Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics and Robotics

Methods for sampling configuration space

Vít Železný

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Železný Vít** Personal ID number: **507273**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Methods for sampling configuration space

Bachelor’s thesis title in Czech:

Metody vzorkování konfiguračního prostoru

Guidelines:

1. Study path planning problem [1] and get familiar with sampling-based path planning methods (e.g., RRT and RRT*) [1,2,3]. Implement basic RRT in C/C++ or Python.
2. Modify the RRT-based planner to generate the random samples along several predetermined primitives. A primitive should specify how to sample certain regions in the configuration space. Consider a 2D map/robot with rotation (3D configuration space) and a 2D multiple-link robot. Design the primitives acquisition by hand for each scenario.
3. Extend the method from task 2) to 3D robots and obstacles (6D configuration space).
4. Design and implement automatic acquisition of sampling primitives for 2D and 3D cases. The method should automatically find several sampling primitives based only on the shape of the robot and obstacles.
5. Experimentally verify all implemented methods and compare them with state-of-the-art planners provided by the OMPL library [4]. Perform experiments in both 2D and 3D cases. Consider both convex and non-convex shapes of the robots and multi-link robots in the experiments.

Bibliography / sources:

[1] LaValle, Steven M. Planning Algorithms. 1st ed. Cambridge University Press, 2006. https://doi.org/10.1017/CBO9780511546877.
[2] LaValle, Steven. "Rapidly-exploring random trees: A new tool for path planning." Research Report 9811 (1998).
[3] Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. The international journal of robotics research, 30(7), 846-894.
[4] Mark Moll, Ioan A. Şucan, Lydia E. Kavraki, Benchmarking Motion Planning Algorithms: An Extensible Infrastructure for Analysis and Visualization, IEEE Robotics & Automation Magazine, 22(3):96–102, September 2015.

Name and workplace of bachelor’s thesis supervisor:

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

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

Date of bachelor’s thesis assignment: **23.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

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

III. Assignment receipt

The student acknowledges that the bachelor'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 bachelor'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 thank my supervisor, *Vojtěch Vonásek*, for his guidance and patience, he has been showing me these past two years. Without him, I might have completely missed out on the work done on the Faculty and the experience of researching something, that might get used by others.

I would also like to thank my friend, *Jakub Jandus*, for always either dragging me, or letting himself get dragged by me into various engineering "adventures", which we both always ended up really enjoying.

Finally, I would like to thank my family, for supporting me through these Cybernetics endeavours, despite all of the things I learn being so foreign to them.

Thanks!

Declaration

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

.....
podpis autora práce

Author statement for undergraduate thesis:

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 praparation of university theses.

Prague, data .....

.....
Signature

Abstract

Sampling based motion planning provides efficient solutions to many otherwise difficult path planning problems. However, it also suffers greatly in environments with dense obstacles or narrow passages. In this thesis we shall discuss existing approaches to sampling based motion planning, provide a novel way of sampling configuration space by using a database of path primitives, and show its utilization for the Rapidly-exploring random tree algorithm.

Keywords: Sampling based motion planning, Sampling configuration space, RRT

Supervisor: Ing. Vojtěch Vonásek, Ph.D.

Abstrakt

Vzorkovací metody plánování pohybu nabízí efektivní řešení řady jinak složitých problémů plánování tras. Zároveň se však jejich účinnost značně zhoršuje v prostředích s četnými překážkami či úzkými průchody. V této práci nejdříve uvedeme existující přístupy k vzorkovacím metodám plánování pohybu, dále představíme nový způsob vzorkování konfiguračního prostoru využívající databázi primitiv cest a ukážeme jeho využití pro Rapidly-exploring random tree algoritmus.

Klíčová slova: Vzorkovací metody plánování pohybu, Vzorkování konfiguračního prostoru, RRT

Překlad názvu: Metody vzorkování konfiguračního prostoru

Contents

Figures

Chapter 1

Introduction

The aim of this thesis is to present a new method of generating samples in the configuration space applicable for sampling based planning (further shortened as SBP) algorithms, specifically on the RRT (Rapidly-exploring random trees [1]) algorithm.

Many problems related to manipulating objects in space can be viewed as a problem of path planning — finding a collision free path for such an object from one point to another. Hence, path planning algorithms can be considered a core component of many robotic projects, are greatly utilized in game development or even in areas such as studies of protein folding. Many of these problems can be tackled using simple deterministic path planning algorithms based on graph search methods such as Dijkstra [11] and A* [12]. This approach offers an efficient way to find optimal paths for many simple problems. Even more complex problems such as those burdened with continuous variables can be altered to use these deterministic methods. The simplest way to do so being the discretization of said continuous variables and converting the entire problem into a graph. However, using this approach we can often lead to creating large and possibly complex graphs, thus making usual graph-based planning algorithms computationally expensive.

The other way to handling planning with continuous variables is to use SPB algorithms such as RRT, PRM (Probabilisitic roadmap) [4] or EST (Expansive state trees) [5]. These algorithms provide a unique approach to path planning by sampling of the configuration space and connecting these samples to either directly create the desired path or construct an efficient connected graph, which can be further explored by the aforementioned graph search algorithms. SPB can be very advantageous in terms of providing a fast solution to even complex problems. On the other hand the resulting solutions are usually suboptimal. Furthermore, due to the randomness involved, these algorithms are not guaranteed to find a solution, even if one exists. This can be especially noticeable in problems with high amounts of obstacles and many degrees of freedom (DOF), where collision configurations are frequent. Some authors have already attempted to tackle this problem by introducing non-uniform sampling methods. The effects of different sampling can be immediately seen in figure 1.1.

This thesis introduces an approach to enhance the RRT algorithm and

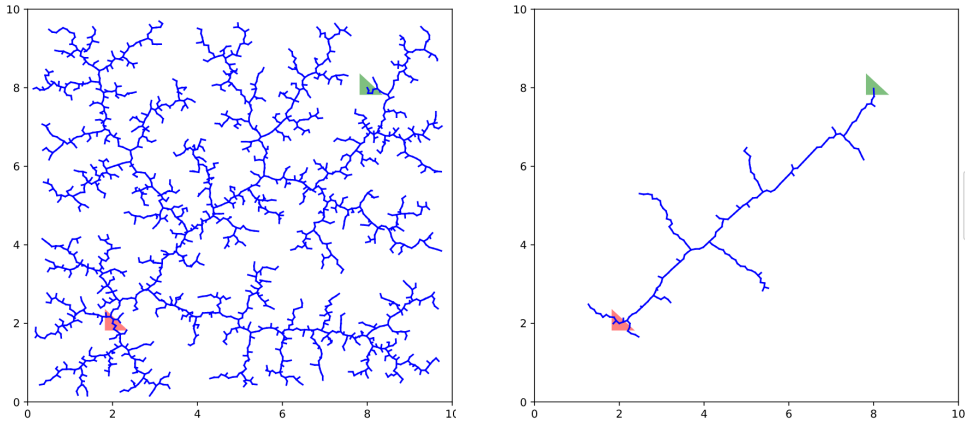

Figure 1.1: The difference between uniform sampling on the left and goal biased sampling, which guides focuses the expansion towards the goal and thus, considerably less samples are necessary to reach it. (Starting position is red, goal is green)

increase its effectiveness in environments with many obstacles. When it comes to path planning in such environments, we may often encounter situations, where certain obstacles repeat themselves or are at least vaguely similar. In such cases, it would be more beneficial to use out knowledge from encountering the same or similar obstacles to aid us in avoiding them. The solution we propose is to create a database of simple "path primitives", which can be inserted into the environment based on the encountered obstacles, thus providing an efficient way to avoid such obstacles.

In the experiments, we provide various methods for creating the said database and then use use them in an attempt to increase the efficiency of RRT. We then compare this newly created algorithm with state of the art implementations of RRT, EST and PRM provided in the OMPL library [13].

Chapter 2

Sampling based planning

Sampling based planning algorithms have undergone considerable development in the recent years [18]. However, they are by no means a complete novelty in the study of motion planning, with the first mentions going as far as late 1980s [19].

Their time efficiency in solving complex problems and implementation simplicity makes them a widely sought out method. They are often utilized in areas such as motion planning for robotic manipulators (see, for example, [20]) or the study of protein folding [22]. Both of these problems suffer from having many DOF and thus a highly dimensional configuration space, making them ideal candidates for SBP.

As with most other planning algorithms the main purpose of SBP is to find a path for the robot (in planning often interchangeable with the word agent) through the given configuration space $\mathcal{C}$ which is a set of all the possible robot's configurations. Configuration is a set of variables which completely describe the robot's state in the physical space. We can further divide the configuration space into the space of free configurations $\mathcal{C}_{free}$ and the space of occupied (or collision) configurations $\mathcal{C}_{occ}$, such that

$$\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{occ}.$$

Hence we can define the desired path as a series of configurations from $\mathcal{C}_{free}$ with the start and goal configuration being the first and last respectively, which can be traversed in order from start to goal by interpolating between individual configurations while always remaining in the space of free configurations.

One of the necessary components of SBP algorithms is thus some form of a collision engine, usually provided by the environment, which given a configuration can determine whether it belongs to $\mathcal{C}_{free}$ or $\mathcal{C}_{occ}$.

The other indispensable component of SBP algorithms is a of metric determining the distance between two given configurations. This is usually determined by the topology of the configuration space of the environment. For example, with the most common topology being $\mathbb{R}^n$ a basic metric such as the Euclidean metric suffices.

2.1 Basic SBP algorithms

Probably the most commonly utilized SPB algorithms are RRT [1] and PRM [4] algorithms. Both of these usually rely on uniform sampling and are the basis for many other variations of algorithms. Among the simpler ones we can mention, for example, RRT* [3] and PRM* [3], which aim to find the optimal solution for a given problem (although it is again not guaranteed to find such solution in finite time). Another common example is RRT-Connect [8], which performs a bidirectional search by constructing a tree graph from both from start and goal positions in the environment and attempts to connect them. Common practice in SBP is also the introduction of a goal bias. Some percentage of samples is either replaced by the goal configuration itself or by sampling from a gaussian distribution in a small area around the goal.

The last mentioned common SBP algorithm, the EST [5], differs from the former two as it does not utilize uniform sampling. Unlike RRT and PRM, the EST algorithm alters the sampling based on already existing nodes of the created graph.

2.1.1 RRT

RRT, introduced by Stephen M. LaValle [1], is undoubtedly the most simplest of the SBP algorithms with high utilization in both practice and for educational purposes. It is also the center point of this entire thesis.

In its simplest form, RRT attempts to create a tree graph of nodes in the $\mathcal{C}_{free}$ by gradually expanding it from the start configuration while simultaneously checking for collisions. An example of how such a tree might look in a 2D environment is shown in figure 2.1.

45 iterations 2345 iterations

Figure 2.1: An example image of the graph produced by the RRT algorithm published by Stephen M. LaValle in the book “Planning algorithms”[2].

