Engineering*, vol. 1, pp. 34–38, 12 2012.

[60] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, *An analysis of several heuristics for the traveling salesman problem*, pp. 45–69. Dordrecht: Springer Netherlands, 2009.

[61] N. Christofides, “Worst-case analysis of a new heuristic for the travelling salesman problem,” tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[62] N. Mladenović and P. Hansen, “Variable neighborhood search,” *Computers Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[63] S. Lin and B. W. Kernighan, “An effective heuristic algorithm for the traveling-salesman problem,” *Oper. Res.*, vol. 21, pp. 498–516, 1973.

[64] K. Helsgaun, “An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems,” tech. rep., Roskilde University, Denmark, 12 2017.

[65] B. Englot and F. S. Hover, “Three-dimensional coverage planning for an underwater inspection robot,” *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1048–1073, 2013.

[66] J. Le Ny, E. Feron, and E. Frazzoli, “On the Dubins traveling salesman problem,” *IEEE Trans. Automat. Contr.*, vol. 57, pp. 265–270, 01 2012.

[67] J. Faigl, P. Váňa, M. Saska, T. Báča, and V. Spurný, “On solution of the Dubins touring problem,” in *2017 European Conference on Mobile Robots (ECMR)*, pp. 1–6, 2017.

[68] K. Savla, E. Frazzoli, and F. Bullo, “On the point-to-point and traveling salesperson problems for Dubins’ vehicle,” in *Proceedings of the 2005, American Control Conference, 2005.*, pp. 786–791 vol. 2, 2005.

[69] C. Noon and J. Bean, “An efficient transformation of the generalized traveling salesman problem,” *INFOR. Information Systems and Operational Research*, vol. 31, 02 1993.

[70] P. Vansteenwegen, W. Souffriau, and D. V. Oudheusden, “The orienteering problem: A survey,” *European Journal of Operational Research*, vol. 209, no. 1, pp. 1–10, 2011.

[71] R. Pěnička, J. Faigl, P. Váňa, and M. Saska, “Dubins orienteering problem,” *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, 02 2017.

[72] R. Pěnička, J. Faigl, and M. Saska, “Physical orienteering problem for unmanned aerial vehicle data collection planning in environments with obstacles,” *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 3005–3012, 2019.

[73] I. A. Sucan, M. Moll, and L. E. Kavraki, “The open motion planning library,” *IEEE Robotics Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[74] “The Open Motion Planning Library.” https://ompl.kavrakilab.org/index.html [online]. Accessed: 2022-03-28.

[75] J. Pan, S. Chitta, and D. Manocha, “FCL: A general purpose library for collision and proximity queries,” in *2012 IEEE International Conference on Robotics and Automation*, pp. 3859–3866, 2012.

[76] M. Moll, I. A. Sucan, and L. E. Kavraki, “Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization,” *IEEE Robotics Automation Magazine*, vol. 22, no. 3, pp. 96–102, 2015.

[77] P. L’Ecuyer, “Random number generation and quasi-Monte Carlo,” in *Encyclopedia of Actuarial Science*, Wiley, 09 2006.

[78] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, p. 3–30, jan 1998.

[79] M. Roberts, “How to generate uniformly random points on n-spheres and in n-balls.”

[80] M. E. Muller, “A note on a method for generating points uniformly on n-dimensional spheres,” *Commun. ACM*, vol. 2, p. 19–20, apr 1959.

[81] P. Jiménez, F. Thomas, and C. Torras, *Collision detection algorithms for motion planning*, pp. 305–343. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[82] S. S. Skiena, *The algorithm design manual.* London: Springer, 2nd ed ed., c2008.

[83] T. C. Project, *CGAL User and Reference Manual.* CGAL Editorial Board, 5.3.1 ed., 2021.

[84] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, p. 509–517, sep 1975.

[85] M. Muja and D. G. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration,” in *VISAPP*, 2009.

[86] G. Reinelt, “TSPLIB 95,” tech. rep., Institut für Angewandte Mathematik, Universität Heidelberg.

Appendix B

Detailed description of the input configuration file for the designed library

Below is a list of main nodes along with lists of their subnodes, detailed descriptions and requirements.

Node `problem` *(required)*

- `solver` – required, type of the planner, one of “sff”, “rrt”, “prm”, “lazy-tsp”, or “lazy-rrt”, and “lazy-sff” (combination of “rrt” setting with multiple cities corresponds to the Multi-T-RRT algorithm)
- `optimize` – required, whether to use asymptotically optimal versions of algorithms, boolean
- `connect-only` – optional, switch for SFF based algorithms to stop expansion after connecting all cities, boolean (default: “false”)
- `iterations` – required, maximum number of iterations, integer
- `dimension` – required, dimensionality of the solved problem, one of ”2D”, ”2DDubins”, ”3D”, ”3DDubins”, ”3DPolynom”

Node `delimiters` *(optional)*

- `standard` – optional, delimiter between values in OBJ file, character (default: ” ”)
- `name` – optional, delimiter of the name in OBJ file, character (default: ”_”)

Node `robot` *(required)*

- `path` – required, path to the file with robot model, string
- `type` – optional, type of the file, one of “tri”, “map” and “obj” (default: “map”)

Node `obstacles` *(optional)* – list, might contain more records

- `path` – required, path to the file with obstacle/environment model, string
- `position` – optional, position of the obstacle, string – point format (default: center of coordinate system)
- `type` – optional, type of the file, one of “tri”, “map” and “obj” (default: “map”)

Node `ranges` *(optional, when not specified, autorange is enabled)*

- `autorange` – required for planning without obstacles otherwise optional, enables deduction of sampling limits from the ranges of specified obstacles, boolean (default: false)

- `x` – required when autorange is disabled, limits of x-axis, string in format "[min; max]", where "min" and "max" are doubles
- `y` – required when autorange is disabled, limits of y-axis, string in format "[min; max]", where "min" and "max" are doubles
- `z` – required for three-dimensional spaces (3D, 3DDubins and 3DPolynom) when autorange is disabled, limits of z-axis, string in format "[min; max]", where "min" and "max" are doubles
- `pitch` – optional for 3D and 3DDubins problem dimensions otherwise ignored, minimum and maximum pitch angle during the flight, string in format "[min; max]", where "min" and "max" are doubles (default: for 3D from "-inf" to "+inf", for 3DDubins from "$-\pi/2$" to "$\pi/2$")

Node `TSP-solver` *(required for Lazy TSP, otherwise optional)*

- `type` – required, type of the embedded TSP solver to use, one of "lkh" and "concorde"
- `path` – optional, path to the solver executable, string (default: path to the embedded solver, according to `type`)

Node `algorithm` *(required when at least one of its subnodes is required, otherwise optional)*

- `m2r-ratio` – optional, ratio of path length to rotation (used for distance calculation), double (default: 1)
- `misses` – optional for SFF-based algorithms otherwise ignored, maximum number of failed expansions before moving a node from the open list to the closed list, integer (default: 3)
- `dubins-radius` – required when planning for Dubins vehicle or airplane otherwise ignored, minimum turning radius, double
- `dubins-resolution` – required when planning for Dubins vehicle or airplane otherwise ignored, number of ingoing/outgoing angles to/from each city node (the rotation of city is taken as an offset), integer, must be even and greater than 1 for multi-goal planning to ensure symmetry
- `bias` – optional, sampling bias for RRT-based or SFF-based algorithms (probability of selecting the node closest to the goal for expansion), double, for Lazy TSP and multi-goal RRT (Multi-T-RRT) must be equal to 0
- `prm-connections` – required for PRM otherwise ignored (incl. PRM*), number of neighboring nodes to connect to, integer

Node `cities` *(required)* – list, at least one record is required
Each record represents the coordinates of one city node, string in point format

Node `goal` *(optional)*
Record representing coordinates of the goal node, string in point format
Note: the behavior when entering multiple cities and the goal node is not defined

Node `distances` *(required)*

- `collision` – required, step size for the local planner checking possible collisions of the robot model with the environment, double
- `sampling` – required, distance between the parent node and its children, used for RRT-based and SFF-based algorithms, double
- `tree` – required, maximum distance between trees to connect them (multi-goal) or tolerance around the goal node (single-goal), double

Node `dynamics` *(required for planning with consideration of vehicle's dynamics)*

- `min-thrust` – optional, minimum possible thrust on the planned trajectory, double (default: 0)
- `max-thrust` – optional, maximum possible thrust on the planned trajectory, double (default: 1)
- `max-rotation-speed` – optional, maximum achievable rotation speed, double (default: 1)
- `gravity` – optional, intensity of gravitational acceleration, directed in negative direction of z-axis, double (default: 9.81)
- `segment-time` – required, time between the parent node and its children, double
- `control-interval` – optional, time interval for feasibility check, double (default: 0.01)

Node `save` *(optional)*

- `goals` – optional, file to save coordinates of all cities and the goal node
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of "tri", "map" and "obj" (default: "map")
- `tree` – optional, file to save final "planning graph" after expansion
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of "tri", "map" and "obj" (default: "map")
  - `frequency` – optional, number of iterations between two consecutive tree saves – enables progressive saving of the graph when not equal to 0
- `roadmap` – optional, file to save final path(s)
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of "tri", "map" and "obj" (default: "map")
- `params` – optional, file to save the output values – identifier, number of elapsed iterations, whether the problem was solved, the order of the cities, the length of paths between them (not printed out for Dubins problems), total length of the tour (when it was found) and elapsed time
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of "tri", "map" and "obj" (default: "map")
  - `id` – optional, identifier of the problem, string
- `TSP-file` – optional, file to save the distance matrix for external TSP solvers

  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of “tri”, “map” and “obj” (default: “map”)
- `TSP-paths` – optional, file to save the final paths in the final tour between cities
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of “tri”, “map” and “obj” (default: “map”)
- `frontiers` – optional for SFF-based algorithms otherwise ignored, file to save nodes in open list in actual iteration
  - `path` – required, path to the file for saving, string
  - `type` – optional, type of the file, one of “tri”, “map” and “obj” (default: “map”)
  - `frequency` – optional, number of iterations between two consecutive saves – enables progressive saving of the graph when not equal to 0

B.1 Point format

Some of the abovementioned nodes have a prescribed specific string format of string denoted as ”point format”. This has not benn further explained before, as the precise format depends on the dimension and type of the problem. The common format of the point is as follows:

$$[x; y; \ldots]$$

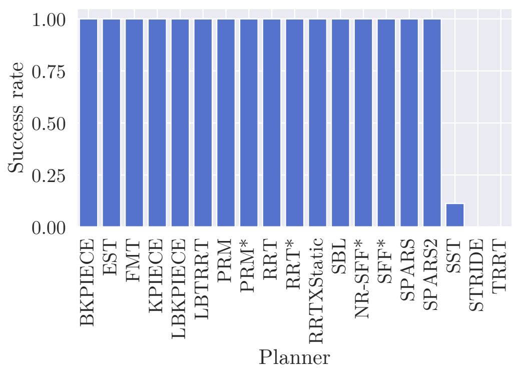
The number of elements is 2 for 2D problems, 3 for 2DDubins problems (x, y, angle), 6 for 3D problems (x, y, z, yaw, pitch, roll), 5 for 3DDubins problems (x, y, z, yaw, pitch) and 6 for 3DPolynom problems (same as for 3D problems). Each element is double, elements must be separated by a semicolon with optional spaces.

