Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

# An Improved RRT* Algorithm for Multi-Robot Path Planning

Poludin Mikhail

Supervisor: Tiago Pereira Do Nascimento, Ph.D
May 2022

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

| | | | |
|---|---|---|---|
| Student's name: | **Poludin Mikhail** | Personal ID number: | **492571** |
| 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:

**An Improved RRT* Algorithm for Multi-Robot Path Planning**

Bachelor’s thesis title in Czech:

**Vylepšený algoritmus RRT* pro plánování cesty pro více robotů**

Guidelines:

Implement RRT and RRT* path planning algorithms for Unmanned Aerial Vehicles (UAV) in 3D and 2D scenarios using C++ and ROS.
Evaluate performance and robustness of RRT type algorithms in different situations with varying parameters. Introduce different obstacle avoidance approaches (such as binary search of collisions) and compare the results when using with RRT family algorithms. Modify RRT and RRT* algorithms to consider path finding for multiple drones. Perform real robot experiments with multi-rotor UAVs.

Bibliography / sources:

[1] Steven M. LaValle “Planning algorithms”, University of Illinois 2006
[2] Steven M. LaValle “Rapidly-exploring random trees: A new tool for path planning”, Iowa State University 1998
[3] W. Zu, G. Fan, Y. Gao, Y. Ma, H. Zhang and H. Zeng, "Multi-UAVs Cooperative Path Planning Method based on Improved RRT Algorithm," 2018 IEEE International Conference on Mechatronics and Automation
[4] M. Kothari, I. Postlethwaite and D. Gu, "Multi-UAV path planning in obstacle rich environments using Rapidly-exploring Random Trees," 48th IEEE Conference on Decision and Control (CDC), 2009

Name and workplace of bachelor’s thesis supervisor:

**Tiago Pereira Do Nascimento, Ph.D. Multi-robot Systems FEE**

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

Date of bachelor’s thesis assignment: **21.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

| | | |
|---|---|---|
| Tiago Pereira Do Nascimento, Ph.D. | prof. Ing. Tomáš Svoboda, Ph.D. | 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

First of all, I would like to thank my supervisor for his guidance, feedback, and supportive mindset.
I would also like to thank Hendrik Scheepers de Bruin, who guided me during the winter semester.
Most importantly, I thank my family for giving me the encouragement and the opportunity to finish this thesis.

## Declaration

## Abstract

This thesis includes a brief overview of the UAV path planning and a detailed explanation of the algorithms implemented in C++. The implementation of the RRT and RRT* algorithms were carried out and extended to handle the generation of trajectories for multiple drones. Two obstacle avoidance approaches were introduced and tested with both RRT family path-planning algorithms. Experiments of autonomous UAV flight in a forest-like environment were conducted in both simulation and real life; for this purpose, detection and mapping of trees using an active lidar sensor was implemented.

**Keywords:** RRT, RRT*, Path planning, Obstacle avoidance, UAV.

**Supervisor:** Tiago Pereira Do Nascimento, Ph.D
Karlovo náměstí 13, Praha 2

## Abstrakt

Tato práce obsahuje stručný přehled plánování cest pro bezpilotní drony a podrobné vysvětlení algoritmů implementovaných v jazyce C++. Byly implementovány algoritmy RRT a RRT*, které byly rozšířeny tak, aby zvládaly generování trajektorií pro více bezpilotních letounů. Byly zavedeny dva přístupy pro vyhýbání překážkám a testovány s oběma algoritmy plánování cest z rodiny RRT. Experimenty autonomního letu bezpilotního dronu v prostředí podobném lesu byly provedeny jak v simulaci, tak v reálném prostředí; za tímto účelem byla implementována detekce a mapování stromů pomocí aktivního lidar senzoru.

**Klíčová slova:** RRT, RRT*, Planování cest, Vyhýbání překážkam, Bezpilotní dron.

**Překlad názvu:** Vylepšený algoritmus RRT* pro plánování cesty pro více robotů.

# Contents

# Figures

# Tables

# Chapter 1

# Path and motion planning

Motion planning and path planning is a challenge to find a sequence of moves for an agent to reach the goal state from the start state. Motion planning algorithms, in particular, give the programmer a way to find this sequence of moves so that it satisfies certain constraints (such as obstacles or computational time). Today, there are a wide variety of completely different approaches, each of them best suited for its particular purpose.

This project is concentrated on dealing with path planning for drones. Autonomous drone navigation and route planning can serve in many applications. For example, search and investigation missions, where it can be useful to use unmanned aerial vehicles; in an environment with high obstacle density, such as caves, forests, etc. An operator could send his autonomous drone for surroundings investigation while dealing with other tasks that require human control. An optimised path-planning algorithm will provide a way for this drone to find its trajectory fast, which is essential for rescue operations. Another common application of drones is to film videos, and it would usually be useful to have a drone that incorporates motion planning and autonomous obstacle avoidance. The control system of such an aerial vehicle would follow a provided target, while the drone is flying according to a planned trajectory.