The entire RRT algorithm can be written in pseudo code in the following

way, where Q is the output graph, q_{start} are the start and goal configurations respectively, n is the number of iterations of the algorithm and Δq is the so called incremental distance:

```
rrt(qStart, n, deltaQ) -> Q:
    Q <- {}
    Q.init(qStart)
    for i from 1 to n:
        qRand <- randomSample()
        qNear <- nearestNode(Q, qRand)
        qNew <- moveTowards(qNear, qRand, deltaQ)
        Q.addNode(qNew, qNear)
    return Q
```

In the beginning of the algorithm, the graph Q is initialized using the start node (the only node of the tree representing the start configuration). The function `randomSample` returns a random configuration from the configuration space using the uniform distribution. The function `nearestNode` finds the nearest node in the already existing tree. Finally, function `moveTowards` attempts to interpolate from node q_{near} towards node q_{rand} until either a collision configuration is reached or a distance of Δq has been traversed from q_{near}. The new node q_{new} is then added to the graph with its parent node being q_{near}.

The algorithm can be further improved by introducing a goal configuration q_{goal} as an input variable and ending the expansion when a configuration in a close proximity of the goal is reached. Additionally, by including a goal bias and replacing some random samples by the goal, the goal configuration itself can be reached. The desired path can then be obtained by backtracking through the created tree from the goal configuration to the start configuration.

A simple example of goal biasing is shown in the following code, which modifies the `randomSample` function. p is the percentage of situations, where we choose the goal configuration instead of a random one. The function $randomFloat$ uniformly samples a number from the interval $[0, 1]$.

```
randomSample() -> qRand:
    if randomFloat < p:
        return qGoal
    else:
        return uniformRandomSample()
```

The `nearestNode` function is for obvious reasons a notable bottleneck of the RRT algorithm, as with its most basic implementation it has a linear time complexity dependent on the number of nodes in the graph Q. This problem can be avoided by organizing the points of the graph into a data structure called k-d tree [21]. This allows for the nearest nodes search to be done with logarithmic time complexity. Additionally, the time complexity of adding nodes to the graph (and hence incorporating them into the existing k-d tree) is also logarithmic.

An noticeable disadvantage of the RRT algorithm is its bad performance in confined spaces and narrow passages (see figure 2.2). In such cases many of the randomly sampled configurations lead the tree expansion into collisions, thus hindering it from further growth. This can naturally lead to the inability of the tree to expand into the vicinity of the goal configuration, making it impossible to find a solution with a limited number of nodes. The `randomSample` function is unexpectedly the main candidate for any possible improvements to avoid this issue, which is after all also the focus of this thesis.

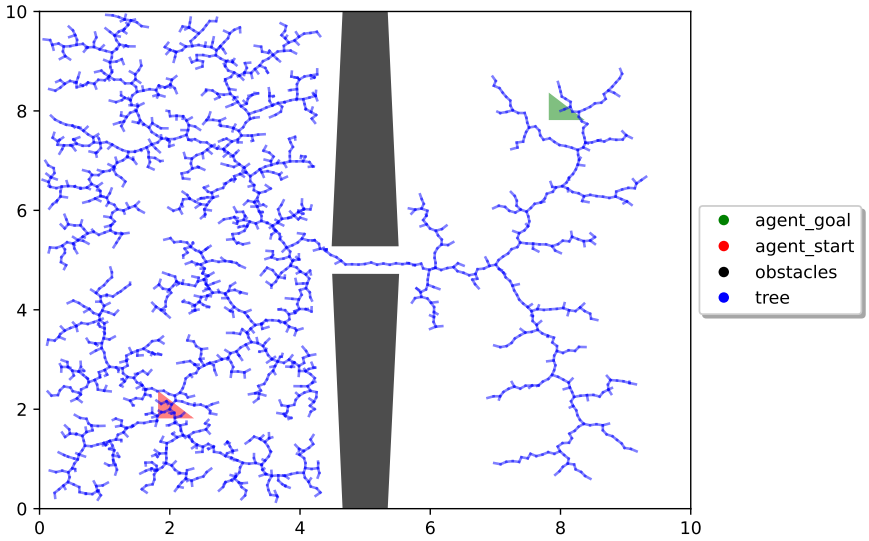

Figure 2.2: An example of an environment with a narrow passage. The tree expansion is stuck for a long time on the left side of the obstacles.

2.1.2 RRT*

RRT* [3], originaly published in [3], is an improved version of the RRT algorithm which guarantees finding the optimal path (if one such exists) as the number of generated nodes reach infinity. Despite that being realistically unfeasible, the RRT* still provides a close to optimal solution even with a finite number of nodes while keeping the benefits of simplicity and speed of basic RRT.

The RRT* algorithm differs from the usual RRT in two points. The one is the process of adding a new node to the generated tree. Each node keeps track of a cost value, which represents the length of the path traversed along the graph from the start to the aforementioned node. The process of sampling, finding closest node in the graph and moving towards the sampled node remain unchanged. However, instead of immediately taking the nearest node q_{near} as its parent, the newly created node instead searches its vicinity (usually in a fixed radius) for possible other neighbouring nodes. If any of

them have a lower cost value than q_{near}, the newly created node will favour the cheaper one as its parent.

The second difference is the rewiring of the tree which occurs after a new node has been connected. The neighbours of the newly added node are once again examined and they get reconnected to the new node, if it means their costs would reduce.

It is self explanatory, that all connections between nodes must be collision-free. Hence, if a collision would occur by connecting two nodes with an edge during either the tree expansion or rewiring parts, such an edge must be discarded.

An example of the generated tree in a 2D environment is shown in figure 2.3.

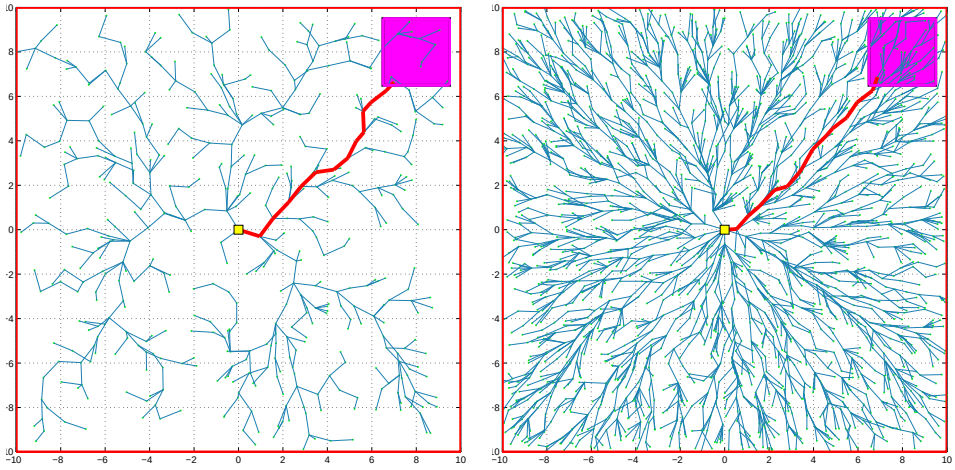

Figure 2.3: An example image of the growth of the graph produced by the RRT* algorithm published in [3]. Purple region is goal.

2.1.3 PRM

Unlike RRT the PRM does not create a tree graph but a normal connected graph (possibly with cycles) covering the entire environment. It is therefore not simply possible to obtain the finished path immediately once the graph is constructed by backtracking from the goal node to the start and a separate graph search algorithm must be used. The entire path planning process can therefore be split into two phases — the construction phase and query phase.

The construction phase is initiated by adding the start and goal configurations into the graph. Then, configurations are uniformly sampled from the configuration space checked for collisions with only collision-less configurations being accepted (hence, it would be better to sample only from C_{free} however, this approach is in most cases not applicable due to its complexity). Each created sample is then connected with an edge to k nearest neighbours or possibly to all nodes within a certain range. The connecting of the neighbours must be accompanied by collision checking. The most common approach is

interpolation between the two nodes which are to be connected while checking any intermediate configurations for collisions. If a collision is detected, the connecting process is aborted and the possible edge between the two nodes is discarded. This process of sampling and connecting is then repeated until a desired density (i.e. desired number of nodes) of the graph is reached.

The query phase is then carried out by an external graph search algorithm. The most applicable being the Dijkstra algorithm (with weights of each edge being equal to the distance of the two nodes they connect) or possibly the A* algorithm.

The PRM algorithm again suffers greatly in confined spaces, which force it to discard many of the sampled configurations due to collisions. Narrow passages on the other hand cause the graph to be disconnected, making it impossible to find a path from one disconnected subgraph to another. Altering the sampling process is again the most apparent way to tackle this issue.

2.1.4 EST

Similarly to RRT, the output of EST is a tree graph. As mentioned before, this algorithm does not create its samples uniformly. Instead, a weight w is assigned to each node in the graph inversely proportional to the number of its neighbours (usually the number of neighbouring nodes in a fixed range around each node):

$$w = \frac{1}{\#neighbours}.$$

Instead of the usual sampling, a node is selected from the existing tree using the weights for a randomized weighted selection. As such, the nodes with the highest weights (and thus least neighbours) get selected most often. The tree is then expanded from the chosen node. The expansion itself can be executed, for example, by uniformly sampling in a fixed range around the node and then connecting the node with the sample (while again verifying, that the created edge is collision-less).

An often used alternative to EST is the Guided EST algorithm [6]. From the original it differs only in the weight function. The Guided version introduces for each of the nodes *order*, how recently the node was created, *out_degree*, number of outgoing edges, and A^*cost, the estimated distance to the goal. The weight equation then looks as follows:

$$w = \frac{(order)^{\alpha}}{(\#neighbours)^{\beta}(out\_degree)^{\gamma}(A^*cost)^{\delta}}.$$

It is naturally possible to alter the behaviour of the algorithm by modifying the parameters α, β, γ, δ.

2.2 Related works

As mentioned before, sampling algorithms are far from being considered a finished area of research. One of the main focuses of such research is without

a doubt the process of sampling utilized in each of the different algorithms. Naturally, with the recently growing popularity of machine learning, not even SBP has been spared from its influence [9].

The aforementioned sampling is one of the places, where various machine learning algorithms can be incorporated. There are, however, other areas closely related to SBP where machine learning can be indispensable. One such area concerns the environments which are used for planning. From the basic examples of SBP algorithms in the previous section it is apparent, that these algorithms are most suited for static environments, that is, environments, where obstacles are fixed and only the agent moves around. That is not to say they cannot be used for environments with multiple agents or movable obstacles. Even for these is SBP applicable by utilizing some simple workarounds, such as having the configuration of all agents incorporated into the configuration space or introducing a variable of time into the configurations and creating an appropriate collision checking system.

However, such approach becomes less applicable for more realistic environments with complex laws and seemingly nondeterministic behaviours. In such cases, the go to solution is to rely on some form of machine learning. Furthermore, machine learning can also aid in alleviating some burden from complicated collision checking algorithms or even improve path quality when searching for optimum solutions.

2.2.1 RRT-Path

RRT-Path [7] is one of the simple algorithms aiming to improve the basic RRT algorithm by altering the inherent sampling mechanism. It also served as an inspiration for this thesis and is used in a simplified form when using database sampling primitives in the actual algorithm.

RRT-Path proposes an efficient method of guiding the tree expansion through the environment by introducing temporary goals. Instead of sampling the configuration space purely uniformly, a specific temporary goal is chosen instead of a set percent of samples (this is the same principle as goal biasing mentioned in the explanation of RRT). Once this temporary goal is reached, the next one in order is selected, thus efficiently leading the tree expansion along the temporary goals towards the main goal. This approach naturally works best when enough information about the environment is known and the temporary goals can be deployed into the most crutial positions.