Appendix C

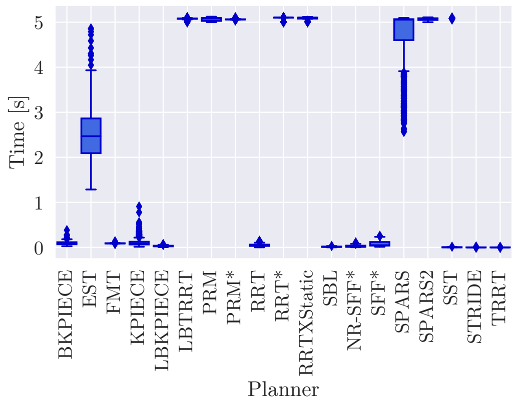
Complete results of single-goal benchmarks

On the following pages the complete results for single-goal path planning benchmarks are available in form of the graphs, generated either from the results of OMPL implementations or the implementations in the custom library.

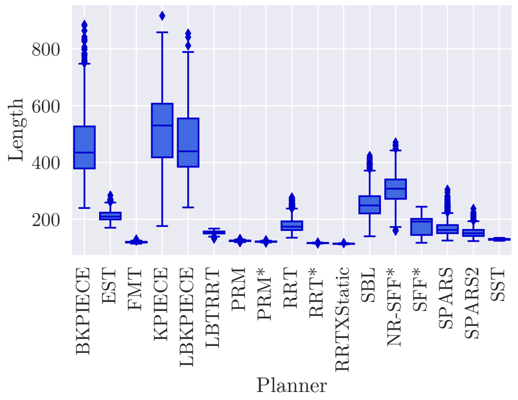
C.1 Planning in 2D Euclidean space



(a) : Success rates

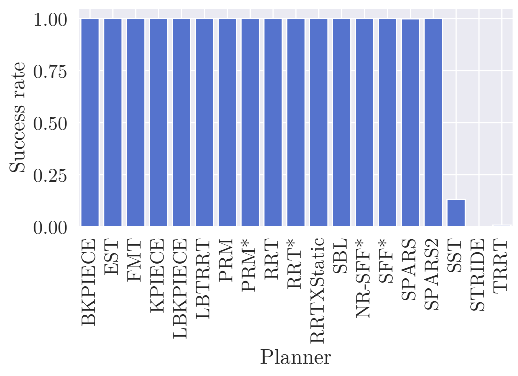


(b) : Times of computations

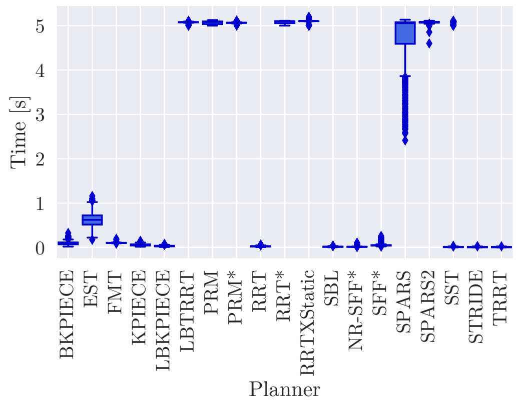


(c) : Lengths of solutions

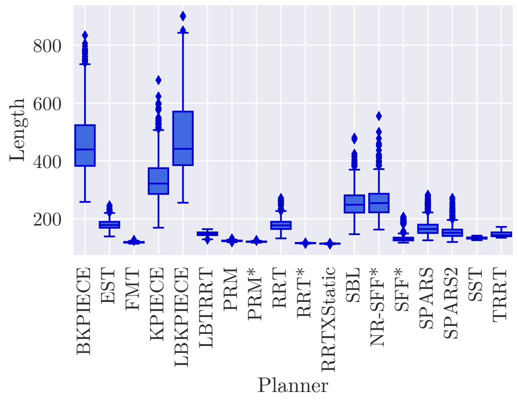
Figure C.1: Results for "Bugtrap in" problem



(a) : Success rates

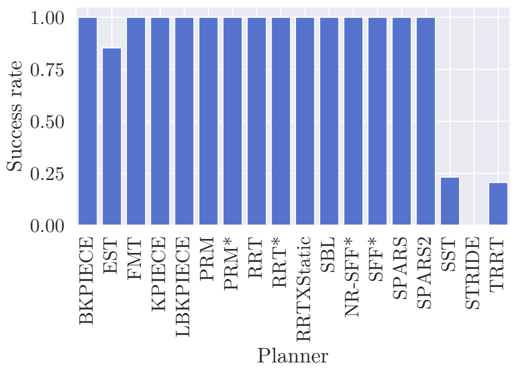


(b) : Times of computations

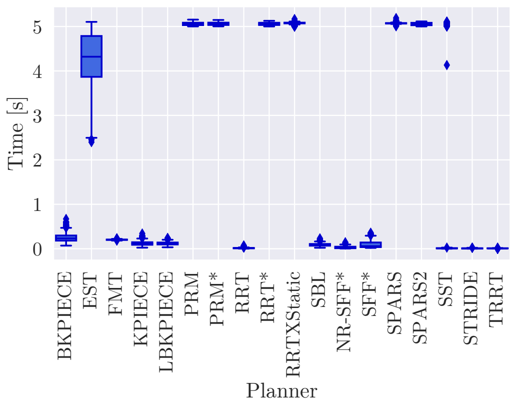


(c) : Lengths of solutions

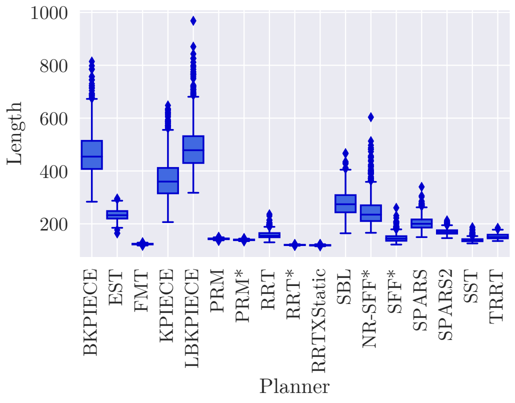
Figure C.2: Results for "Bugtrap out" problem



(a) : Success rates

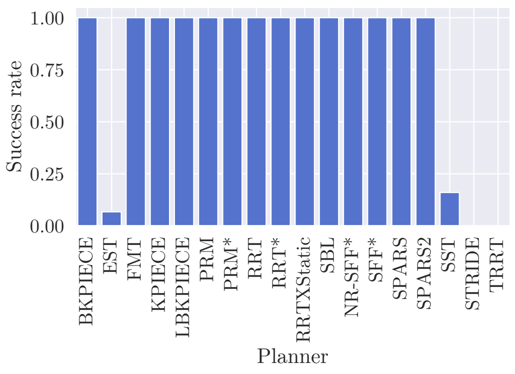


(b) : Times of computations

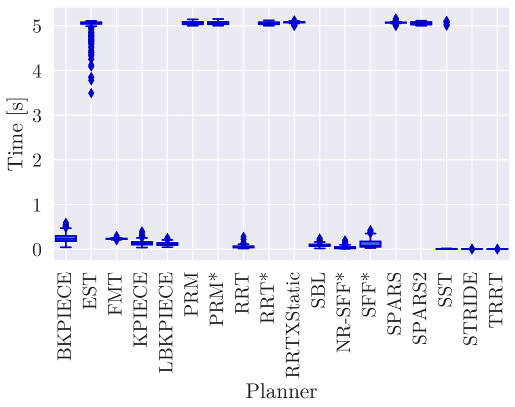


(c) : Lengths of solutions

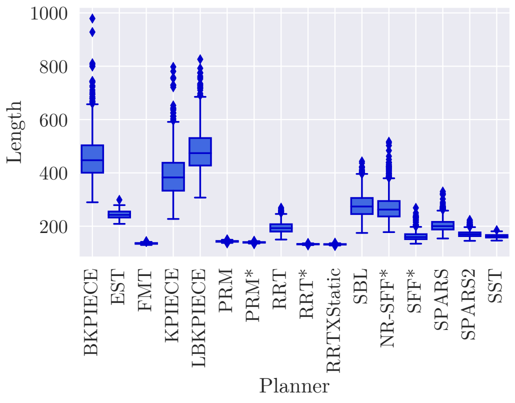
Figure C.3: Results for "Dense 0" problem