As the demand for autonomous drones for path planning grows, a wide range of problems may arise. The first problem is that there are many different environments, and implementing a simple path planner suitable for most world settings is a difficult and expensive task. The second problem, for example, can be the optimisation of motion planning, so that it can be used on a moving vehicle, which is a task requiring complex automatic control systems. An important topic raised during the implementation of this project is the planning of routes for multiple aerial vehicles. In this case, the drones need to take into account the positions and movements of other drones. But this information needs to be translated from one vehicle to another. This can be done through shared communication systems, i.e. having one drone sending positions of every other drone. Or, every member of a drone swarm could rely on its own sensors and data gathering and tracking movements of others autonomously.

Several algorithms for path planning and obstacle avoidance for single and multiple drones were implemented during this project. Together, they all provide a solid basis for choosing the right parameters and algorithms for certain tasks, such as autonomous path planning and movement in a forest-like environment. The code and tools were specifically designed to allow easy introduction of new algorithms into this project or link up implemented work to another project.

## 1.1 Objectives of the work

The main goal of this project is to propose and implement improvements on the RRT and RRT* path planning algorithms to allow them to consider building paths for multiple UAVs. To achieve that, it is necessary to prepare by implementing the following algorithms in a structured way.

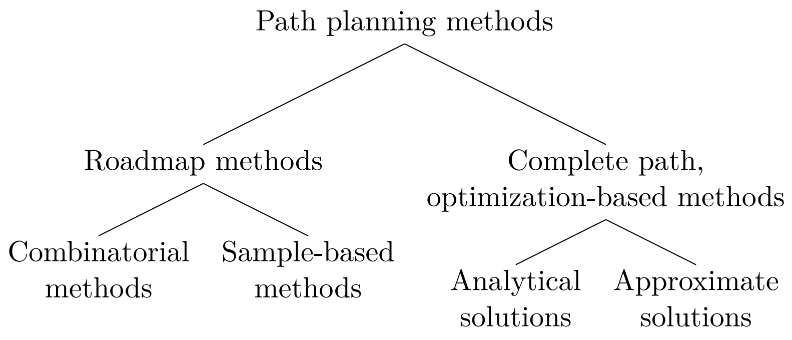
- RRT path planning algorithm,
- RRT* path planning algorithm,
- *Point UAV - Inflated objects* obstacle avoidance,
- *Binary search of collisions* obstacle avoidance.

The final objective is to test and compare the above algorithms on the performance side. In addition, conduct experiments of autonomous flight through a forest-like environment both in simulation and on a real drone to prove practical usability of the planning algorithms as well.

# Chapter 2

# Examples of path planning approaches

Here is a small overview tree of different motion planning approaches:



Roadmap algorithms are generally considered easier to implement and their computational time is better, but optimisation-based approaches may find the path with a smaller energy loss. For the purpose of this thesis, this chapter will focus on roadmap methods. Roadmap methods in particular are split into two different ones; here are some typical examples of these algorithms:

- Combinatorial methods:
  - Using Voronoi diagrams
  - Cell decomposition
  - Visibility graphs
- Sample-based methods:
  - Probabilistic roadmaps
  - Rapidly exploring random trees

## 2.1 Combinatorial path planning

The main concept of combinatorial path planning is to divide the configuration space $Cspace$ into connected regions by any type of algorithm and then to find the way from the points derived from these regions. The configuration space of a robot, $Cspace$, is a set of all possible positions in which the robot may be. In this project, mainly focused on UAV-like agents, we will take all the possible positions of a drone in Euclidean space, with 3 coordinates and 3 angles of rotation. The crucial part of the efficiency of the combinatorial method is the algorithm to divide the search space.

The space division can be done by many methods and the most naive one is to divide the space into equal square blocks. For a bit more complex example, let us take the polygon cell decomposition approach. As shown in Figure 2.1, the search space is divided into polygons, using a triangulation algorithm. Then, the centre of gravity is calculated, and the final roadmap is built out of them.

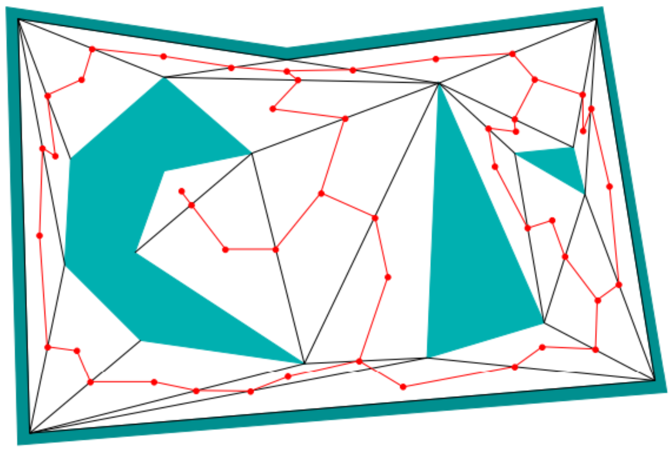


**Figure 2.1:** Polygon cell decomposition with a roadmap example, [1].

Here is another example, Figure 2.2. This approach is called cylindrical decomposition. The follow-up roadmap is derived from the centres of constructed vertical lines, represented by two end points. In the picture, one of the possible start-goal paths is shown with the orange line.