2.2.2 MS-RRT

Another one of the algorithms which aim to augment the sampling method of RRT is the Multi-Sample RRT (MS-RRT) [23]. In this case, instead of sampling only once per RRT iteration, multiple samples are generated and the tree is then directed towards their mean.

Two different algorithms have been introduced based on this one. The first being MS-RRTa, which alters the nearest node selection process of RRT. For each of the created samples, their nearest node in the preexisting tree is

found. The tree is then expanded from the node, which was selected as being "nearest" the most with the direction of the expansion being the mean of all samples, which selected the node as its nearest.

The second alternative called MS-RRTb is very similar to MS-RRTa when it comes to handling the acquired samples. The only difference is, that the samples are not generated in every iteration of RRT and are instead uniformly generated at the beginning of the algorithm.

2.2.3 Planning with diffusion

One of the approaches which rely on machine learning to alter the sampling in planning algorithms is presented in [10]. In this case, the underlying idea is to use a diffusion model on a large amount of randomly generated samples and effectively converge them to the desired path. The information about obstacles is provided into the model in the form of a reward. This makes it also possible to guide the sampling distribution from undesired areas into more desirable ones. The authors show, that in the best case scenario the planning itself becomes mostly a question of ordering the shifted samples from start to goal.

2.3 Open Motion Planning Library

As a final part of this chapter it is necessary to introduce one of the largest open source libraries aimed at sampling based motion planning, the Open Motion Planning Library (OMPL) [13], which is also the center piece for the computer program of this thesis. This C++ library provides a wide assortment of state of the art sampling based planning algorithms, including several variations of RRT, PRM and EST. Its structure is meant to be used as a base building block, connectable with standalone collision and visualization engines. Finally, the library also provides several ways for creating new custom planners and samplers as well as means for testing and benchmarking experiments (those can be easilly visualized using the Planner Arena tool [17]).

Chapter 3

RRT with path primitives database

In this chapter, we will introduce our proposed method for augmenting the RRT algorithm as well as the suggested implementation of its various components. As mentioned before, the main idea of this thesis is to modify the process of sampling of the RRT algorithm by introducing simple path primitives (i.e. short paths created by the basic RRT algorithm) and sampling along those to avoid obstacles. One such example of a primitive can be seen in figure 3.1.

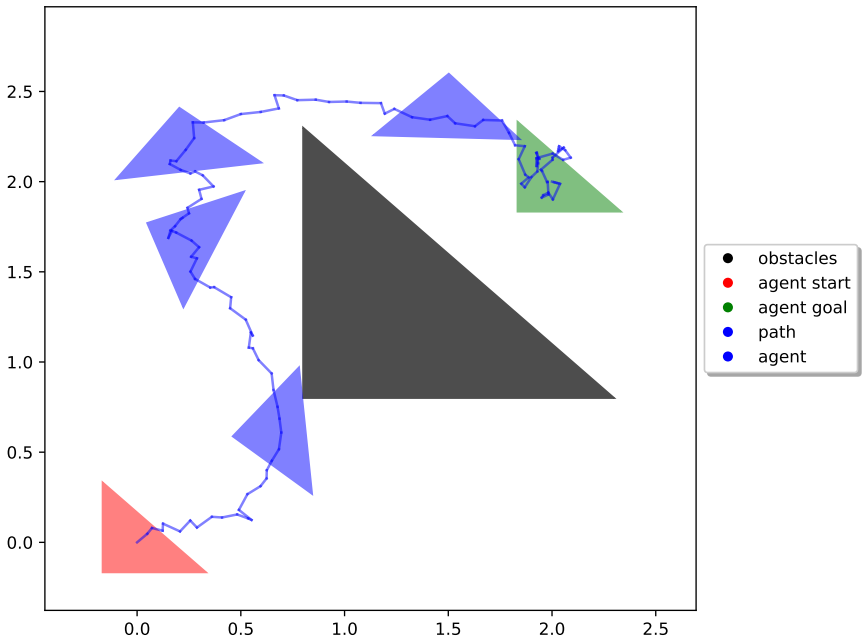

Figure 3.1: Basic path primitive for avoiding an obstacle. Start and goal configurations are shown in red and green respectively along with a few intermediary configurations in blue. The blue line is the primitive itself.

The said path primitives will be generated in advance for any specific

agent. The main aim is to provide a simple solution for avoiding burdensome obstacles such as narrow passages, which cause a decrease in performance for the RRT algorithm. As such, the primary focus will be on creating primitives for preset kinds of obstacles which we expect to appear in the environment. However, we will also discuss the applicability of this method for an environment with a more general variety of obstacles.

In the following section we will explain what kind of environments this thesis will cover and what approaches were used to implement them. Following that, we will focus on the only part of the RRT algorithm from part 2.1.1 which is to be modified — the `randomSample` function. We will discuss the possibilities for generating the database of path primitives as well as the ways for choosing the ideal primitive for a given situation and expanding the generated tree with it.

3.1 Environments

In this thesis we will be strictly limiting ourselves to simpler static environments with a single agent in 2D an 3D. Hence, the obstacles will remain unchanged with time. Furthermore, we introduce the possibility for the agent in 2D to have simple revolute joints, since simple 2D environments are usually do not possess much of a challenge for the RRT algorithm.

The configuration space will hence take on several different forms depending on the environment. In terms of topology, for 2D environments without rotation it is simply $\mathbb{R}^2$, similarly in 3D it without rotation it is $\mathbb{R}^3$. Naturally, these configuration variables will be kept in preset bounds, as it would not make sense for our environments to be endless. If we introduce rotations, we need to start working with the special Euclidean groups SE(2) or SE(3) respectively. These spaces can also be rewritten using the spherical topology $\mathbb{S}^n$ as

$$SE\left(2\right) = \mathbb{R}^2 \times \mathbb{S}^1,$$

$$SE\left(3\right) = \mathbb{R}^3 \times \mathbb{S}^3.$$

When it comes to the revolute joints, we simply need to add an rational additional configuration variable for each joint. Thus we expand our topology by additional $\mathbb{R}^n$, where n is the number of joints in the model. It might be logical to use a spherical topology for the revolute joints, as they could possibly rotate around indefinitely, however, this approach would often incite the agent to "fold in on itself" so that it could easily overcome confined areas of the environment. Thus it is more beneficial for our purposes to use a simple rational number bounded in some range of values. Further expanding on this idea, it might be more mathematically proof to express the topology as an interval $[k, m]$ where k and m are the appropriate bounds limiting the joint's rotation, however, for simplicity's sake, we will allow ourselves to keep using the notation of $\mathbb{R}$, under the assumption that appropriate bounds will be provided later.

The configuration space $\mathcal{C}$ of a 2D object with rotations and 3 revolute joints will therefore look like

$$\mathcal{C} = SE(2) \times \mathbb{R}^3 = \mathbb{R}^2 \times \mathbb{S}^1 \times \mathbb{R}^3.$$

Mathematically it would not pose much of a problem to combine the $\mathbb{R}^2$ and $\mathbb{R}^3$ into $\mathbb{R}^5$, however, due to the way the OMPL library calculates distances, this would be undesirable.

In the following subsections we shall briefly tackle the topic of the necessary metrics calculations. Afterwards, we will introduce our approach to collision checking and the methods used to represent the agent and obstacles.

3.1.1 Important metrics

Since our algorithm is built upon the OMPL library, there is no need for us to implement any of the metrics ourselves. Despite that, it is highly desirable to understand the underlying principles behind distance calculations, which are one of the core mechanics of RRT and other SBP algorithms.

The first and most basic metric which will be used for distance calculations in the $\mathbb{R}^n$ space is the Euclidean metric. For such spaces we can calculate the distance d_e between two points $\mathbf{x}$ and $\mathbf{y}$ as

$$d_e = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

When it comes to distance calculations in the $\mathbb{S}^1$ topology necessary for 2D rotation calculations, we must simply realize, that any variable $v \in \mathbb{S}^1$ can have only values in the interval $[0, 2\pi)$ (or possibly $[-\pi, \pi)$, depending on our preferences) and can "overflow" the border of these intervals. As such, the distance between two points is necessarily constraint into an interval of $[0, \pi]$. With this, we can calculate the distance between two points x and y by first calculating an intermediary value

$$d = |x - y|$$

with the resulting distance d_{s1} being calculated as

$$d_{s1} = \begin{cases} 2\pi - d, & \text{if } d > \pi \\ d, & \text{otherwise} \end{cases}$$

The final distance metric we need to introduce is for the $\mathbb{S}^3$ group. This is a moderately complicated problem which cannot be solved if we only rely on convenient methods of 3D rotation interpretations such as euler angles and the rotation matrices. For this purpose, we need the rotation to be expressed in the form of quaternions. Quaternions offer a highly efficient, though also quite user unfriendly, way to represent rotations in 3 dimensions. Appropriate conversions between quaternions, Euler angles or angle axis representations can be easily found in countless sources. Using this representation, it is

possible to compute the distance of two rotations in 3D space as the arclength between their respective quaternions.

The only remaining part of our distance computations is the case of compound spaces such as the previous example of a 2D agent with 3 joints. In the case of OMPL, this is done with a weighted sum of the distances from each component. As such, if the distance in the position of the agent is calculated as d_{e1}, the distance in its rotation as d_s and distance for its angles as d_{e2}, then the total distance will be calculated using preset weights w_1, w_2 and w_3 as

$$d = w_1 d_{e1} + w_2 d_s + w_3 d_{e2}.$$

The default in OMPL is to set the weights of euklid metrics to 1 (here w_1 and w_3) and the rotation metrics to 0.5 (here w_2).

3.1.2 Collision engine

For the purposes of this thesis we opted out for using the Robust and Accurate Polygon Interference Detection C++ library (known as RAPID) [14]. This library provides a simple and fast method for defining rigid bodies expressed as a triangle mesh and computing any collisions between two such defined bodies. The time complexity is logarithmic with respect to the number of triangles in the objects. It is even possible to determine the specific triangles of the mesh which are in collision, although in our case the basic information about whether two objects are in collision with each other or not will be sufficient.

All the required objects for obstacles and agents are stored using .obj files. These files proved to be the simplest solution as they provide an easy way to represent simple triangle meshes without any unnecessary redundant text. They can be easily made either by hand for smaller 2D objects or using external programs such as Blender [15] and loaded into the RAPID library or used for visualizing purposes.

With this combination we can already process most of the environments we have envisioned in the previous section. RAPID library is set up to handle 3D objects, however, any 2D environment can be expressed in 3D by simply keeping all triangles of every mash on a single plane in space and allowing rotations only along an axis perpendicular to such plane.

With this, the only problem we need to solve is the addition of revolute joints for an agent. This can be solved by a few preliminary calculation for determining the position of each linkage of the agent and then feeding them one by one into the RAPID library. We will discuss this issue only in 2D, as that is the most we require.