(a) : Success rates

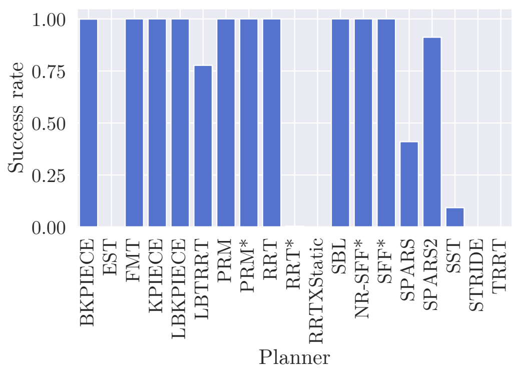


(b) : Times of computations

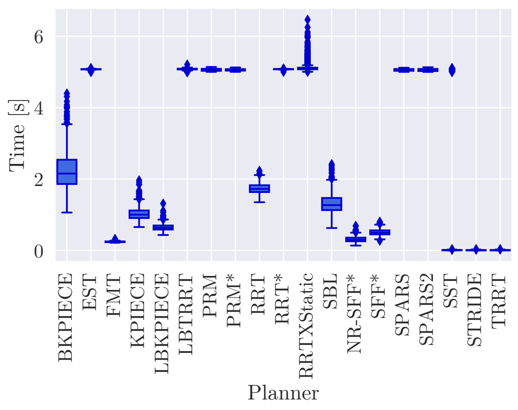


(c) : Lengths of solutions

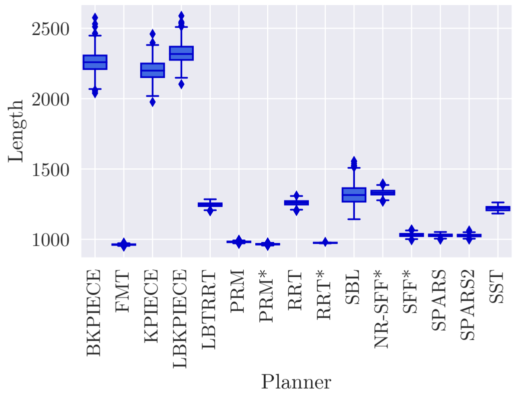
Figure C.4: Results for "Dense 1" problem



(a) : Success rates

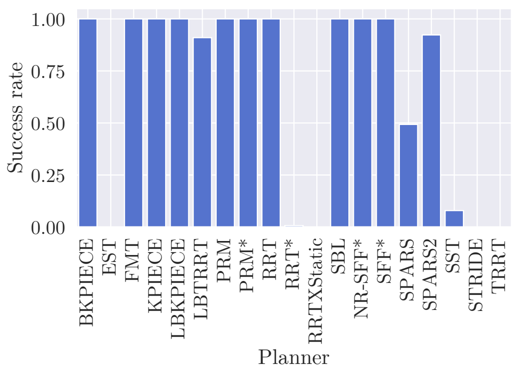


(b) : Times of computations

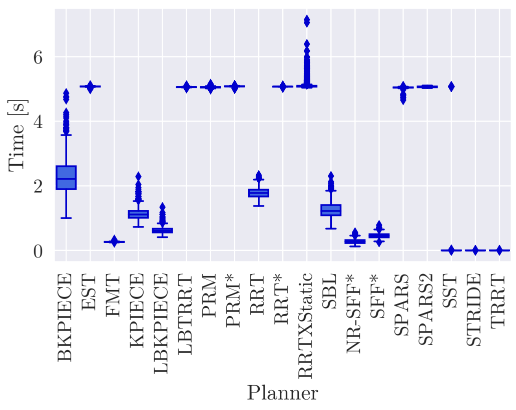


(c) : Lengths of solutions

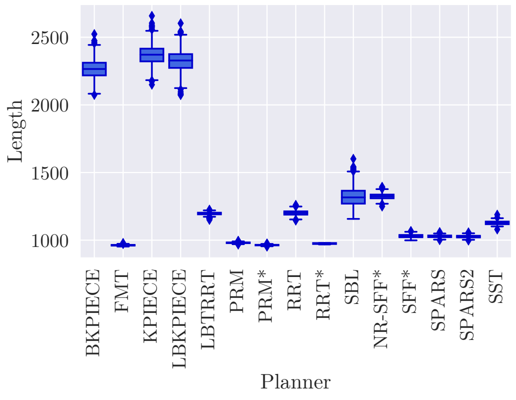
Figure C.5: Results for “Spiral in” problem



(a) : Success rates

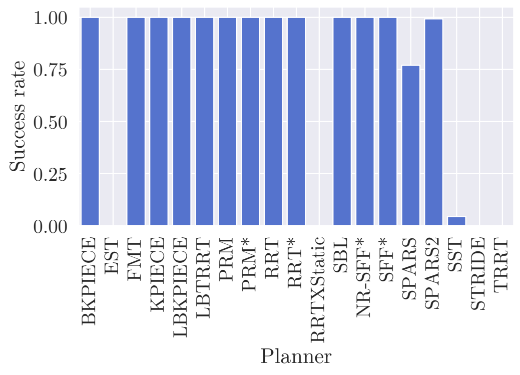


(b) : Times of computations

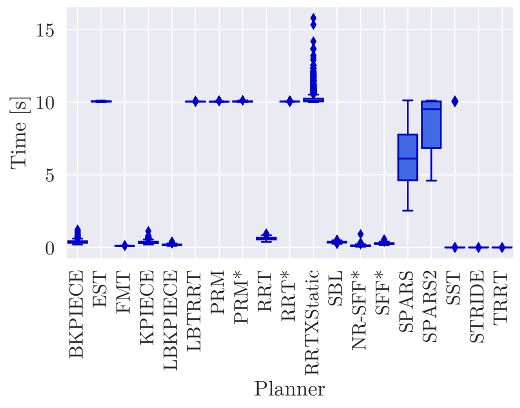


(c) : Lengths of solutions

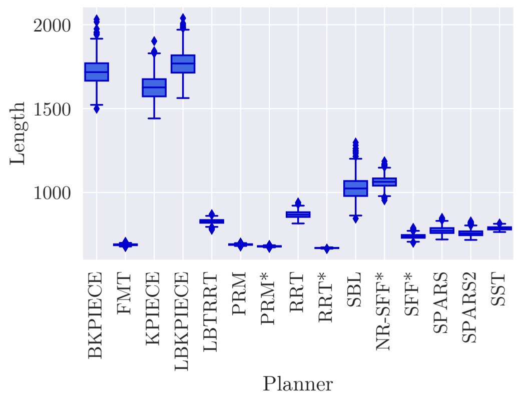
Figure C.6: Results for “Spiral out” problem



(a) : Success rates

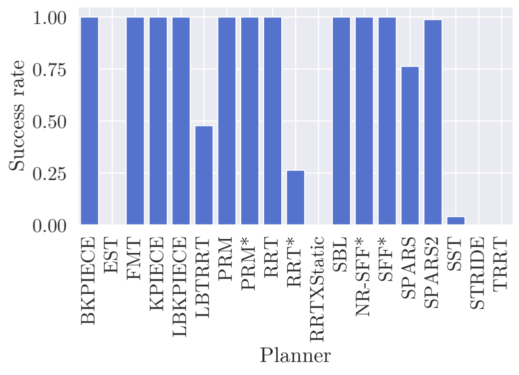


(b) : Times of computations

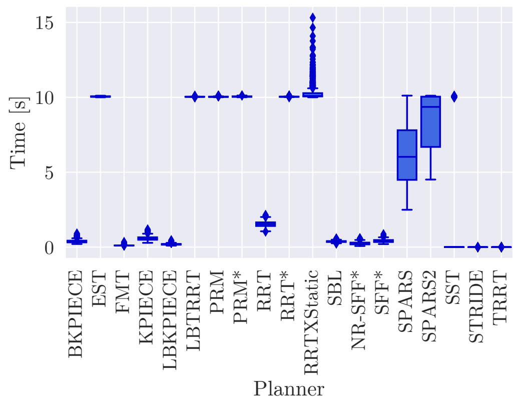


(c) : Lengths of solutions

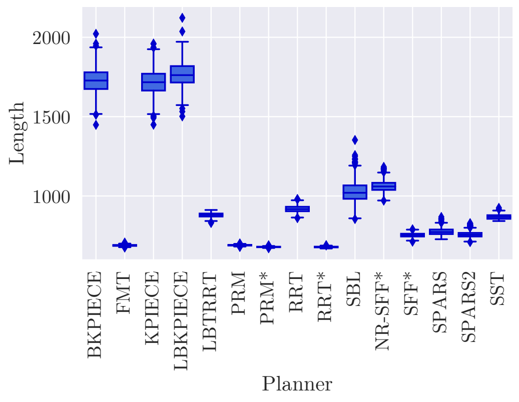
Figure C.7: Results for "Maze corridor" problem