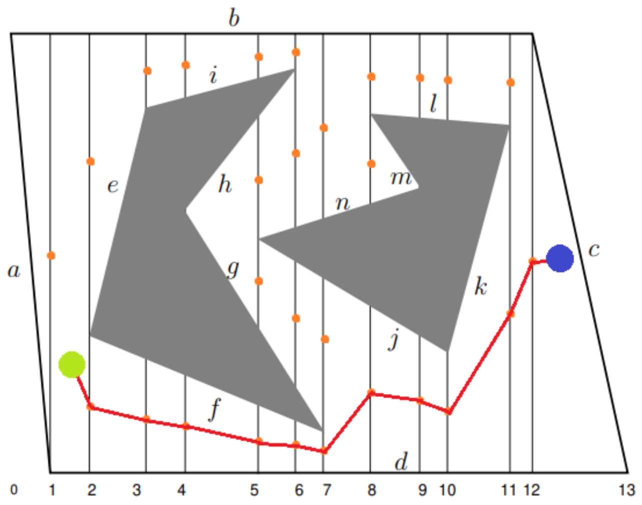


**Figure 2.2:** Cylindrical cell decomposition example, [2] with an added path in orange.

For more efficient search and decomposition, tree-shaped graphs are used. The most common ones are the so-called quad trees (oc trees in case of 3D path planning). The areas of interest are recursively divided into smaller ones, Figure 2.3. Such trees make the sections smoother, and therefore, the found path will get closer to the optimal one.

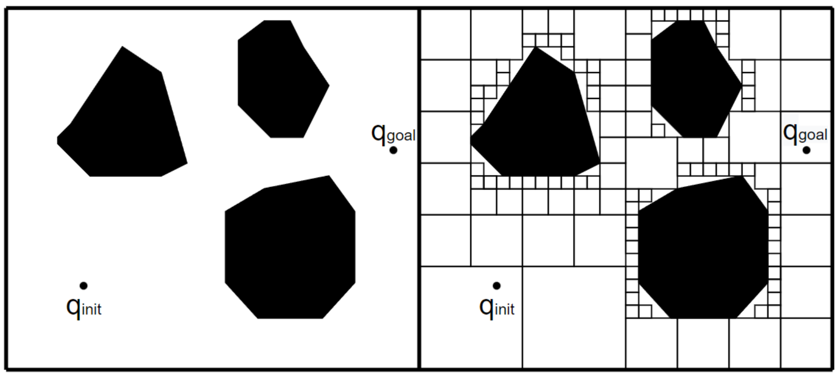


**Figure 2.3:** "Quadtree" decomposition, redrawn/inspired by [3].

## 2.2 Sample-based path planning

The main idea of the sample-based path planning approach is to implement a search that explores the given search space $Cspace$ with a certain sampling scheme. The exploration is usually done with obstacle avoidance modules that tell the sample module how to create new samples in the particular search space.

The count of discrete samples/points can be infinite, since the real-life problem is usually represented by real numbers that can be divided into smaller and smaller ones. Since computers have a finite amount of memory and the computational time is often constrained by a given task, the number of samples/iterations of the algorithm is also purposely limited. However, when using particular methods, such as the RRT* algorithm, it is more than enough to have a relatively small finite number of iterations to find the close-to or the optimal way.

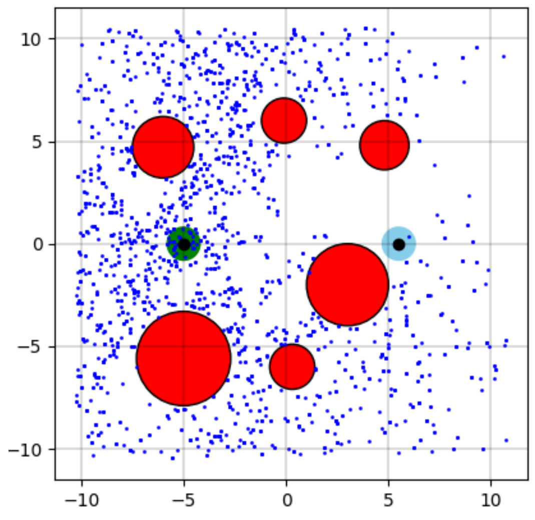


**Figure 2.4:** Random point sampling in 2D search space example. Restricted regions are shown in red.

For better understanding of the sampling idea, consider a 2D square search space (Figure. 2.4), which needs to be explored. Sample-based approaches split the whole rectangle into smaller ones and gradually fill it with 2D vectors/points of interest to investigate. Generation of points of interest is called "sampling". Samples can be produced using different approaches. The space could be split uniformly, exponentially from the start position, or even randomly. The random sampling is the main concept behind so called **R**apidly Exploring **R**andom **T**rees, that happened to be the main focus of this project.