We can define the joints for each child linkage as having an offset p_{offset} from their parent. The child linkage will then take the joint as the center of its respective coordinate system. The rotation of each child is then determined by the configuration variable corresponding to the joint connecting the child with the parent (we shall call it α). As such we can find the transformation matrix from the coordinate system of the child (c) to the coordinate system

of the parent (p) as

$$\mathbf{T}_c^p = \begin{bmatrix} R(\alpha) & p_{offset} \\ \mathbf{o}^T & 1 \end{bmatrix},$$

where $R(\alpha)$ is a rotation matrix.

The transformation matrix from the coordinate system of the base linkage (b) to the coordinate system of the environment (e) can be directly obtained from the configuration space, as both the position and rotation is represented in the SE(2) part of it. If we call the position of the agent (and hence the base linkage) p_base and the angle of its rotation β, we can write the resulting transformation matrix as

$$\mathbf{T}_b^e = \begin{bmatrix} R(\beta) & p_{base} \\ \mathbf{o}^T & 1 \end{bmatrix}.$$

If we were to assume an agent with three linkages connected in a way, such that the base linkage has one child (c1) linkage which itself has one child (c2), we could compute the transformation matrix from the coordinate system of the second child to the environment as

$$\mathbf{T}_{c2}^e = \mathbf{T}_b^e \mathbf{T}_{c1}^b \mathbf{T}_{c2}^{c1}$$

As such, we can compute the transformation matrix of any linkage with respect to the environment. With this, we can produce a simple recursive function using an assumed linkage class (with access to information about the model of the linkage, joint offsets, its parent and children linkages and configuration variables), which computes all possible collisions between linkages and obstacles.

```
updateTransformAndCheckCollisions(linkage) -> FoundCollision:
    // compute transformation of this
    if linkage.has_parent:
        linkage.tf <- (linkage.parent.tf
                    * tfFromJoint(linkage.jointInfo))
    else:
        linkage.tf <- tfFromConfigVar()

    // check collisions
    if isInCollision(linkage.model, linkage.tf):
        return True

    // recursively check other linkages
    for child in linkage.children:
        if updateTransformAndCheckCollisions(child):
            return True

    // no collision for itself or children
    return False
```

The functions `tfFromJoint` and `tfFromConfigVar` represent the transformation matrix calculations from joint information and configuration variables respectively, which were introduced earlier. The `isInCollision` function is presumed to call upon the underlying RAPID library and provide it with the mesh model of the linkage and its transform with regards to the environment. The information about obstacles is presumed to be unchanging and is thus omitted.

With slight modifications we could also introduce a system to check for collisions between the linkages themselves, however, this will be unnecessary for our purposes, as we will be dealing with agents with only a small amount of linkages and limiting their possible movement range by introducing appropriate bounds for the configuration variables.

3.2 Database of path primitives

With all the preliminary requirements done, we can now focus on the main topic of this thesis. In this section we shall discuss the database of primitives, the methods which will be used to create them, as well as the process for their deployment into the environment.

As mentioned before, the idea behind these primitives is to aid us in overcoming select types of obstacles. They can also be used as a means to guide the expansion of the tree in a certain direction. The primitives we aim to create are mainly shaped in the form of one or multiple paths. For multiple path primitive see figure 3.2.

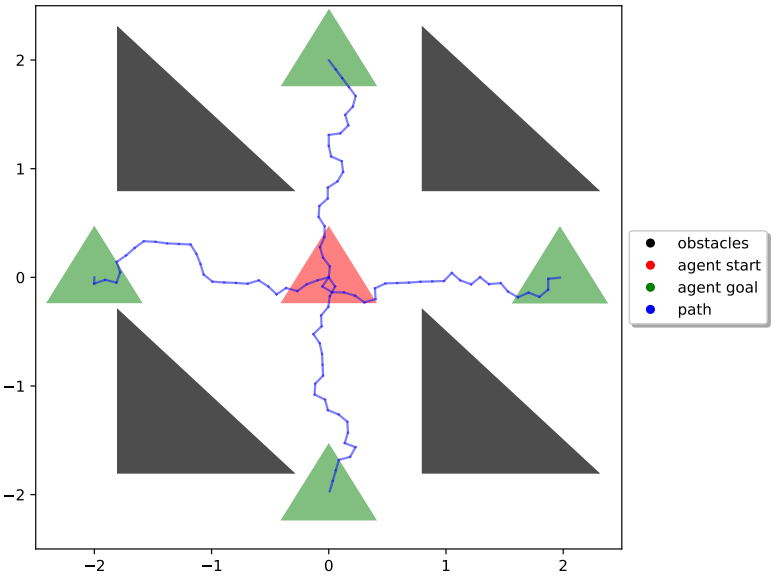

Figure 3.2: Basic path primitive with multiple paths for avoiding obstacles.

It is quite apparent, that these path primitives will usually have to be created for different agents separately. It is naturally completely impossible to reuse them for environments with different topologies. However, when it comes to the obstacles themselves, some of them may be interchangeable. As seen in figures 3.1 and 3.2, the path primitives will achieve their goal even when a similarly sized and shaped obstacles replaces the ones in the pictures.

3.3 Selecting optimal path primitives

Selecting the optimal path primitive for any specific situation is a major problem we will have to tackle. A wrong primitive might be deployed into the graph in such a way, that it overlaps with obstacles and stops the expansion of the tree due to collisions. The easiest solution would be, to have a few path primitives for every kind of obstacle we expect to encounter. One issue with this method is, that even when we know the kinds of obstacles we might encounter, we still don't know from which direction we might be approaching them and what shape and rotation our agent (possibly with joints) has. Furthermore, we would require to have quite extensive knowledge of the environment and its obstacles, which is often very unrealistic.

However, for our purposes, it is not necessary to know exactly what kinds of obstacles we are dealing with. Instead, our only concern is their approximate shape. Furthermore, we only need to take note of a few obstacles in the vicinity of the space, where we want to deploy our path primitive.

There are countless methods how we could determine whether some obstacles are similar. Already implemented algorithms for directly matching shapes can be found in libraries such as OpenCV [16]. Another option might be to scan the surrounding obstacles in a "lidar-like" fashion in order to create a point cloud and match it with existing ones in the database. Most of these approaches, however, require a considerable amount of computational time. The way we chose to tackle this issue is to use an occupancy grid.

3.3.1 Occupancy grid

An occupancy grid is a set of points which help us determine occupied areas of the environment. As shown in figure 3.3 we select a few test points in a fixed predetermined pattern in the vicinity of the area where we wish to place our primitive. At each of these points we determine, whether a collision occurs and store this information into a binary array. For this collision checking we may use our existing agent. However, with more complex agents, this might result in a messy occupancy grid. It is therefore better, to substitute the agent for a simpler object of adequate size (compared to the obstacles — too small object might miss smaller obstacles, too big might on the other hand miss gaps between obstacles). The comparison between using an complex and simple agent is visible in figure 3.3.

Collision checking and creating binary arrays would be repeated even when generating said primitives. This way, it is possible to classify the

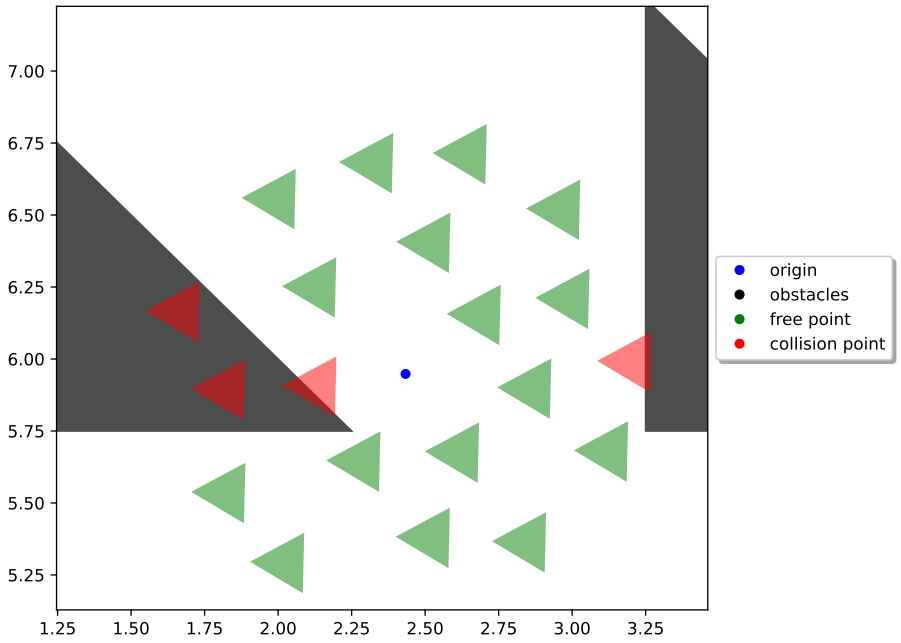

Figure 3.3: A simple example of an occupancy grid with a triangle agent serving for testing each position in the vicinity of origin.

shape of obstacles each primitive is made for using but a few binary numbers. For example, the occupancy shown in figure 3.3, might get saved as 11000100000000000001 — with four ones in specific indexes denoting the collision occurring at the respective four triangles.

When we wish to use our database in a specific place, we simply need to compute the occupancy array in said place and search the database for the most similar an array. To determine the most similar array it is possible to either test each entry of the database against the newly created one by counting dissimilar elements (i.e. calculate their Hamming distance). For larger databases this approach might create some unnecessary time delay. To solve this, we can also load each of these binary arrays from the database to a k-d tree (which we are already necessarily using for our RRT algorithm) and then use said k-d tree to find the closest entry to the newly created array.

When it comes t said occupancy grids. The first and most natural one would be to generate its test points in an actually orderly grid. This approach, however, begins to be lacking, when we start introducing environments with highly dimensional configuration spaces. In such cases, the amount of points necessary to cover an area with a grid increases exponentially with dimensions.

The better approach is, therefore, to generate the test points randomly, exactly as in figure 3.3. Since we wish to have these points concentrate around a certain area, the best method for generating them is using Gaussian

distribution. It is also favourable to add conditions such as minimum distance between test points, and maximum distance from the center location. Furthermore, since we are merely generating a basic pattern of points which will then be moved to the locations of interest it is desirable to generate these points around the origin (that is, the point with coordinates [0, 0, 0, ...]).

3.3.2 Applying occupancy grid to proposed environments and structuring database

The approach of spreading the occupancy grid into all dimensions of the configuration space is one we might wish to take for a general environment, ideally with a simple linear topology. However, if we wish to tackle the environments we have introduced previously, which have rotations and possibly joints, we will soon run into various issues.

The main problem is, that spreading out the occupancy grid in the rotation and revolute joint parts of our topology does not bring us many benefits. Especially if we were to replace our agent with a simpler one for the collision detection purposes. Since we are primarily interested in the shape of the obstacles, we will obtain the best results by covering only the spatial component of the topology with test points. This will also decrease the amount of primitives we need to create, as we will no longer need put the original rotation of the agent into consideration when choosing the optimal primitive.

Since we now have no information from the rotational and joint parts of the topology, we need to either find a way to avoid needing them when choosing from the database, or incorporate their information directly into our database.

When it comes to rotations, there is a simple way to make them irrelevant. The only thing we need to do is to perform the occupancy grid calculations and path primitive deployment in reference to the rotated coordinate system of the agent. Hence, if we wish to check the occupancy at some specific point in the environment where our agent currently is, we must transform every point of the occupancy grid (which now has only parts related to the spatial component of our topology; we shall call such a point $\mathbf{v}$) using the transformation matrix of the agent.