(a) : Success rates



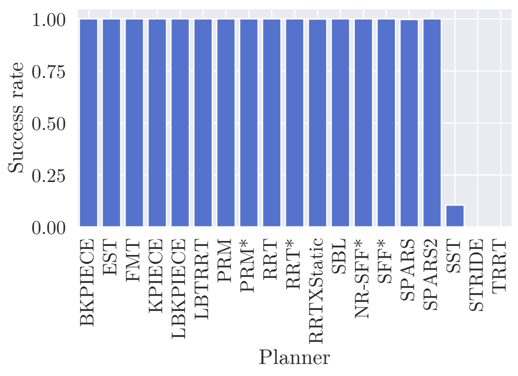
(b) : Times of computations



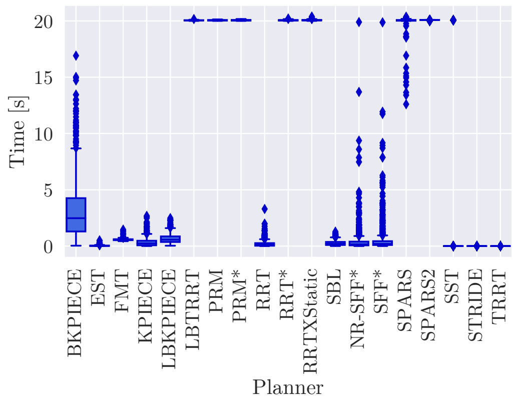
(c) : Lengths of solutions

Figure C.8: Results for "Maze walls" problem

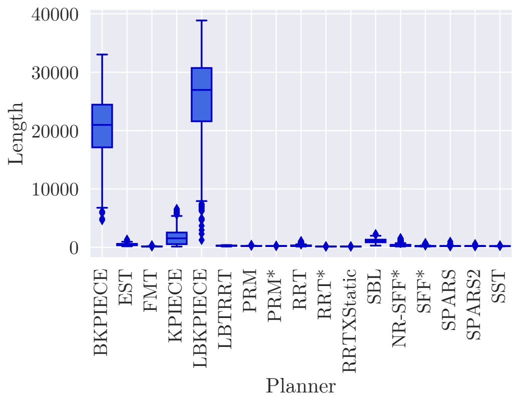
C.2 Planning in 3D Euclidean space



(a) : Success rates

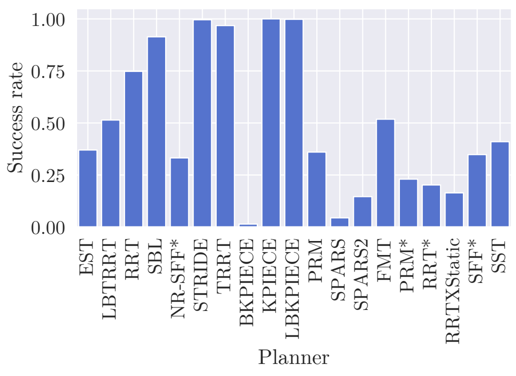


(b) : Times of computations

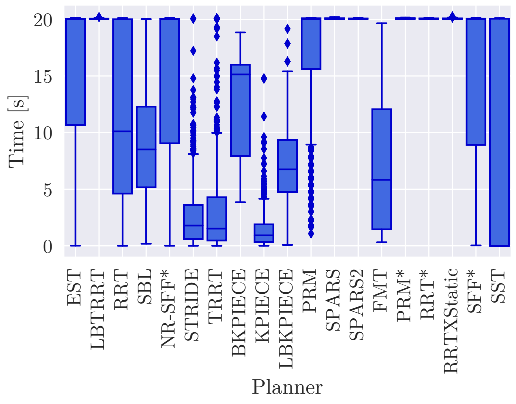


(c) : Lengths of solutions

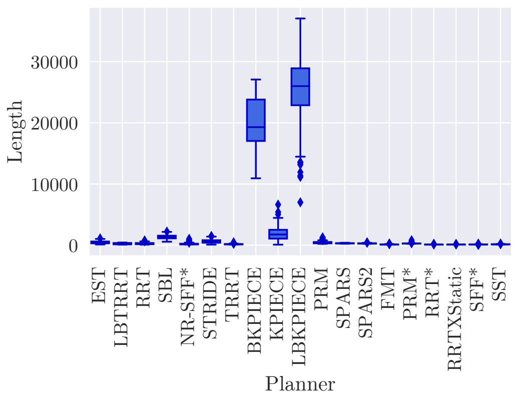
Figure C.9: Results for "Easy" problem



(a) : Success rates



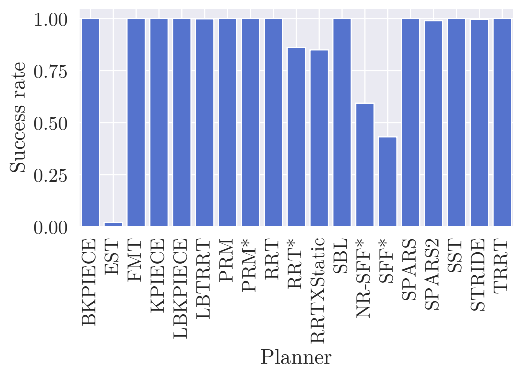
(b) : Times of computations



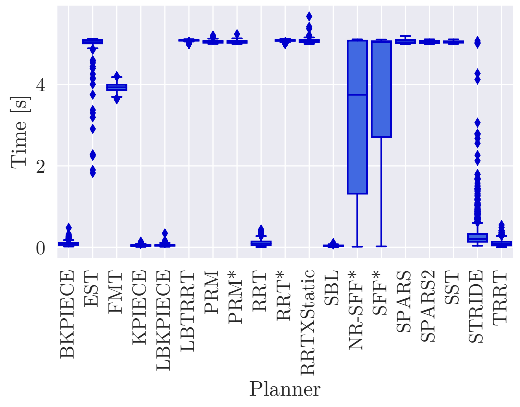
(c) : Lengths of solutions

Figure C.10: Results for "Twistycool" problem

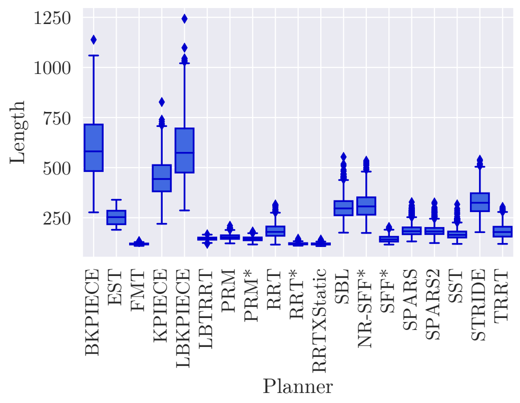
C.3 Planning for Dubins car



(a) : Success rates

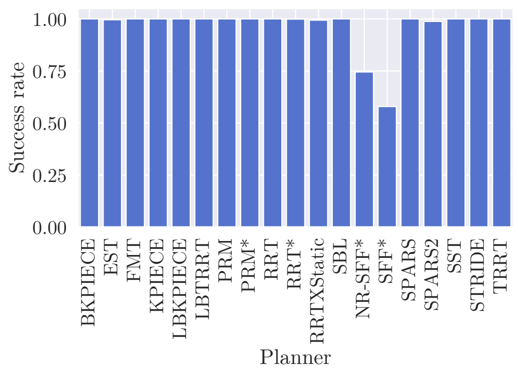


(b) : Times of computations

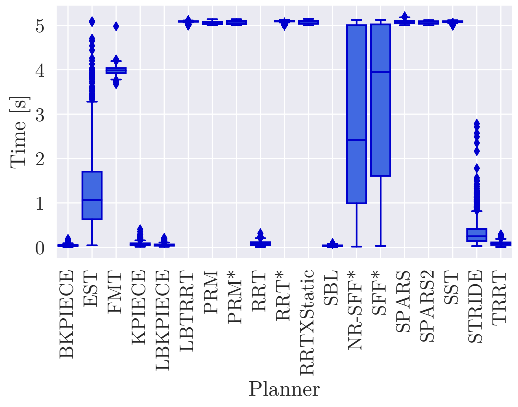


(c) : Lengths of solutions

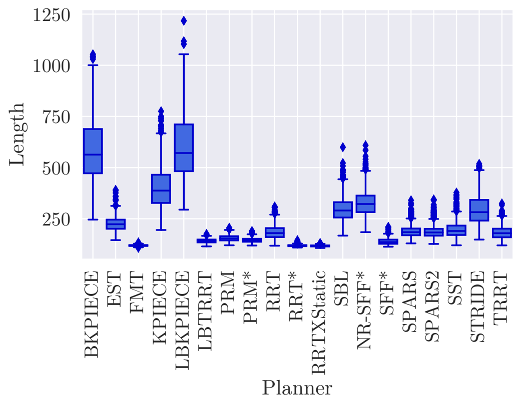
Figure C.11: Results for “Bugtrap in” problem



(a) : Success rates

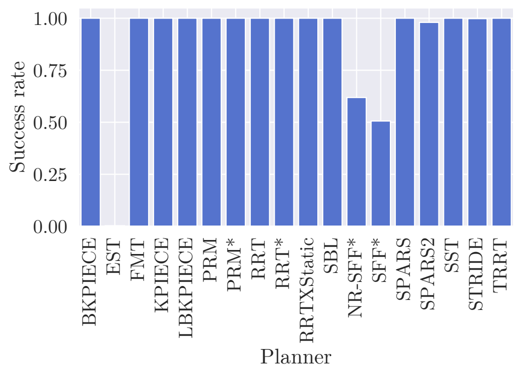


(b) : Times of computations

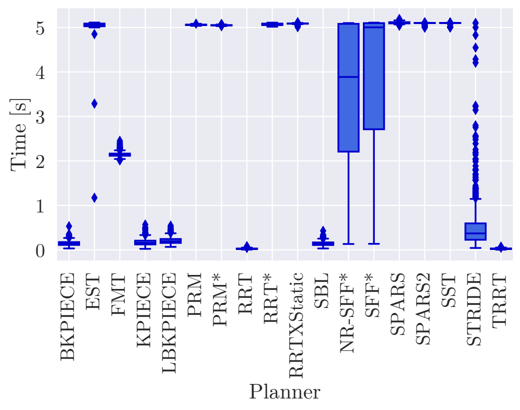


(c) : Lengths of solutions

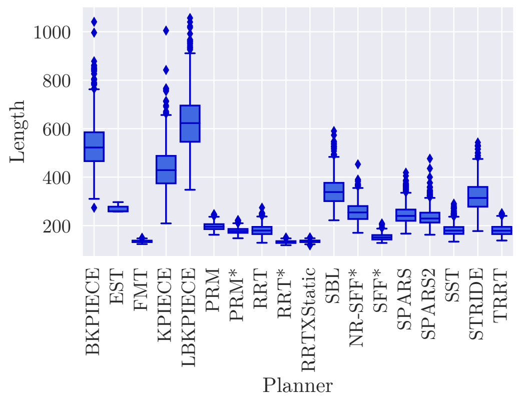
Figure C.12: Results for “Bugtrap out” problem



(a) : Success rates

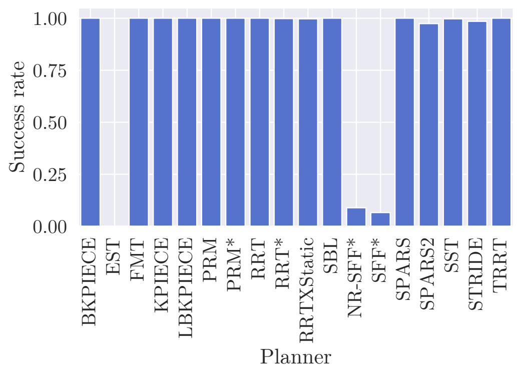


(b) : Times of computations

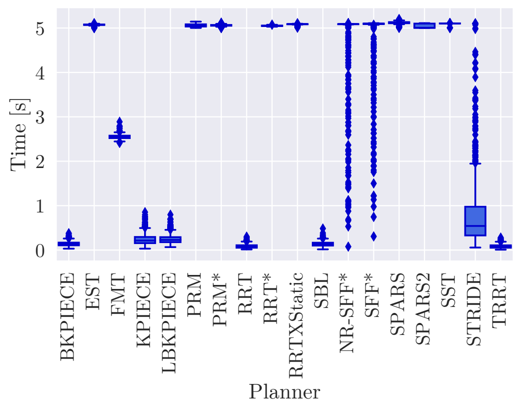


(c) : Lengths of solutions

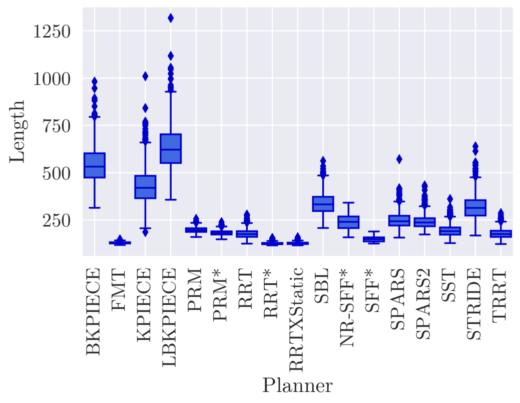
Figure C.13: Results for “Dense 0” problem