# Chapter 3

# Related works

Planning path and motion is a broad area with many approaches and many articles have been written about it. This chapter serves as a general overview of the literature studied to establish a solid foundation for understanding the path planning concept from all aspects.

Briefly discussed in Chapter 2, roadmap methods are one of the main path planning movements in research. It was experimentally proven that due to the simplicity and intuitiveness of the algorithm, it is a fast and reliable method. It can also be applied to almost all holonomic robot systems [4]. PRM can be further improved and modified according to practical needs. For instance, it can be applied to object manipulation planning, a so-called Two Level Fuzzy PRM [5] modifies the algorithm for that purpose by introducing edge probabilities. PRM can be optimised to work with hundreds of thousands of obstacles. Lazy PRM [6] postpones collision detection for the second phase of execution, drastically reducing computational time. This is possible because most of the collision checks will not be valid in the resulting path, and, if postponed, thousands of conditions will not be conducted.

Voronoi diagrams [7] and visibility graphs [8] are often compared. The Voronoi diagram can be built in a time of only $O(nlogn)$ while the most efficient known algorithms for building visibility graphs can even be $O(n^2)$ complexity in the worst case. Since the Voronoi diagram has much fewer edges, extracting a path from a roadmap based on the Voronoi diagram is also less time-consuming than extracting from the visibility graph [9]. However, the optimality of the path obtained using the Voronoi diagram can be pretty bad. But on the other hand, the visibility graphs generate a trajectory that can be too close to obstacles. That is why, for many applications, it is better to find a compromise between two. For example, Generalised Voronoi Diagram [10] is suitable for polygon-based environments and can be efficiently applied to narrow entrances and coridors, which can be problematic for some algorithms.

Potential field methods [11] are interesting examples of path planning algorithms. A potential field is mapped onto a given search space inspired by static electric fields. An agent or vehicle is represented as a positive or negative charge that will be attracted to the point of lowest potential. This method uses a simple gradient descent algorithm [12] to find a path. However, the gradient descent only finds a local minimum, which could not be the goal state, and the algorithm may stall without finding a path. The navigation function [13] is introduced to solve this problem. It creates such potential field mapping that there could only be one local minimum (global minimum) in the whole state space.

RRTs, being the main topic of this thesis, are one-of-a-kind tools for path planning. They can be applied to basically any search and motion planning tasks. For instance, algorithm for ubran autonomous car driving [14] constructs RRT structure, keeping in mind the constraints given by the circular movements of a car using a so-called forward motion model. Of course, the RRT algorithm can be modified to find routes faster in some cases. Bidirectional RRT [15] keeps two processes running at the same time - growing two independent trees from a start and a goal point. The algorithm runs until two tree branches are close enough to be connected. Then, the path is constructed as a way from the start to that connection point plus the way from the connection point to the goal. RRT-Connect [16] modifies the expansion step of the RRT. It iteratively grows the tree from the last node added until the time when the goal is reached. This can significantly improve the speed of space exploration; however, this approach does not perform well with narrow entrances and tunnels because of the small probability of exploring the details of the search space right after the start.

If the path finding problem extends to a swarm of agents or vehicles [17], the algorithms tend to become more sophisticated and complex. First, it must be decided which type of agent relationship will be used: centralised or distributed [18]. The centralised approach makes use of one main agent planning all the trajectories and commanding other vehicles. Introducing communication between drones brings certain problems to the scene [19]. However, if the system is implemented without agent-to-agent communication, the detection and localisation modules need to be programmed. For example, using visual cameras [20] or ultraviolet markers onboard [21]. The planning of the path of the swarm was also successfully attempted using reinforcement learning (Q-learning) [22]. This method can also be split into two types: having a single guiding drone with a neural network or having multiple drones, each having its own neural net. These neural networks would essentially have the same architecture but different weights for every particular purpose.

Path planning cannot be done without obstacle avoidance modules. It is essential to use sensors to map the environment around the agent; e.g., expensive outer lidars or low-cost infrared sensors [23]. Detection can also be done using neural networks and segmentation using only an rgb or depth camera [24]. After mapping the obstacles, one must implement an intersection module that tells whether an obstacle affects the proposed path. A possible solution is to have all obstacles represented as standard convex bodies [25], such as cylinders, shperes or cones. The agent then plans a path with constraints given by these objects. [26] provides a detailed and extensive overview of different concepts of collision avoidance.

My project, compared to the projects in the reviewed literature above, first describes the detailed algorithm implementation process and, second and important, proposes a new approach for planning multiple UAV paths in an environment defined by simple objects. While reviewing articles, I have not found a single one implementing path planning for multiple drones the same way. The designed algorithms are tested on the performance side, and the experiments are conducted not only in simulation, but also in the real world.