The process of obtaining said transformation matrix was explained in section 3.1.2. The resulting moved points shall be called $\mathbf{p}$. Since we are working with transformation matrices, it is important to convert the points into holonomic vectors (by appending a 1 to them); we will further denote these with a lower index h. As such, the whole transformation of each occupancy grid point can be written as

$$\mathbf{p}_h = \mathbf{T}_b^e \mathbf{v}_h.$$

A similar process will be necessary for the deployment of the path primitives, which shall be discussed later.

The second part of topology we have omitted from the occupancy grid are the configuration variables related to joints. If we wish to choose the

appropriate primitive, we need to take into consideration the configuration the agent begins in. If an agent with several joints has its linkages folded, we cannot try forcing it to go along a path primitive which begins with an unfolded configuration.

From this, we can see that for agents with joints it is necessary to create path primitives beginning with various initial joint states. When picking the optimal primitive, we first need to choose a group of primitives whose initial joint state is the most similar to the agents current situation. This can be again done by comparing each available initial joint state in the database with the state of the agent's joints and choosing the one which is the closest in terms of Euclidean distance. Again, we can also input all the possible initial joint states (without the spatial and rotational parts) into a k-d tree to increase the speed of the search. Afterwards we can proceed to choose from selected group of primitives in the already introduced manner using an occupancy grid.

Overall, the database can be structured into a set of directories, with each directory containing primitives with the same initial joint conditions. The primitives can then be saved as individual files into these directories. The occupancy obtained during the generation of each primitive can then be set as the name of said files (either keeping the occupancy in binary or converting it into hexadecimal notation for shortening purposes).

With all this done, we can now write a simplified algorithm for choosing the optimal path primitive from the database, given the complete initial configuration of the agent and the pattern of points for the occupancy grid.

```
chooseAppropriatePrimitive(agentConf, occGrid) -> bestPrimitive:
    // transform the occupancy grid's points
    // (using transformation matrix as mentioned before)
    transformedOccGrid <- transformOccGrid(agentConf, occGrid)

    // get the binary array representing occupancy
    occupancy <- checkOccupancy(transformedOccGrid)

    // find the directory with the closest initial state
    // in the ones available in the database
    bestDir <- getClosestInitJointState(agentConf)

    // finaly get the primitive with the most similar
    // occupancy from those in the selected directory
    bestPrimitive <- closestOccupancyPrimitive(occupancy, bestDir)

    return bestPrimitive
```

3.4 Deploying path primitives

We have already discussed the classification of path primitives in the database. With that we can focus on the actual method of using them in the environment as well as the possible approaches for deciding whether we actually need to use them in a given situation.

The process we have been hinting at is to pause the usual RRT tree expansion at some suitable moment, take the position of the agent at the point where we wish to expand our tree and deploy a path primitive into that are, which the tree is to follow.

Overall, we are attempting to modify the `randomSample` function mentioned in section 2.1.1 with a modified algorithm for using primitives. The algorithm would look as follows:

```
// external variable for determining what we sample
usingPrimitive <- False
primitive <- None
conf <- None

randomSample(...) -> sample:
    if conditionsToUsePrimitive(...) and not usingPrimitive:
        usingPrimitive <- True
        conf <- ...
        primitive <- chooseAppropriatePrimitive(conf, occGrid)

    if usingPrimitive:
        sample <- samplePrimitive(conf, primitive)
        if primitive.usedUp():
            usingPrimitive <- False
    else:
        sample <- uniformRandomSample()

    return sample
```

The function `conditionsToUsePrimitive` is meant to determine, whether the situation the current tree is in requires the usage of primitives. As such, we leave the parameters of the related functions undefined, since it would require information, that is usually not provided to the `randomSample` function. Similarly, the exact configuration (variable `conf`) from which we intend to expand the tree might need some outside variables to be determined. Hence we leave its specifications blank for now and shall discuss them later.

In this section, we will ponder the idea of the deployment itself and the problems related to it as well as the issue of finding the suitable moments and parts of the tree, which should be expanded using the path primitives.

3.4.1 Choosing ideal situation

As mentioned before, the first thing we need to do is determine some conditions for switching between usual uniform sampling to sampling primitives. We shall hence tackle the `conditionsToUsePrimitive` function mentioned before as well as the process of determining the configuration, where we wish to use the primitive.

The first and most basic approach is to use the primitives randomly or in specific intervals. That is, we can set a fixed (or slightly random) number of nodes which shall be generated using the uniform distribution and afterwards switch to path primitive sampling. The configuration from which we shall expand can be, for example, the last generated node in the tree. Since in the early stages of the expansion most of the nodes appear on the edge of the tree, we can afford to use this approach to aid the tree in its growth.

Another trivial method would be to start using a primitive when we fail to expand the tree, i.e. when we detect a collision in the `moveTowards` function of RRT (see 2.1.1). The configuration to expand from would then be the last node, from which we failed to grow the tree. There are, however, several issues with this approach. The first one being, that the primitive will start from a place, where the agent is almost in collision. Since the primitive is chosen only using vague information about the obstacles, it can lead to the agent being unable to move from its already tight spot even with the aid of the primitive. The second issue concerns the environments we shall be using. Since we are mostly interested in environments with large amounts of obstacles, the situation where collisions happen would occur so often, that this approach might not differ much from using the primitives randomly.

An easy improvement that we can make for our methods is to store the configurations where we already used a primitive. Since using path primitives in such areas will not give us much benefit, as they might yield the same or similar results as the ones already used and would only increase the density of the tree in such places. We can therefore discard any attempts at deploying the primitives in places too close to these already stored configurationss. This can be again efficiently done using k-d trees.

3.4.2 Sampling primitives

Once we have decided to use a primitive, have selected a configuration where we wish to deploy it, and have chosen the optimal primitive, we can move on to sampling said primitive. However, before that, we still need to move the primitive to our desired location. Since we have previously decided to tackle everything with regards to the agent's coordinate system we must first transform the primitive using the transformation matrix of the agent. This is the same process we described when dealing with the occupancy grid. This time, however, the individual points of the primitive contain not only spatial, but also rotational and possibly revolute joint parts of the configuration space. These must be handled separately.

For the spatial parts, an identical approach as with the occupancy can be

taken. Hence, if we represent the transformation matrix of the agent (with regards to the environment) using its position $\mathbf{p}$ and rotation matrix $\mathbf{R}$ as

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{o}^T & 1 \end{bmatrix},$$

we can then transform the spatial part of each point $\mathbf{v}$ of the path primitive as

$$\mathbf{v\prime}_h = \mathbf{T}\mathbf{v}_h$$

(again using holonomic coordinates, variables with apostrophe are the ones which shall be used for the deployed primitive)

To transform the rotational part of the primitive we first need to represent it in the form of a rotation matrix $\mathbf{R}_{prim}$. With that, we can rotate it using the rotation matrix of the agent:

$$\mathbf{R\prime}_{prim} = \mathbf{R}\mathbf{R}_{prim}.$$

The revolute joint parts of the topology can be left unchanged. If the path primitive guides the agent to fold its linkages, then this should not change when we simply transform the primitive. The agent should still be guided to fold itself.

With the path primitive in place, we can finally start sampling the points from it. This process was heavily inspired by the RRT-Path algorithm mentioned section 2.2.1. We can, however, still make several variations to this approach.

The most simple way to implement the sampling is to return the points of the path primitive one by one until we reach the end of the primitive. This is, however, highly susceptible to unexpected variations in obstacles. If a path primitive which leads us into a collision was chosen (for example, because the obstacles were not grasped completely with the occupancy grid or because there was no perfect match in the database), then the expansion could be quickly stopped and a large part of the primitive's samples would be wasted.

To avoid this, we can adopt a similar approach to RRT-Path. As mentioned before, RRT-Path combines sampling of intermediate goals and uniform sampling. As such, we can treat the points of the path primitives as intermediate goals (we do not even have to use all of them; we can, for example, use every n-th point). Since uniform sampling would expand the RRT tree even in areas unrelated to the current primitive, we can instead use a Gaussian distribution centered either on the primitive itself, or one of the intermediate goals. We also don't necessarily need to wait until we reach the intermediate goals and can limit ourselves to taking a fixed amount of samples from each, before we move on to the next one. The rest is only a question of determining the percentage of samples taken from the intermediate goals themselves and from the distribution around them.

3.5 Approaches to generating path primitives

The last thing we need to discuss is the process of creating the path primitives. For this, we can use the usual RRT algorithm to find single or multiple short paths avoiding obstacles. This process will differ noticeably depending on how big we wish to create our database, what kinds of obstacles we expect to encounter and possibly what other effects we wish for the path primitives to have on the expansion of the tree.

The simplest approach is to manually introduce the agent to the most problematic situations we expect to find and create a small amount of path primitives to handle them. We can also use this method to create path primitives aimed at guiding the expansion of the tree in a certain direction. For example, if we are working with environments, where we expect to reach the goal by increasing a specific configuration variable (such as moving along one of the spatial axes of the environment), we can create primitives which attempt to avoid the given obstacles while expanding in the desired direction. Overall we can, for example, manually set the start and goal of a normal RRT algorithm to be at the opposite sides of a simple obstacle, run the RRT algorithm and save the outputted path. We can also set up multiple goals for each start and save all the outputted paths into one primitive, with the idea, that it would try overcoming encountered obstacles using multiple different routes.

It is important to notice, that when creating such primitives, our agent must start from the origin of the coordinate system without any rotation, since the primitives will be later transformed into the position of the agent during deployment. If we wish to start from an arbitrary position in the environment, we will need to save the primitive with respect to the coordinate system of the agent. Hence, we must perform an inverse operation to the one during deployment. The spatial parts of the path primitive's points $\mathbf{v}$ must be transformed with the inverse of the agents transformation matrix T (variables with an apostrophe are the ones, which shall be saved into the database):

$$\mathbf{v}\prime_h = \mathbf{T}^{-1}\mathbf{v}_h.$$

The rotation parts of the points must be again represented with a rotation matrix as $\mathbf{R}_{prim}$ and rotated by the inverse of the agents rotation matrix $\mathbf{R}$:

$$\mathbf{R}\prime_{prim} = \mathbf{R}^{-1}\mathbf{R}_{prim}.$$

The joint parts of the configuration space can once again remain unchanged.

With this, we can also afford to generate our database automatically inside a given environment. We simply need to randomly select an unoccupied configuration in the environment and note down its occupancy using the occupancy grid. This selected configuration will be treated as the starting configuration for the simple RRT algorithm. The goal configuration can be set, for example, by moving slightly in a predetermined direction from the start or by randomly selecting another unoccupied configuration in the

vicinity of the start. Afterwards, the RRT algorithm can be used to connect these configurations and the resulting path (the soon to be path primitive) will be saved under the computed occupancy as its identifier. This approach is especially useful, if we expect all environments to be structured in the same, or at least similar manner, since a single database will be usable for all of them.