(a) : Success rates

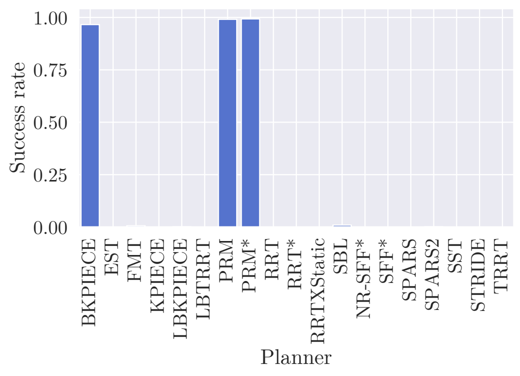


(b) : Times of computations

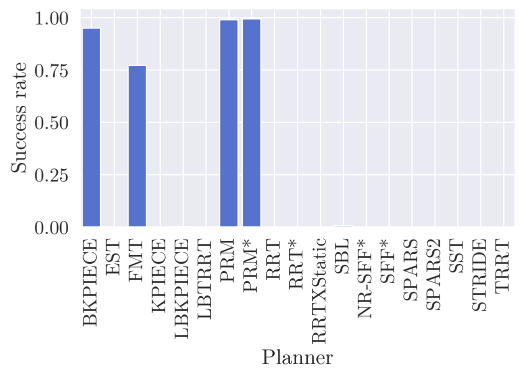


(c) : Lengths of solutions

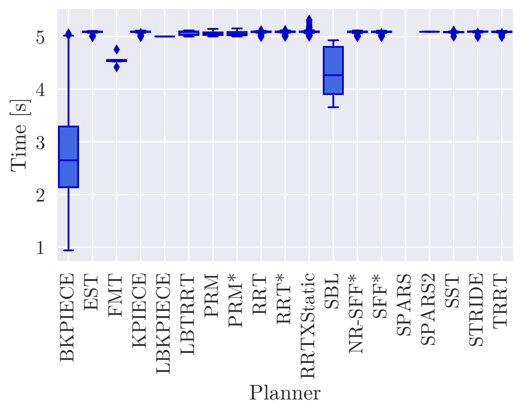
Figure C.14: Results for “Dense 1” problem



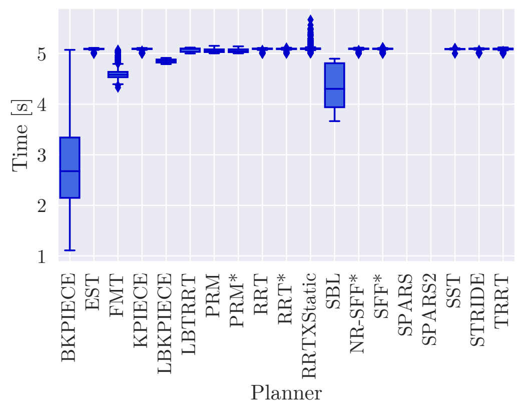
(a) : Success rates



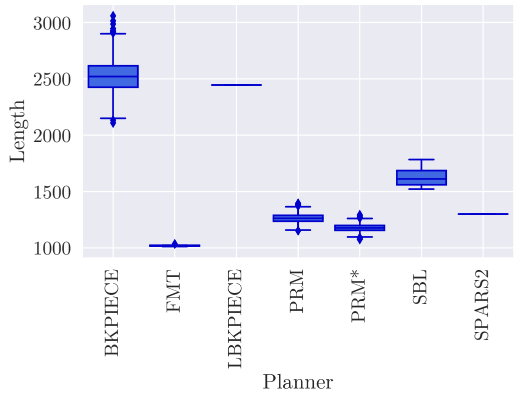
(a) : Success rates



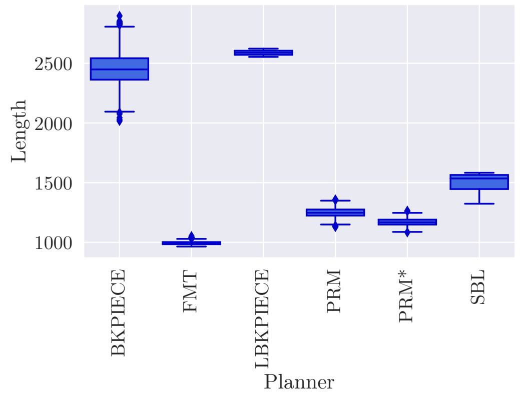
(b) : Times of computations



(b) : Times of computations



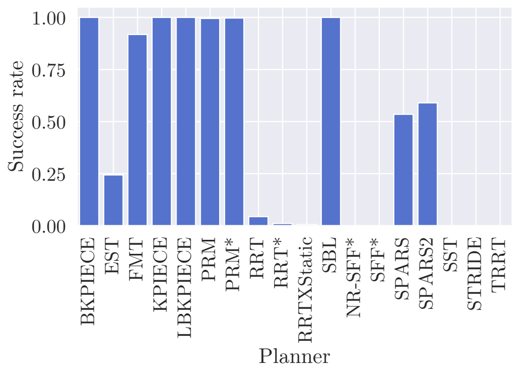
(c) : Lengths of solutions



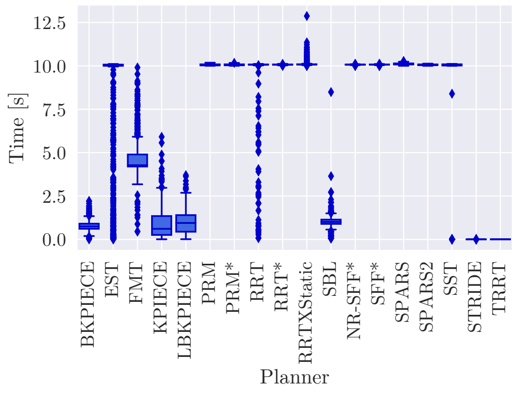
(c) : Lengths of solutions

Figure C.15: Results for “Spiral in” problem

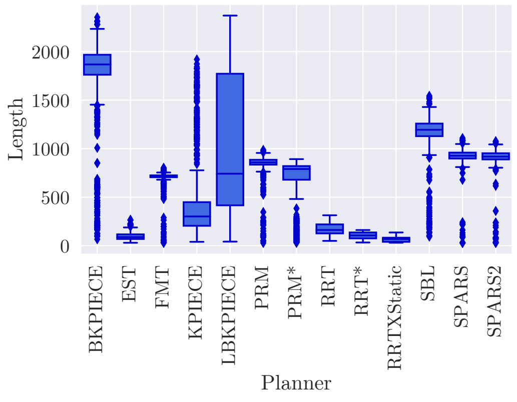
Figure C.16: Results for “Spiral out” problem



(a) : Success rates

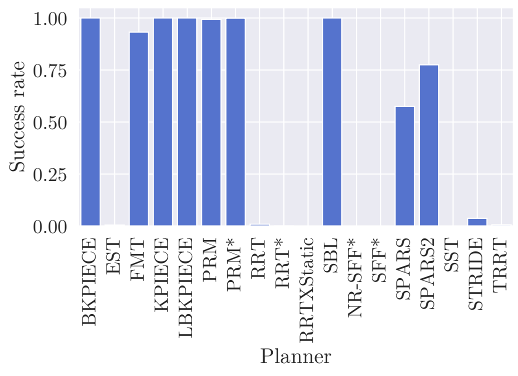


(b) : Times of computations

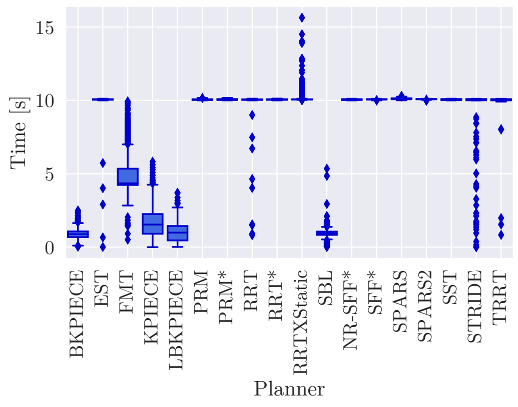


(c) : Lengths of solutions

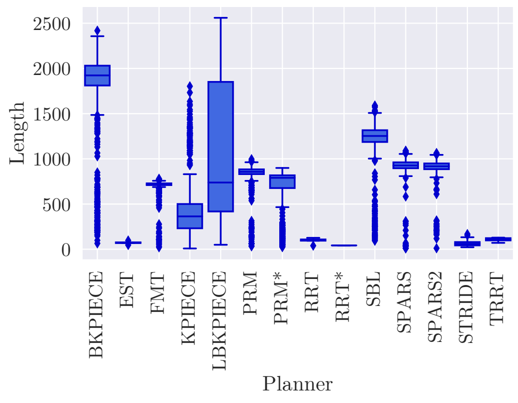
Figure C.17: Results for “Maze corridor” problem