# Chapter 4

# Background theory

## 4.1 RRT algorithm

Rapidly Exploring Random Tree (RRT) is a sample-based approach algorithm. It is widely used in autonomous robotics, mainly because it is very intuitive and effective. It is an algorithm to quickly scan high-dimensional spaces by constructing a search tree graph (Figure 4.1) from randomly selected points in obstacles-free areas. RRT generates very evenly distributed rectangular graphs because the new random state is always attached to its nearest neighbour.

It is a probabilistically complete algorithm, which means that if the algorithm is run for a long enough time, the graph will have a solution, the path found, if one exists. It is also a fast method compared to other planning algorithms. The most significant disadvantage of RRT is the fact that the found way is not always optimal. For UAVs, when the search space is usually huge and a wide variety of positions to be in is available, this approach would usually find a very zigzag-shaped path. To reduce that to a certain degree, the maximum possible distance $d$ between two states is defined. When the time to connect two vertices arrives, it is verified that the distance between them is less than $d$. If it is not, then this connection cannot be made.

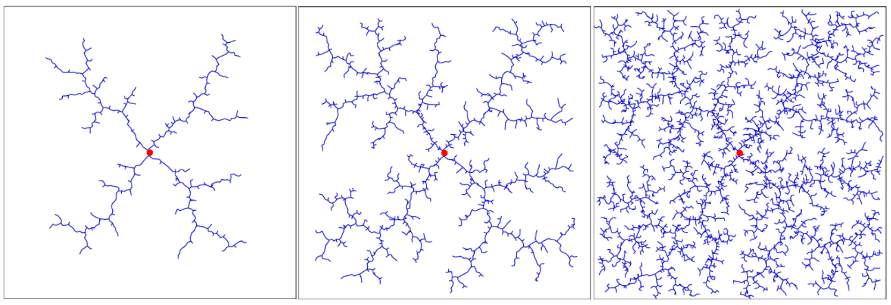


**Figure 4.1:** Rapidly exploring random tree expansion example, [27].

### 4.1.1 Explanation of the algorithm

The search starts with a certain starting point and a goal point in multidimensional space. Let the starting point be $S_0 \in C_{space}$, and the goal be $S_g \in C_{space}$. Also, the goal radius $r_g$ is defined - the radius of a sphere around $S_g$, in which the path is considered found. The tree graph $T$ is initialised with one single node, $S_0$. A random point $S_{new} \in C_{space}$ is generated at each iteration of the algorithm. Then $S_{new}$ is investigated to obey the given constraints. First, it is controlled to determine whether it lies inside any obstacle using obstacle intersection search methods. If the answer is negative, the algorithm finds the nearest point $S_{nearest}$ in the whole $T$ and checks if a path between $S_{new}$ and $S_{nearest}$ is collision-free. If so, then $S_{new}$ is added to $T$ with $S_{nearest}$ as the parent node. Lastly, if $S_{new}$ is inside $r_g$, then the path is considered found and the only thing left to do is extract the path from $T$, which can be done by iteratively going back through the parents, from $S_g$ to $S_0$.

### 4.1.2 RRT algorithm pseudocode

Here are the pseudocodes of the RRT path-finding algorithms that were followed in my implementation - Algorithms 1 and 2:

**Algorithm 1** RRT algorithm for path finding

**Input: Initial state - $S_0$, goal state - $S_g$, maximum number of vertices - $n$.**
**Output: Array of states/path - $P$.**

```
1: $T \leftarrow$ tree_init($S_0$)
2: number_of_iters $\leftarrow 0$
3: while number_of_iters $< n$ do
4:     $S_{rand} \leftarrow$ get_random_state()
5:     $S_{nearest} \leftarrow$ get_nearest_state($S_{rand}$)
6:     if path_is_clear($S_{nearest}, S_{rand}$) then
7:         $T$.add_new_edge($S_{nearest}, S_{rand}$)
8:     end if
9:     if $S_{rand} \in S_{goal}$ then          ▷ If new state is inside the goal, consider path found
10:        $T$.add_new_edge($S_{rand}, S_{goal}$)
11:        return extract_path_from_RRT($T$)
12:    end if
13:    number_of_iters $\leftarrow$ number_of_iters $+1$
14: end while
```

**Algorithm 2** Function to extract a path from an RRT tree

**Output: array of states/path - $P$**

```
1: function EXTRACT_PATH_FROM_RRT(Tree T)
2:     $P \leftarrow S_g$                ▷ P - path/array of following each other states
3:     $S_{tmp} \leftarrow S_g$
4:     while $S_{tmp} \neq S_0$ do
5:         $P$.push_back($S_{tmp}$)
6:         $S_{tmp} \leftarrow S_{parent\_of\_tmp}$
7:     end while
8:     return $P$
9: end function
```

## 4.2 RRT* path planning algorithm theory

To find a path that is shorter than an average path produced by the RRT algorithm, **RRT*** is introduced. It is essentially an optimised version of RRT. It requires more computations, and, unlike RRT, when the path is found, it can continue to search and optimise the path for a given number of iterations $n$. It is an asymptotically optimal algorithm. This means that theoretically, when the number of iterations approaches infinity, the algorithm finds the shortest possible way. The growth of RRT* is shown in Figure.4.2.