The last and most general approach is to randomly generate obstacles of different sizes and shapes and attempt to find a path leading through them which will, for example, result in the agent moving a certain distance from the start. This approach naturally works best for a very large database and a fine occupancy grid, as it attempts to cover most situations the occupancy grid can differentiate (naturally, the distinguishable situations double with each additional point in the occupancy grid). As such, to make this approach beneficial for us, we would need to create a considerably large database, which would in turn slow down the search for optimal path primitives and thus slow down the whole path planning. Hence, we will not focus on this approach further.

Chapter 4

Experimental results

In this chapter, we shall focus on several different environments and compare our proposed methods for enhancing RRT with a state of the art implementation of SBP algorithms provided by the OMPL library (see 2.3).

4.1 Simple 2D environments

To begin the experiments, we have opted for a simple grid environment with triangle agent in 2D space without rotations (see figure 4.2). The premise is to manually create only very simple path primitives akin to straight lines (as shown in figure 4.1). Hence, we can use them to aid the tree in advancing through straight corridors in the environment.

The primitives were all created in a scaled down version of the environment visible in figure 4.1 by shifting the surrounding obstacles around the start position while testing for occupancy using an occupancy grid with 20 test points. Two variations of the database-based RRT were tested. One with 100 samples between each usage of path primitives, one with 200 (figure 4.2 was created using the first one). Goal bias was set to 5% (and it shall stay unchanged for all the other experiments as well).

In figure 4.2 we can see the tree created by our algorithm. There are noticeable instances, where the red primitives clearly aided the expansion such as in the middle of the picture. On the other hand, there are also cases, where a primitive was created close to an already grown part of the tree and was engulfed into the tree. Such case happened in the bottom right of the image, where a part of the primitive got mostly wasted and only increased the density of the existing tree.

In figures 4.3 and 4.4 we can take at how our algorithm compares to standard RRT, RRTConnect and PRM. Unfortunately, the our approach is noticeably slower than the other two. This shows, how extremely efficient the original algorithms are, especially on such easy environments. We can also deduce, that the usage of primitives is, in this case, clearly detrimental, due to the necessary computations needed and due to the time wasted, when a primitive is placed inefficiently and leads the expansion into collisions.

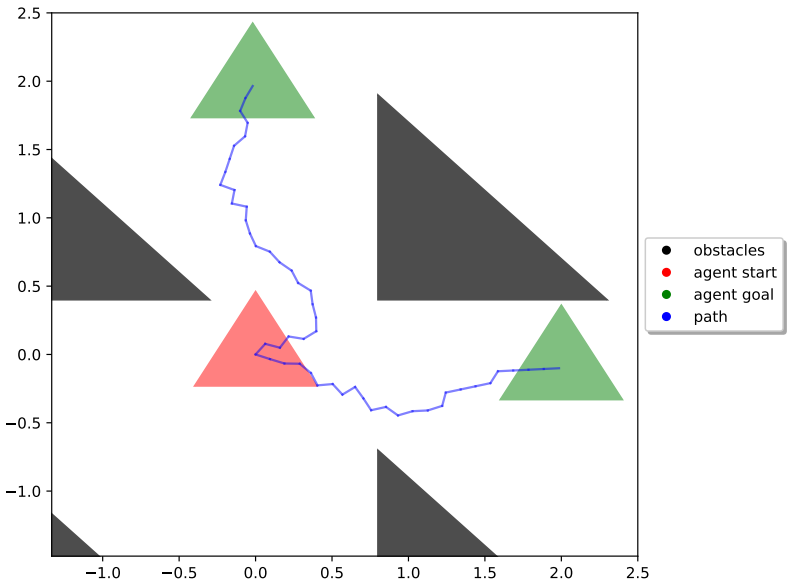

Figure 4.1: A simple primitive consisting of two paths guiding the expansion in the positive direction of both axis

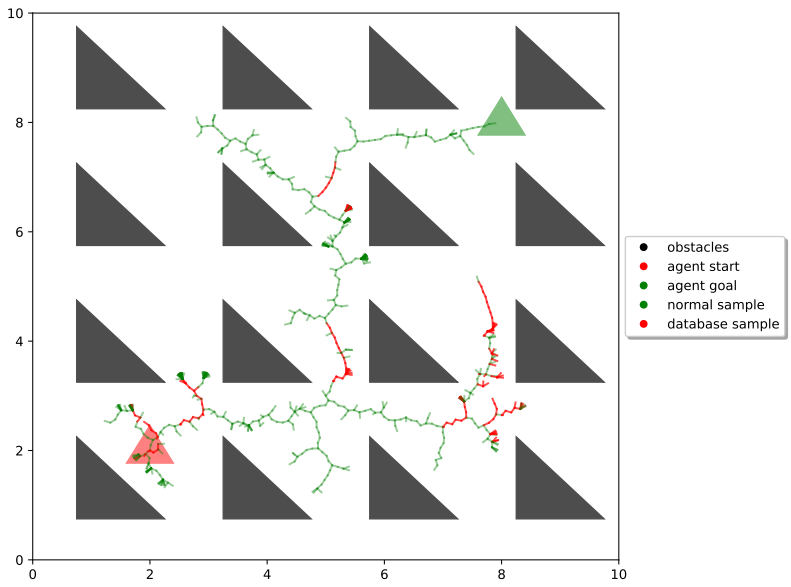

Figure 4.2: A simple environment. 2D topology without rotations.

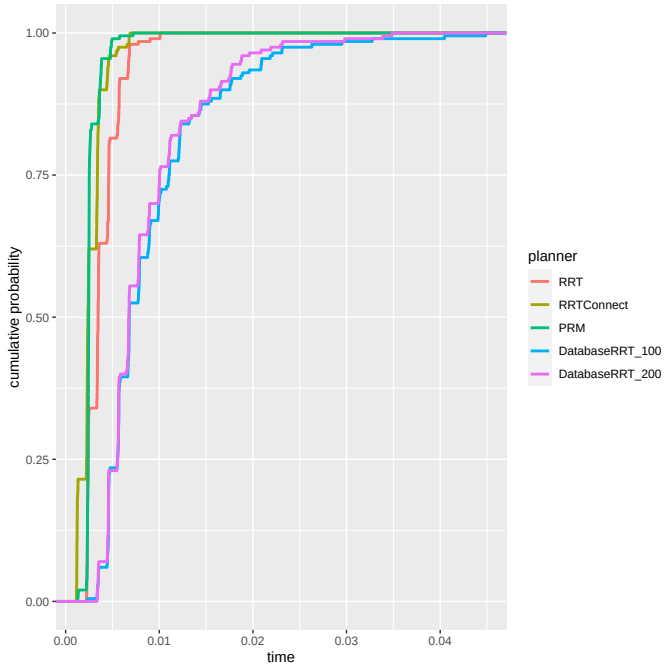

Figure 4.3: The cumulative distribution function denoting the percentage of finished runs of the algorithms for a given time. Normal RRT, RRTConnect and PRM are compared against our database-based RRT with 100 or 200 samples between each deployment of a primitive. It was created using OMPL's built-in benchmarking tools and the Planner Arena.

The next environment we will tackle is a 2D environment with rotations, denser obstacles, and an L-shaped agent. Due to the design of the environment, the agent is forced to use its rotation to get through. The environment is shown in figure 4.5.

In this case, a database consisting of 40 different primitives was created by randomly selecting start and goal configurations in the environment and connecting them with a path, as discussed in section 3.5.

As we can see in figure 4.5, the tree became visually significantly denser. This is caused by the additional dimension (for rotations), which is projected onto the 2D canvas of the image.

Using this specific environment we shall try to get some insights into what effects have different aspects of our algorithm, such as the method of following the primitive or the frequency of using said primitives, on its efficiency. Afterwards, we will compare the best variation of our program with standard RRT, RRTConnect and PRM.

The first variation of our algorithm, which we shall call "basic", will follow the path primitives by sampling them one point after another. The next variation ("skip") will use only every tenth configuration of each path primitive. After using said configuration ten times while altering it slightly (by adding a random number from the interval $[-0.1, 0.1]$ to each of its coordinates) it will move on to the next one configuration. The last variation ("gap") will

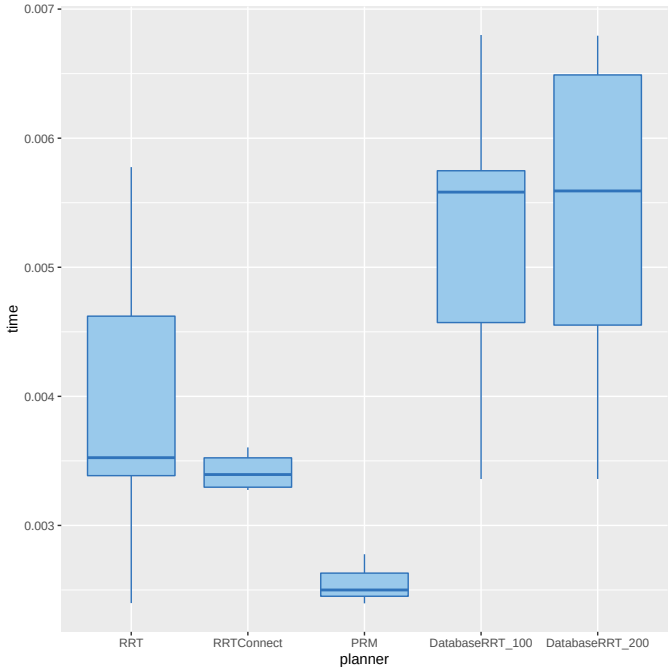

Figure 4.4: Box plot graph with hidden outliers describing the same data as figure 4.3. (Time is in seconds)

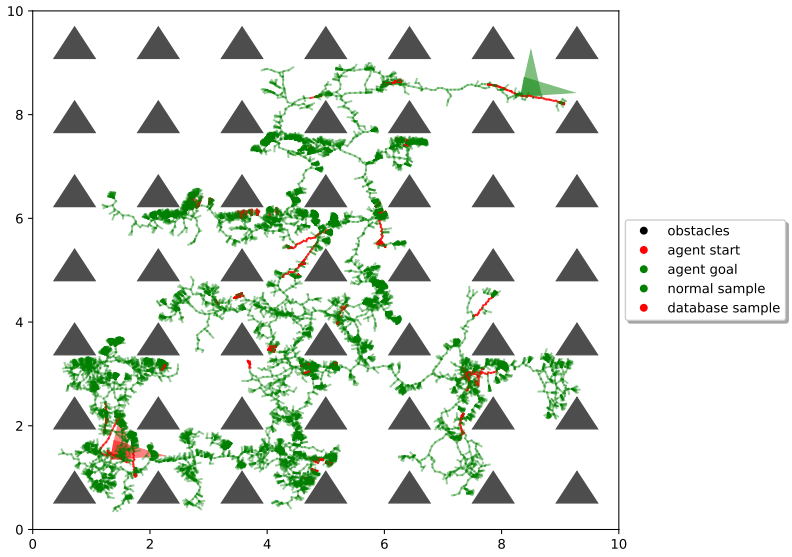

Figure 4.5: A 2D environment with rotations, denser obstacles, and more complex agent.

ensure, that no primitive will be used near a place, where another one was already deployed (the distance is calculate purely from the spatial part of the

configuration space and the minimum distance between primitives is 0.5).