(a) : Success rates



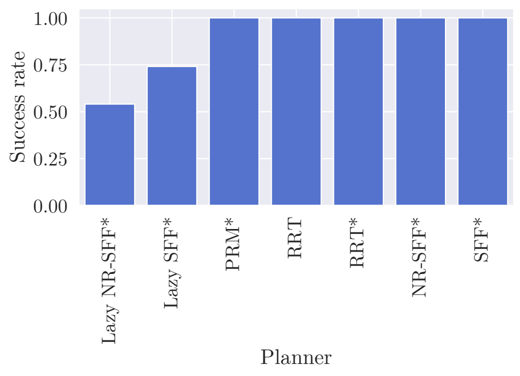
(b) : Times of computations



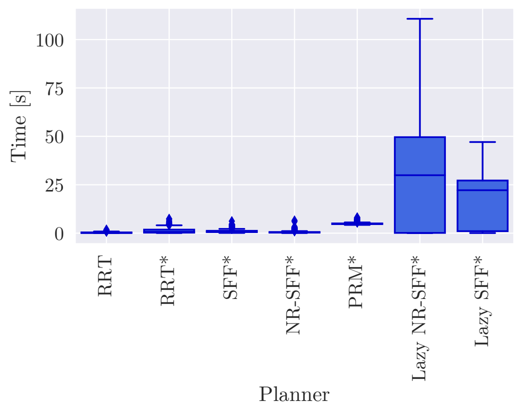
(c) : Lengths of solutions

Figure C.18: Results for “Maze walls” problem

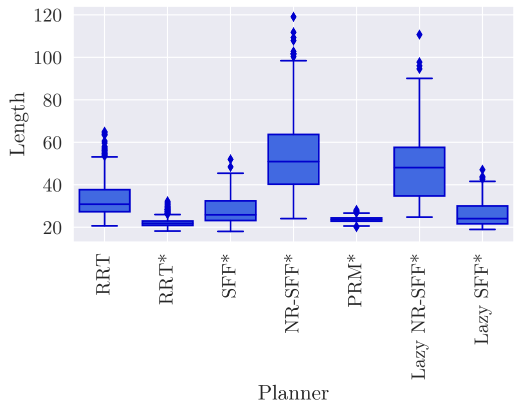
C.4 Planning for Dubins airplane



(a) : Success rates

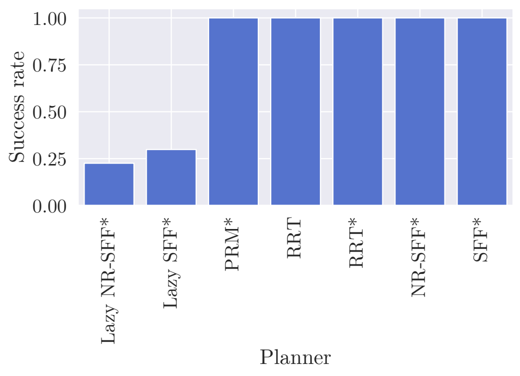


(b) : Times of computations

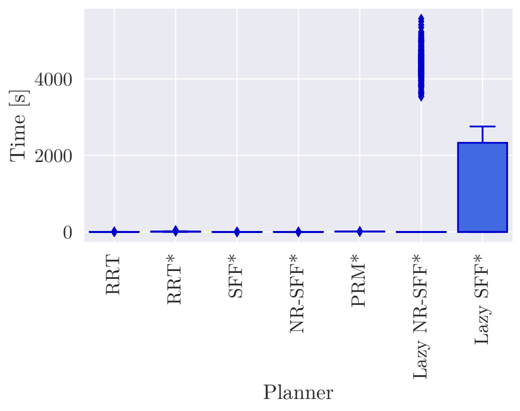


(c) : Lengths of solutions

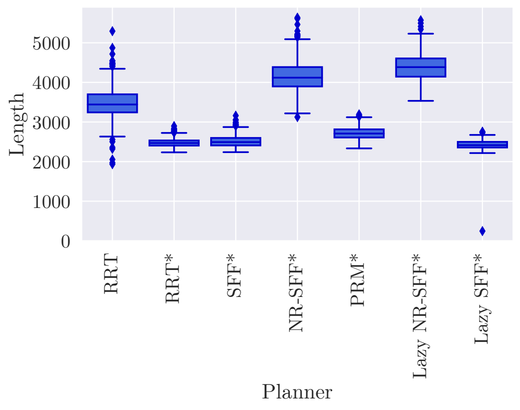
Figure C.19: Results for “Building” problem



(a) : Success rates

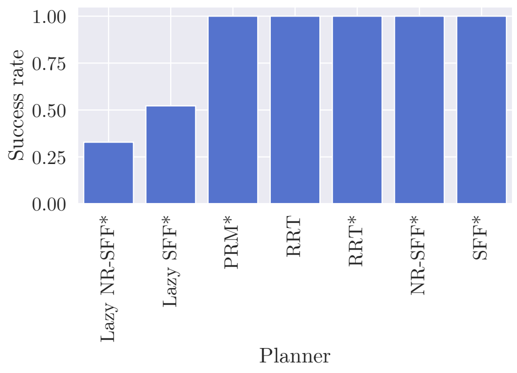


(b) : Times of computations

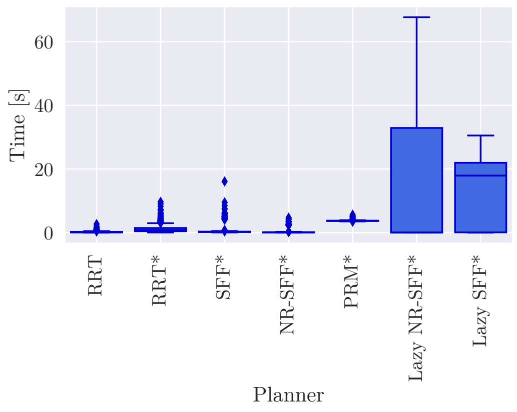


(c) : Lengths of solutions

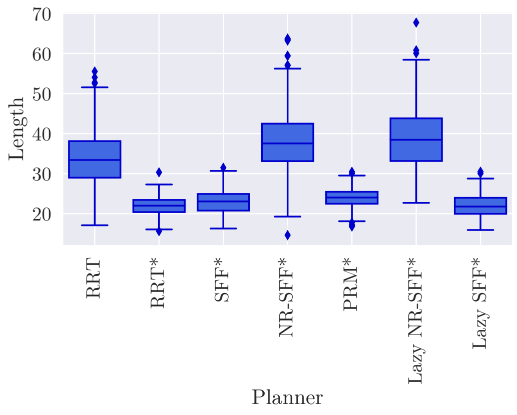
Figure C.20: Results for “Dense” problem



(a) : Success rates



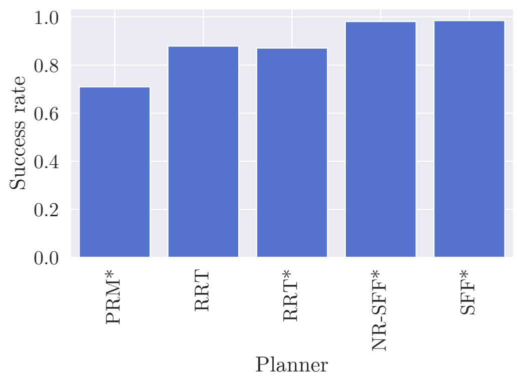
(b) : Times of computations



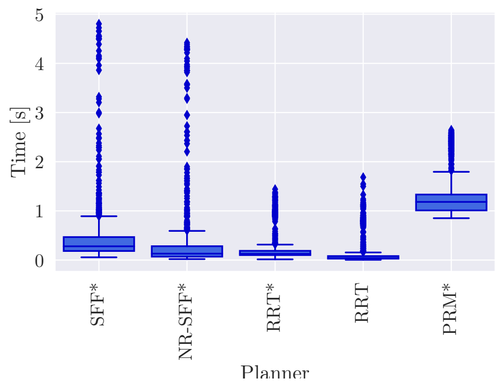
(c) : Lengths of solutions

Figure C.21: Results for “Triangles” problem

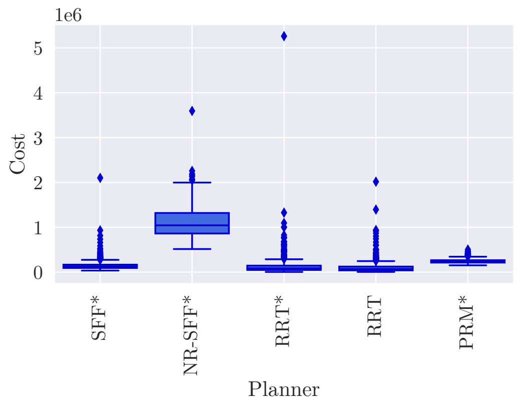
C.5 Planning on polynomial trajectories in 2D



(a) : Success rates

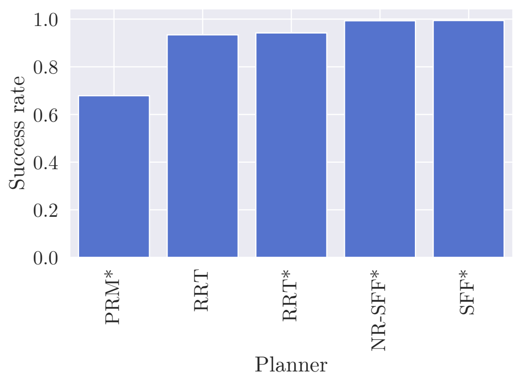


(b) : Times of computations

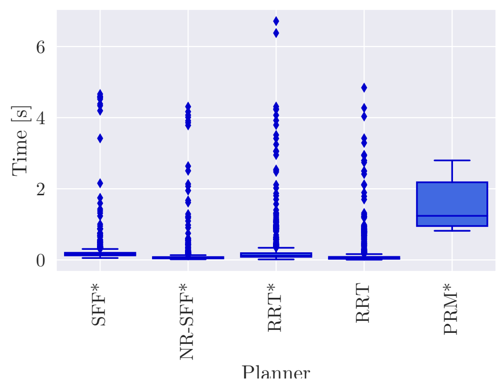


(c) : Costs of solutions

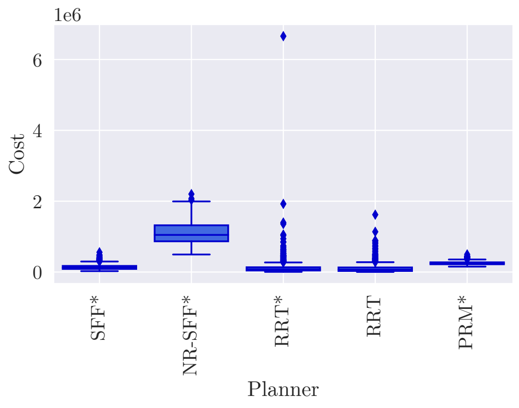
Figure C.22: Results for "Bugtrap in" problem