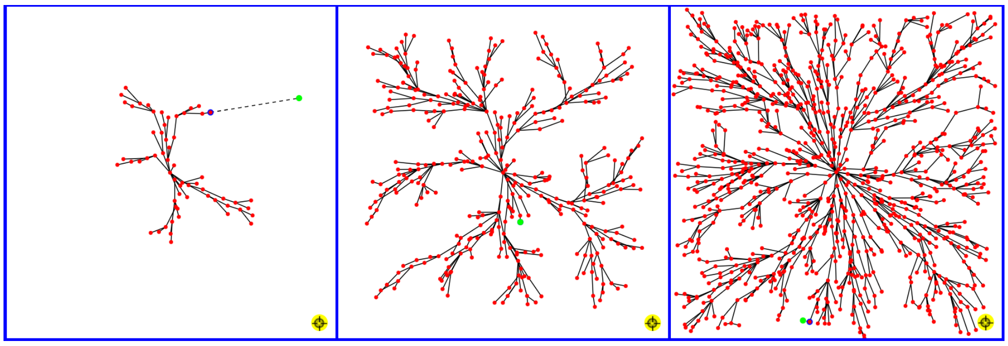


**Figure 4.2:** Example of RRT* expansion, [28].

### 4.2.1 Contrast to RRT

The basics of RRT* are the same as those of RRT (generating a random state and finding the nearest neighbour). However, a couple of improvements and additions generate completely different results:

- Tracking distance from $S_0$ to each vertex of a tree. Each vertex would now have a **cost**, which is substantially a sum of distances along the current branch of the tree. Now, by performing the step of finding the nearest state, the algorithm will also look for a state $S_{best}$ that provides the $\min\{\text{cost}(S_{best}) + \text{dist}(S_{best}, S_{new})\}$ within a given neighbouring radius. Now we have a new way to connect a random state to the tree in the state $S_{best}$,

  $$S_{best} = argmin_i\{cost(S_i) + dist(S_i, S_{new}) \mid dist(S_i, S_{new}) \leq r_n\}$$

  This feature tends to eliminate rectangular shapes in a graph.

- Second feature is called **rewiring** of the graph. It happens right after the new state $S_{new}$ has been connected to $S_{best}$. All nodes $S_i$ within a defined neighbour radius $r_n$ are inspected to see whether their cost would decrease if their parent vertex was $S_{new}$. If it is true, then the tree is rewired so that $S_{new}$ is now the parent of $S_i$. This step makes the path look more polished, without rough angles, as opposed to RRT.

### 4.2.2 RRT* algorithm pseudocode

The pseudocode explaining the RRT* algorithm is shown in algorithm 3:

**Algorithm 3** RRT* algorithm to find a path

**Input: Initial state -** $S_0$**, goal state -** $S_g$**, maximum number of vertices -** $n$**, neighbour radius -** $r_n$**.**
**Output: array of states/path -** $P$**.**

```
1: $T \leftarrow$ tree_init($S_0$)
2: number_of_iters $\leftarrow 0$
3: while number_of_iters $< n$ do
4:     $S_{rand} \leftarrow$ get_random_state()
5:     $S_{best} \leftarrow$ get_best_cost_state($S_{rand}$, $r_n$)      ▷ Find the cheapest neighbour in $r_n$
6:     if path_is_clear($S_{best}$, $S_{rand}$) then
7:         $T$.add_new_edge($S_{best}$, $S_{rand}$)
8:         $S_{rand}$.cost $\leftarrow$ dist($S_{rand}$, $S_{best}$) + cost($S_{best}$)
9:     end if
10:    for $S_i \in$ Neighbours do          ▷ Go through all neighbours and rewire the tree
11:        $S_{parent} \leftarrow S_i$.parent
12:        if dist($S_{new}$, $S_i$) + cost($S_{new}$) < dist($S_{parent}$, $S_i$) + cost($S_{parent}$) then
13:            $S_i$.parent.remove_child($S_i$)
14:            $S_i$.parent $\leftarrow S_{rand}$
15:        end if
16:    end for
17:    if $S_{rand} \in S_{goal}$ then          ▷ If new state is inside the goal, consider path found
18:        $T$.add_new_edge($S_{rand}$, $S_{goal}$)
19:        return extract_path_from_RRT($T$)
20:    end if
21:    number_of_iters $\leftarrow$ number_of_iters +1
22: end while
```

# Chapter 5

# UAV flight in GNSS-Denied environments using RRT and RRT*

## 5.1 Preparation and tool investigation

Before the implementation started, the first thing to do was investigate the simulation software used by the Multi Robot Systems group.