There are 500 samples between each usage of path primitives. When "gap" algorithm determines the chosen configuration as too near to an already deployed primitive, it has to wait another 500 samples (that is, to avoid unnecessary search for empty space, especially in a situation, where there might be none).

The performance of these three variations as well as the performance of RRT, RRTConnect and PRM is shown in figures 4.6 and 4.7.

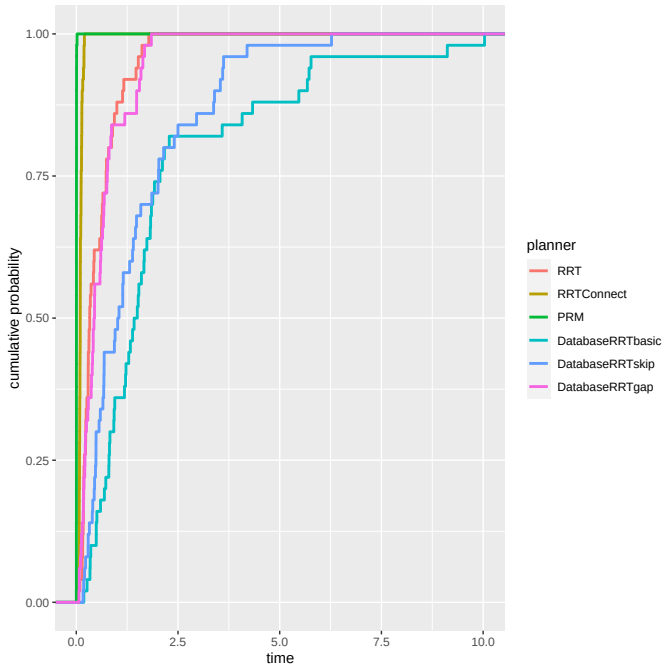

Figure 4.6: Cumulative distribution function describing the performance of the tested algorithms in the more advanced 2D environment for three variations of the database-based RRT as well as normal RRT, RRTConnect and PRM. (Time is in seconds)

From the shown results, we can clearly state, that an approach similar to RRT-Path, as proposed in section 3.4.2, has a positive effect on the efficiency of our algorithm. The reason is simple — while sampling path primitives using this approach, we gain a small leeway in the placement of the primitives and can avoid obstacles more easily. Hence the expansion of the tree is also slightly accelerated.

The improvement caused by distancing primitives from each other in the "gap" variation might be at first sight surprising. However, upon closer inspection we can find out, that the effect our modification caused is mainly a decrease in the usage of path primitives. Hence, the algorithm became more similar to normal RRT.

This also means, that moderate usage of path primitives is not much detrimental to the efficiency of the algorithm. Unfortunately, the difficult question of when to use which primitive still remains.

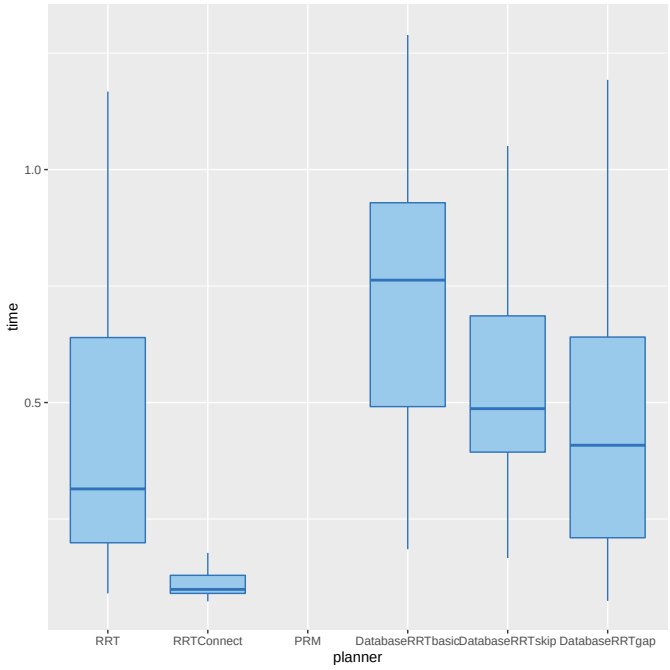

Figure 4.7: Box plot graph with hidden outliers describing the same data as figure 4.6. (Time is in seconds)

What is surprising is the effectiveness of the PRM and RRTConnect algorithms, which cannot be even compared to normal RRT or any of our RRT-based variants in these environments.

4.2 2D agent with joints

In this section we will take a look at a simple agent with multiple links in a 2D environment with rotations. In our case, the agent will be shaped as a simple "snake" with three body parts joint by two links. The links' movements are limited in such a way, that they can be at most about perpendicular to one another (here, the joints' angles are limited to an interval of $[-1.5, 1.5]$ radians). The environment will again be a grid, although this time, with a few extra obstacles to increase the difficulty. Both the agent and the environment are shown in figure 4.8. The full generated tree is not shown this time, since we are working with a 5D configuration space and the visualization in 2D space would be messy.

The primitives were once again generated by placing the agent into the environment, checking for occupancy (this time with an occupancy grid with 25 test points for better resolution) and creating a path towards a randomly selected nearby configuration. This time, however, it was necessary to divide the database into separate directories according to the position of the two joints in the start configuration of each primitive (as explained in 3.3.2). 9 joint starting positions were selected, hence, 9 folders were created,

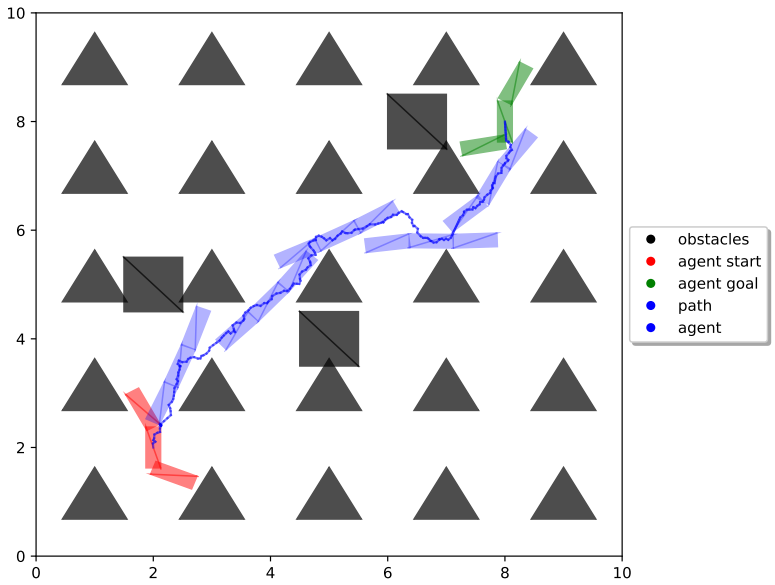

Figure 4.8: The environment with a multiple-link agent and a path connecting the start and goal configurations. In this case the agent had a tendency to straighten itself for most of the path.

each containing approximately 100 primitives. Furthermore, the number of configurations in the primitives ranged from around 30 to 100.

Both findings from our previous environment were adopted when testing the newly created database. That is, the sampling is again done similarly to RRT-Path and deploying primitives close to one another is prohibited.

Our algorithm was tested with four different intervals of samples between each path primitive deployment — 250, 500, 750, and 1000. The resulting plots are shown in figures 4.9 and 4.10 alongside the usual RRT algorithm. This time, we are focusing purely on the comparison with the RRT algorithm.

From the results we can clearly see, that we have finally reached a scenario, where our database-based RRT triumphs over the traditional one. It should be noted, that the improvement is not as drastic as we would wish. Furthermore, it is was necessary to find the appropriate balance of the frequency of using primitives, the distance between two path primitives deployment and the optimal method for sampling the primitive. All these values are influenced by the properties of the environment.

On the other hand, the results also show, that the optimal way to utilize the path primitives is to use them moderately (although also not too little), while having a big enough database to accommodate most possible situations in the environment.

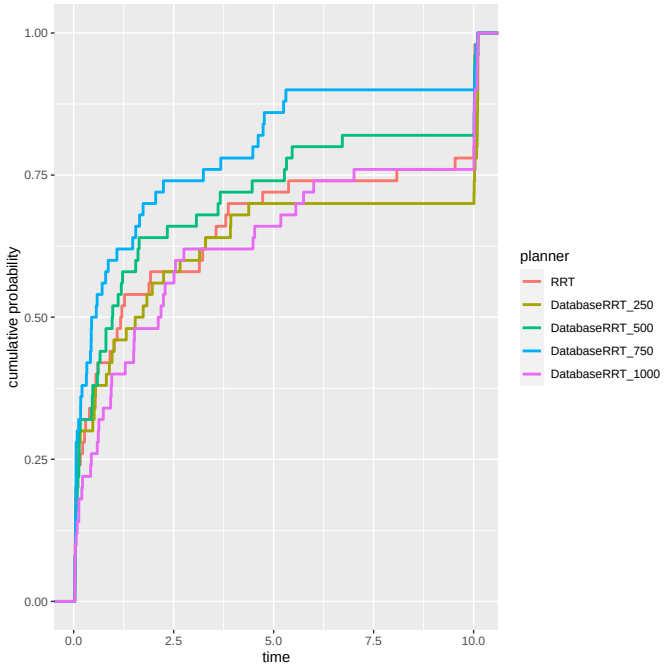

Figure 4.9: The cumulative distribution function describing the performance of the tested algorithms in the advanced 2D environment with a multiple-link agent. The cut-offs on the right hand side means, that the algorithms were not able to find a solution in the allocated time. (Time is in seconds)

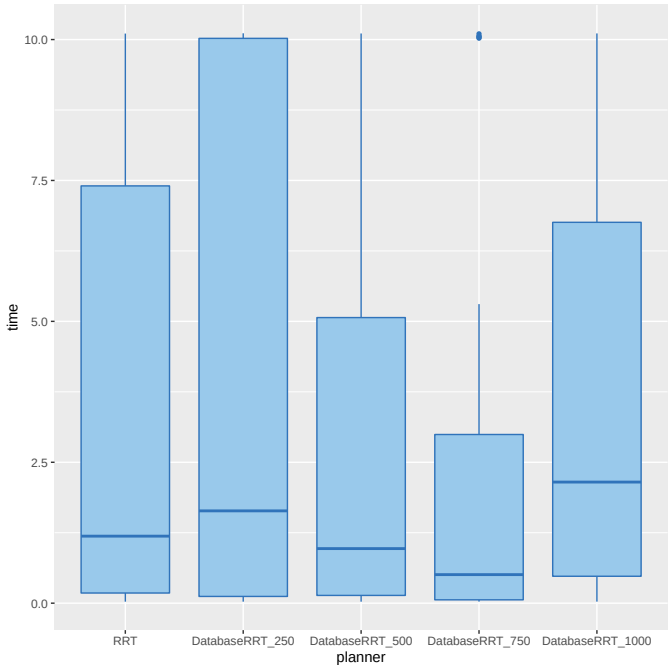

Figure 4.10: Box plot graph with hidden outliers describing the same data as figure 4.9. (Time is in seconds)

4.3 Simple 3D environment

In the last section of this chapter we will use the database-based RRT to solve a simple 3D environment with rotations, that is to say, an environment with 6-dimensional configuration space.