(a) : Success rates

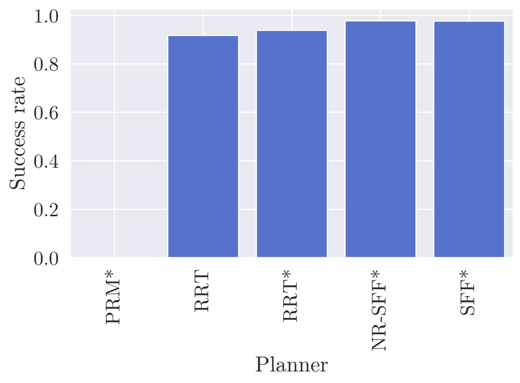


(b) : Times of computations

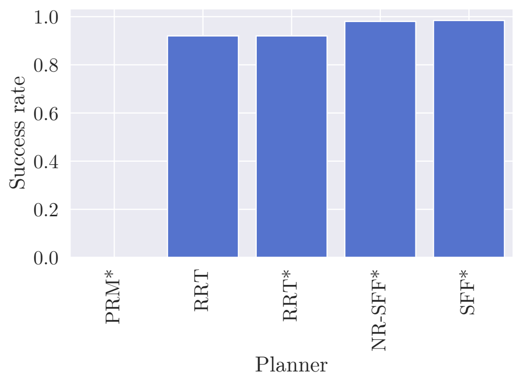


(c) : Costs of solutions

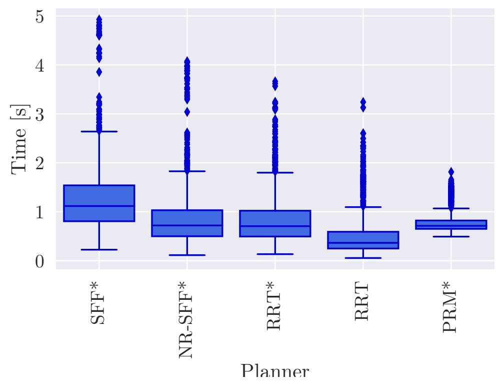
Figure C.23: Results for "Bugtrap out" problem



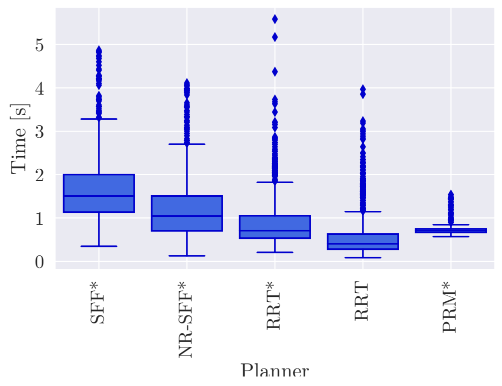
(a) : Success rates



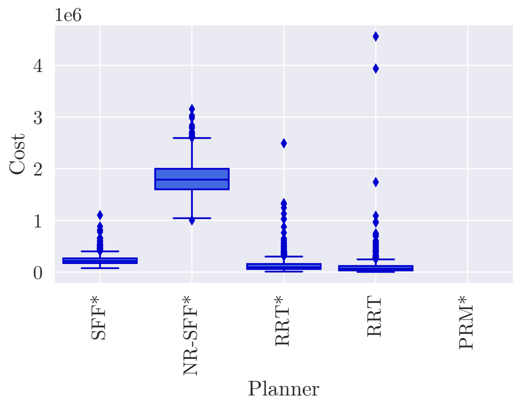
(a) : Success rates



(b) : Times of computations

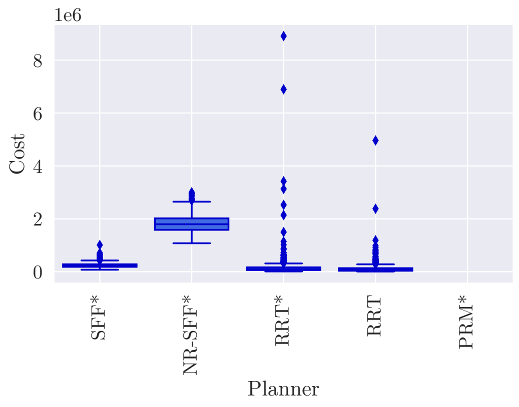


(b) : Times of computations



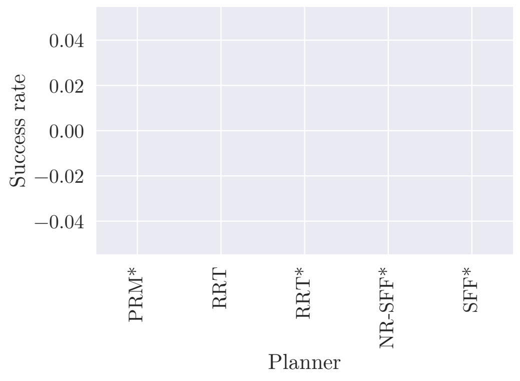
(c) : Costs of solutions

Figure C.24: Results for “Dense 0” problem

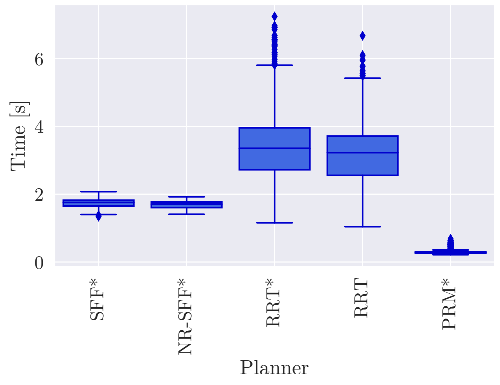


(c) : Costs of solutions

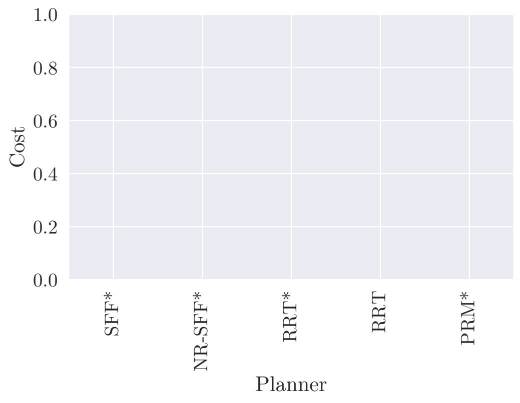
Figure C.25: Results for “Dense 1” problem



(a) : Success rates

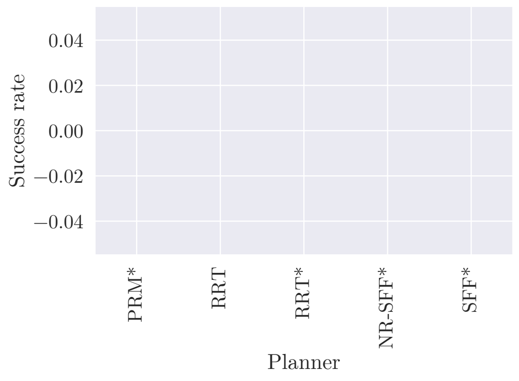


(b) : Times of computations

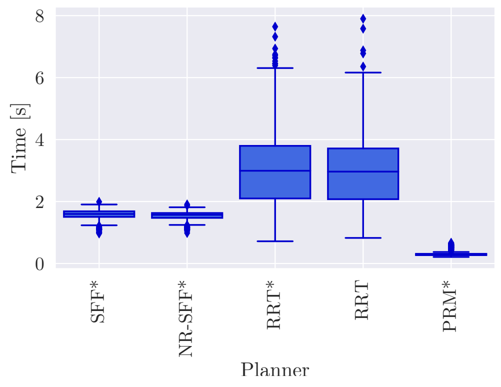


(c) : Costs of solutions

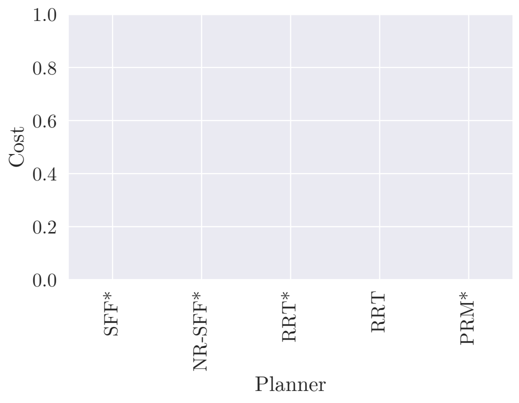
Figure C.26: Results for "Spiral in" problem



(a) : Success rates

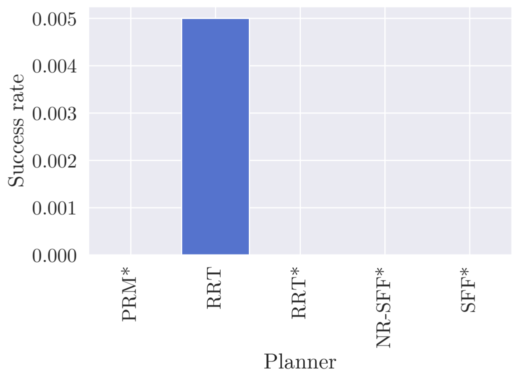


(b) : Times of computations

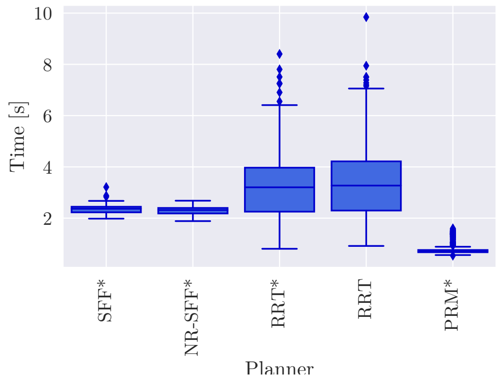


(c) : Costs of solutions

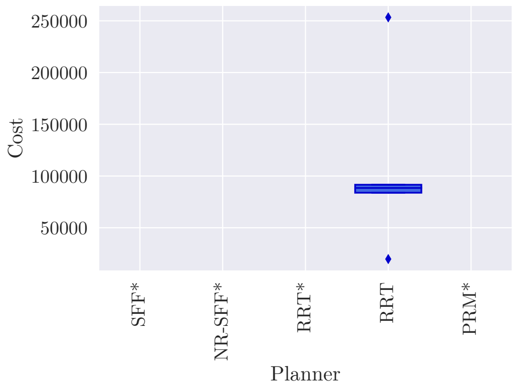
Figure C.27: Results for "Spiral out" problem