For simplicity of UAV control and simulation, **Robot Operating System** (ROS) is used. It provides services such as hardware abstraction, implementation of commonly used functionalities, and most importantly, for this project: message passing between several processes and package management. Different running processes are represented in a graph architecture, where the edges show message-based communication. The user can create abstract structures called nodes that can be subscribed to or published on a specific topic. Such an architecture is also good for debugging and error control. It is a shell application, so to see all the messages that are sent through the given topic, you can simply write $*rostopic echo /topic_name*. Official tutorials [29], were studied to learn how to use ROS correctly.

Graphical visualisation is performed with two simulators: **Gazebo and Rviz**. For the sake of simulating drones as in the real world, Gazebo is used. For the implementation of path planning and 3D tree visualisation, I used Rviz. It only takes publishing formatted data from your code to a couple of ROS topics to achieve that, as Rviz supports ROS. This simulator can show structures as simple as spheres and complex objects as polygon meshes. It also accepts arrays of points that can be used as a way to graphically visualise tree-shaped figures exactly as is needed.

To monitor different running processes and ROS servers, I used **htop**. This tool also helps to kill applications so that nothing unnecessary remains while next simulations are run. **Tmux** - terminal multiplexer is a tool that is also useful for the execution of multiple processes at the same time. It is a way to efficiently use several terminal windows at a time (sweep through them, create sub-panels).

The main **programming language** I decided to use for my project is **C++**. When running algorithms such as RRT and RRT*, and using them live-action on a flying drone, it is preferred to improve performance and minimise the execution time as much as possible. Python is considered to be slower than C++, so for a project that contains many numbers, searches, and computations, C++ is much better suited.

I used **python** and **matlab** for data analysis and plotting graphs. To manage data after execution of a one program and use it later in another one (serialisation of certain objects), I used **JSON** - JavaScript Object Notation file format, which is a simple text-based way to send and store data in a structured and simultaneously readable shape. Despite its name, it is a language-independent data format and can be used with python and C++.

## 5.2 Project source files and structure

In this section, a small part of the source files will be briefly and adequately explained. Because it would be inappropriate to explain all aspects of the code here, the project is publicly available on [30].

### 5.2.1 Environment classes and usage of the main function

#### World and Object classes

These classes are meant to be used to create and store different objects in the 3D space. This ensures easy working with visualisation of objects and obstacles in the simulators. Some World methods accept a ROS publisher and send all object data to Rviz in a specified form. Rviz then shows it in its simulation. The Object class is used to contain information about a single object, such as its 3D position and sizes. Sphere and Cylinder classes publicly inherit from the Object, so that they can be differentiated while dealing with obstacle avoidance.

#### Three-dimensional point class - Vec3

Instances of this class are essentially 3D points that only contain their $x$, $y$ and $z$ coordinates. The class has methods and overloads of operators that provide easy maths operations, for instance, these: dot product, division, distance between vectors, norm of a vector. It also has a method to generate a random point in space that is used drastically in random search trees.

#### Main function and TestSelector class

Considering the fact that there are many comparisons of different scenarios and simulations, the TestSelector class was implemented. The main function conveniently calls one of the available test scenarios, which are represented with a simple enum type. In each of the written scenarios, the first thing to do before testing set-up algorithms and their classes is to initialise a new ROS node, which will be associated with a drone that will have the code running. The ros::Publisher and ros::Subscriber object types will be used to handle the desired ROS topics. To see how path planning methods work, I created a subscriber to a odometry topic, publisher to drones velocity control and a publisher to Rviz with a name *visualization_marker*.

### 5.2.2 Tree structure implementation

I decided to represent the whole tree in a classical way as a bunch of independent nodes that all contain an attribute pointer to their parent instance. Each of them also has an array of pointers to its own child nodes.

### Tree node class - Node

Instance of this class will contain 3 coordinates in a Vec3 object form, the cost of this node, a raw pointer to a parent node, and a vector of shared pointers to all of its children.

```
class Node {
public:
    Vec3 coords;
    Node *parent = nullptr;
    bool inside_the_goal = false;
    std::vector<std::shared_ptr<Node>> children;
    double cost;
...
}
```

Class includes methods for adding children, changing a parent, finding all neighbour Nodes placed in a given radius, and finding the nearest Node. These are frequently used throughout the entire execution of the RRT algorithm.

For *get_neighbors_in_radius* and *find_the_closest_node* methods, *std::queue* was used to sequentially go through all nodes and check if they satisfy a certain condition.

### RRT_tree class

The pointer to a tree root is stored in a RRT_tree instance. In addition to the constructor and a couple of useful methods, such as serialising the entire tree into a JSON file, it involves the *find_path* method, which accepts the algorithm to be used and grows a tree according to that algorithm.