We shall still use a grid-like environment, as they provide sufficient complexity and will give us a more clear view of the situation the planning is in (although rendering it just as an image might still make it probably confusing). In this case, block shaped obstacles and a "double-L" shaped agent shall be used, as seen on figure 4.11.

The database creation will proceed similarly as in section 4.2 with the difference, that the occupation grid is 3D. Method of sampling primitives and keeping gaps between deployed primitives remains unchanged since previous section. Three different intervals between path primitive usages were tested.

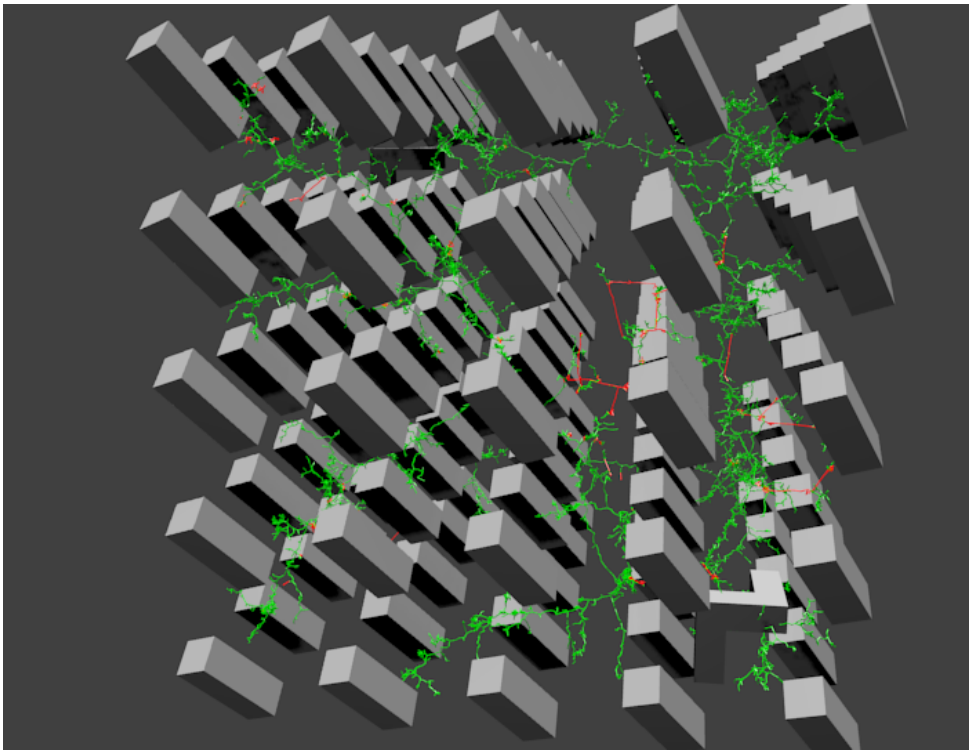

Figure 4.11: A simple 3D environment with block obstacles in a grid and a "double-L" shaped agent (Visible in the lower right of the image). The tree graph was created using the database-based RRT. Green nodes were sampled uniformly, red were created using path primitives.

Once again, we can see the results of benchmarking our algorithms and others in figures 4.12 and 4.13. Unlike in the previous section, we were not able to achieve better results than RRT. It is clear, that the environment was not by any means too easy, as especially normal RRT seemed to be struggling, though not as much as in section 4.2. Despite that, the database-based RRT was not completely ill-suited for this environment, as it even achieved a similar performance to RRT.

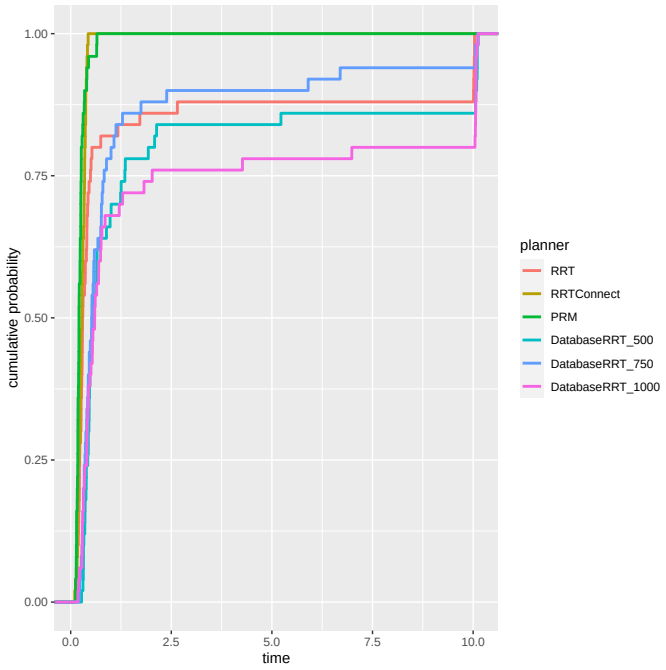

Figure 4.12: The cumulative distribution function describing the performance of the tested algorithms in the 3D environment. The cut-offs on the right hand side once again means, that the algorithms were not able to find a solution in the allocated time. (Time is in seconds)

Judging by the experiences we gained throughout this chapter, it might still be possible to further increase the solving speed of our algorithm by tinkering with the size of the database, the method of sampling primitives, or the interval between usages of primitives. However, such approach would be already akin to "cherry picking" and would be detrimental to possible future attempts at generalizing our algorithm.

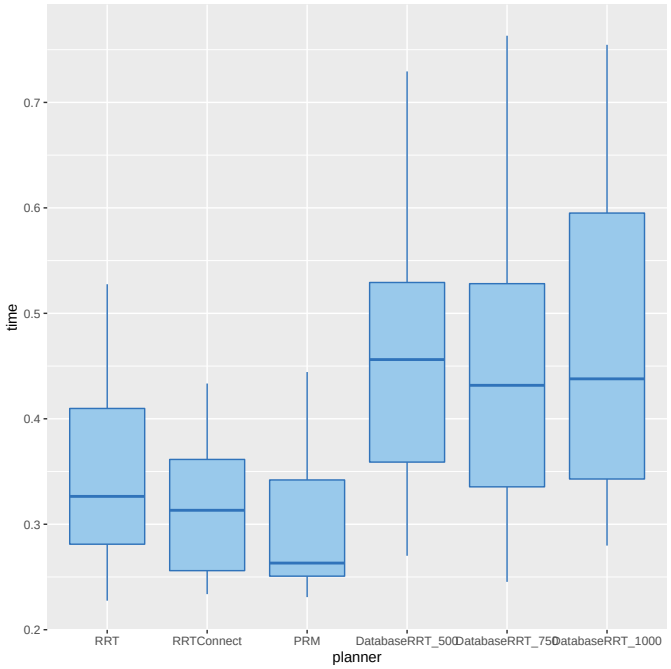

Figure 4.13: Box plot graph with hidden outliers describing the same data as figure 4.12. (Time is in seconds)

Chapter 5

Conclusion

In this thesis, we discussed existing sampling based motion planning algorithms, their advantages, and also their disadvantages. We attempted to tackle one of these disadvantages by introducing a novel way of sampling configuration space by following predetermined path primitives. We have shown a few methods for creating databases of said primitives and used them in an attempt to enhance the RRT algorithm.

In the experimental section, we have tested the proposed algorithm in various environments and compared it against state of the art implementations of sampling based algorithms. The results show, that in simpler environments the path primitives do not provide much improvement for the already highly efficient RRT algorithm and are often rather hindering it. However, in more complex environments, we have seen the database-based RRT not only keep up with with standard RRT algorithm, but in some cases even outperform it.

The best results were obtained by using a large database of path primitives, efficient methods for sampling the primitives, and by deploying the primitives moderately. This leads us to believe, that the path primitives should never overtake the RRT algorithm. Instead, they should only be used when the RRT algorithm encounters a complicated situation hindering the expansion of its tree. However, the question of how to recognize or predict the optimal moment for using a path primitive, remains still largely unanswered and is, therefore, open to further research.

Bibliography

[1] "Rapidly-exploring random trees: A new tool for path planning," Computer Science Department, Iowa State University, Tech. Rep. 9811, Oct. 1998.

[2] Steven M. LaValle, "Planning Algorithms", Cambridge University Press, 2006.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," Int. J. Robot. Res., vol. 30, no. 7, pp. 846–894, Jun. 2011.

[4] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," IEEE Trans. Robot. Autom., vol. 12, no. 4, pp. 566–580, Aug. 1996.

[5] D. Hsu, J. C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in Proc. IEEE Int. Conf. Robot. Autom., vol. 3. Apr. 1997, pp. 2719–2726.

[6] J. M. Phillips, N. Bedrossian, and E. E. Kavraki, "Guided expansive spaces trees: A search strategy for motion- and cost-constrained state spaces," in Proc. IEEE ICRA, vol. 4. Apr./May 2004, pp. 3968–3973.

[7] Vojtěch Vonásek, Jan Faigl, Tomáš Krajník, and Libor Přeučil, "RRT–Path: a guided Rapidly-exploring Random Tree", The Gerstner Laboratory for Intelligent Decision Making and Control Department of Cybernetics, Faculty of Electrical Engineering Czech Technical University in Prague, 2009.

[8] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in Proc. IEEE International Conference on Robotics and Automation ICRA'00., vol. 2. IEEE, 2000, pp. 995– 1001.

[9] Troy McMahon, Aravind Sivaramakrishnan, Edgar Granados, Kostas E. Bekris, "A Survey on the Integration of Machine Learning with Sampling-based Motion Planning", Rutgers University, 2022.

[10] Michael Janner, Yilun Du, Joshua B. Tenenbaum, Sergey Levine, "Planning with Diffusion for Flexible Behavior Synthesis", 2022.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische mathematik, vol. 1, no. 1, pp. 269–271, 1959.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Trans. Syst. Sci. Cybern., vol. 4, no. 2, pp. 100–107, 1968.

[13] OMPL — The Open Motion Planning Library.
https://ompl.kavrakilab.org/ (cited: May 2024).

[14] RAPID — Robust and Accurate Polygon Interference Detection
http://gamma.cs.unc.edu/OBB/ (cited: May 2024).

[15] Blender
https://www.blender.org/ (cited: May 2024).

[16] OpenCV
https://opencv.org/ (cited: May 2024).

[17] Planner Arena.
https://plannerarena.org/ (cited: May 2024).

[18] Mohamed Elbanhawi (Student Member, IEEE) and Milan Simic, "Sampling-Based Robot Motion Planning: A Review", 2014.

[19] B. R. Donald, "A search algorithm for motion planning with six degrees of freedom," Artif. Intell., vol. 31, no. 3, pp. 295–353, Mar. 1987

[20] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," Int. J. Robot. Res., vol. 5, no. 1, pp. 90–98, Mar. 1986.

[21] Louis Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", Stanford University, 1975.

[22] I. Al-Bluwi, T. Siméon, and J. Cortés, "Motion planning algorithms for molecular simulations: A survey," Comput. Sci. Rev., vol. 6, no. 4, pp. 125–143, Jul. 2012.

[23] S. R. Lindemann and S. M. LaValle, "Steps toward derandomizing RRTs," in Robot Motion and Control, K. Kozłowski, Ed. London: Springer London, 2006, pp. 287–300.