(a) : Success rates

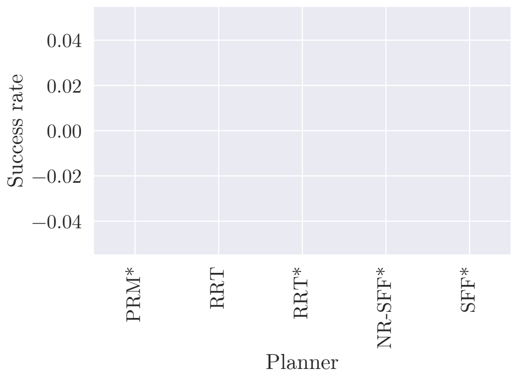


(b) : Times of computations

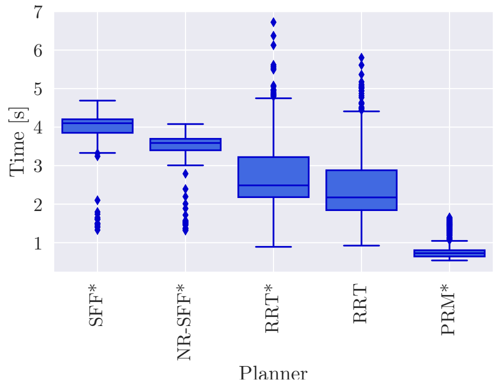


(c) : Costs of solutions

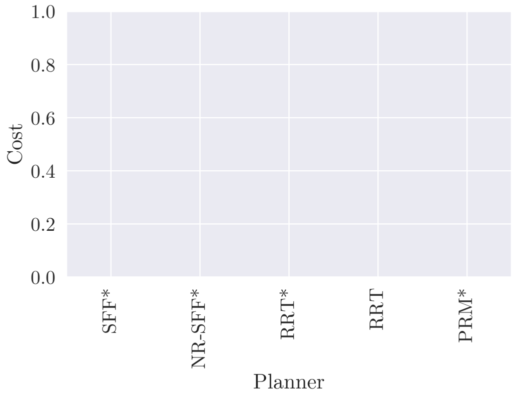
Figure C.28: Results for "Maze corridor" problem



(a) : Success rates



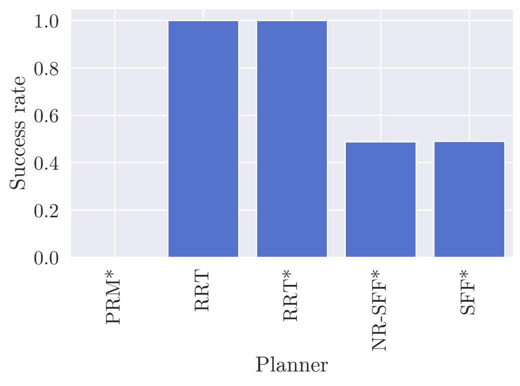
(b) : Times of computations



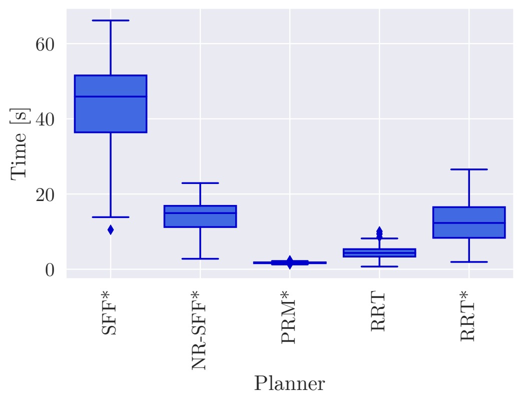
(c) : Costs of solutions

Figure C.29: Results for "Maze walls" problem

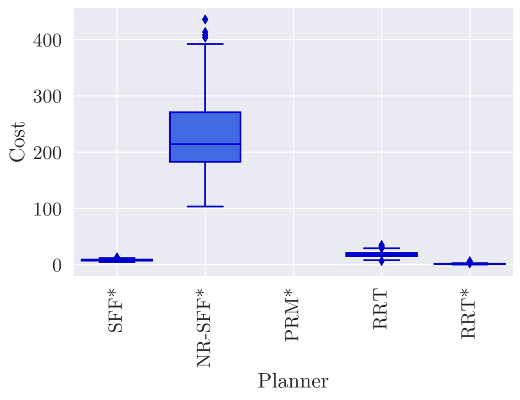
C.6 Planning on polynomial trajectories in 3D



(a) : Success rates

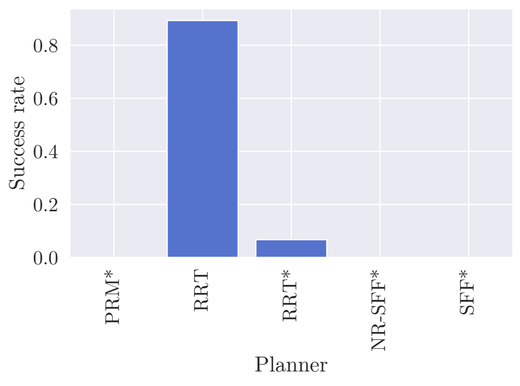


(b) : Times of computations

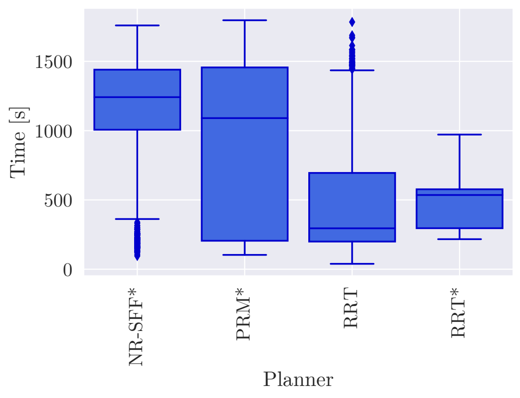


(c) : Costs of solutions

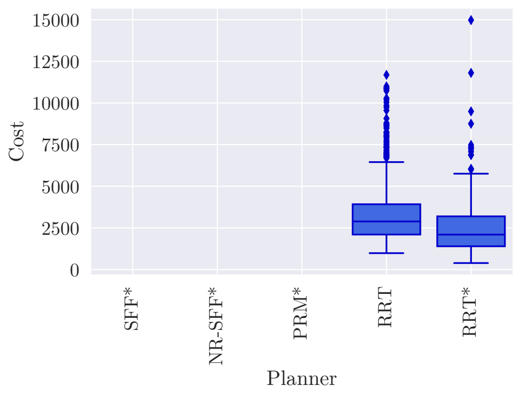
Figure C.30: Results for "Building" problem



(a) : Success rates

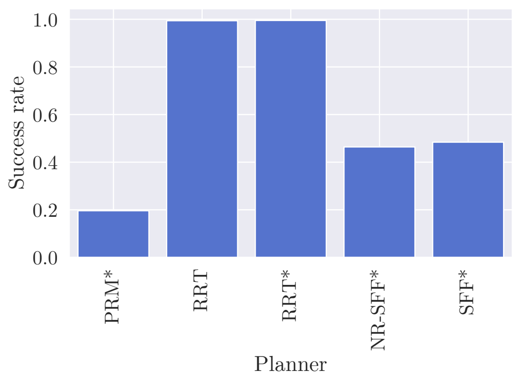


(b) : Times of computations

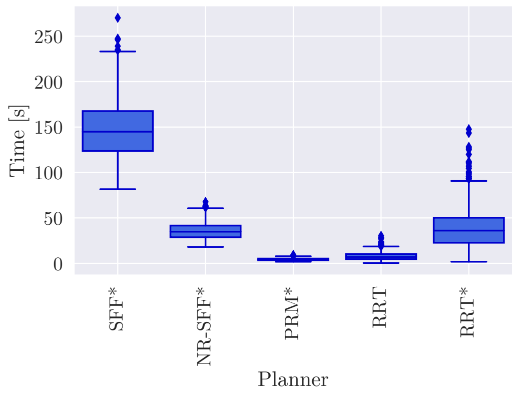


(c) : Costs of solutions

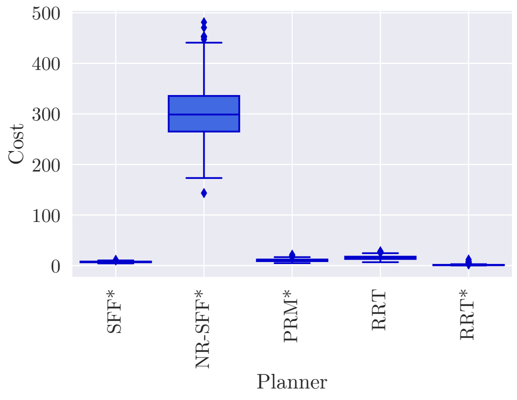
Figure C.31: Results for "Dense" problem



(a) : Success rates



(b) : Times of computations



(c) : Costs of solutions

Figure C.32: Results for “Triangles” problem

Appendix D
Attachments

The attached ZIP file has following structure:

- `library_planning_algorithms` – Source codes of the created library of planning algorithms. Instructions on how to install it and run it are provided in the included README.md file.
- `omplapp` – Source codes of OMPL and OMPL.app extended by the implementations of NR-SFF* (registered as "sff") and SFF* (registered as "sffstar").
- `thesis` – Source codes and the final PDF version of this thesis.