## 5.3 Implementation of the algorithms

The cornerstone of actual RRT and RRT* classes is an Algorithm class. I decided to use the virtual function declaration to ensure the simplicity of adding more path-finding functions when needed. The virtual function of the base class Algorithm *find_path_according_to_alg* is redefined in the derived classes RRTAlgorithm and RRTStarAlgorithm.

```
class Algorithm {
public:
    virtual std::vector<Vec3> find_path_according_to_alg(
                                        const World *world_ptr,
                                        const AvoidanceAlgorithm &
                                            avoid_alg,
                                        const Node *root,
                                        const Vec3 &start_point,
                                        const Vec3 &goal_point,
                                        double goal_radius,
                                        double neighbor_radius,
                                        double droneRadius) const=0;
    ...
}
```

After the last node has been added to the tree and the path has been successfully found simultaneously, this path must be extracted from that tree to separate the node array. There is a static *find_way_from_goal_to_root* method for this particular purpose.

### 5.3.1 RRT algorithm implementation

The definition of the *find_path_according_to_alg* virtual function is in the RRTAlgorithm class. It is essentially an implementation of the pseudocodes 1 and 2 using all the useful methods described above.

The code is designed in a way that allows you to easily change the dimension of the search: 1D, 2D or 3D. To test the algorithm, I straightforwardly decided to make the search area as trivial as possible - a rectangle with a side of $2 \cdot dist(S_0, S_G)$ and a centre on a line connecting them. The start state $S_0$ is located on one side and the goal $S_G$ on the other, as shown in the figure 5.1.

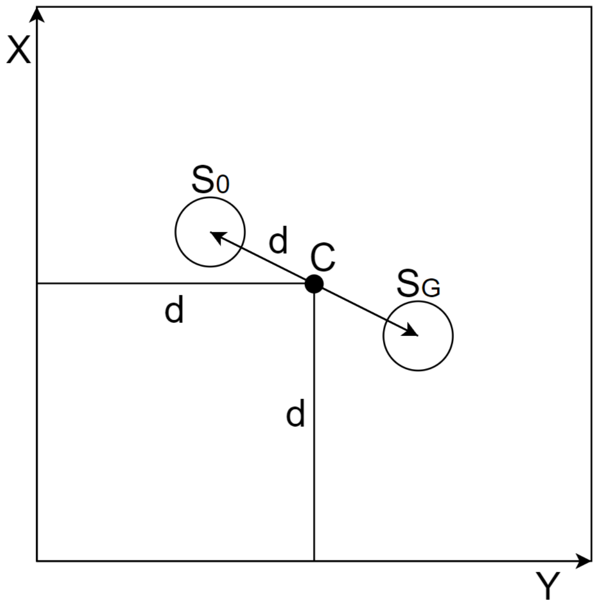


**Figure 5.1:** Trivial search area definition, 2D example.

More sophisticated implementations could define this area as an ellipse with two focusses in $S_0, S_G$.
I decided to represent obstacles as spheres and cylinders because they are the easiest to work with and essentially cover my needs to test the algorithms. After generating a random point, this point needs to be checked on whether it is inside any obstacle or whether the obstacle is located somewhere along the flying trajectory (line from the closest point to the current, random one). I programmed two approaches to solve this problem, which are described in Chapter 6.

#### Results

The algorithm finds a route successfully and quickly in both 2D and 3D scenarios. Here are some examples of simulations without adding obstacle avoidance. Figure 5.2 shows several simulations in Rviz. The path is found successfully; however, it is by far not an optimal path. Figure 5.3 represents the difference between the paths depending on the maximum distance between the nodes. When using a step size that is too small (bottom left), algorithm requires much more time to find a way; however, usually the found way is smoother than when using a big step size (bottom right).

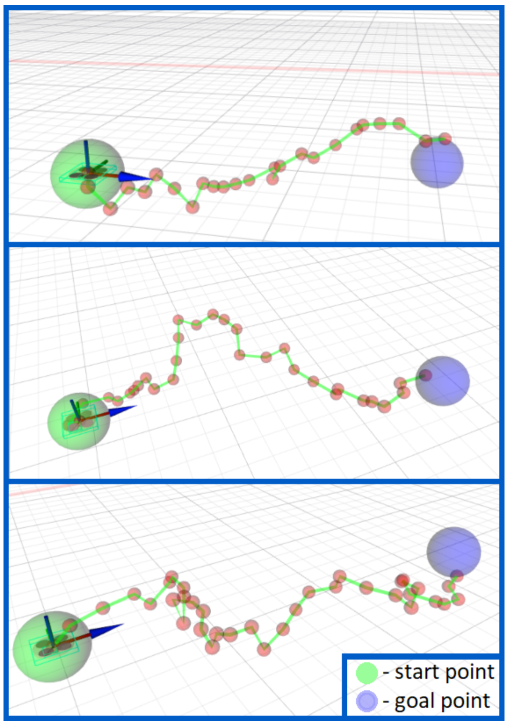


**Figure 5.2:** 3D RRT found path without obstacles, Rviz simulation.

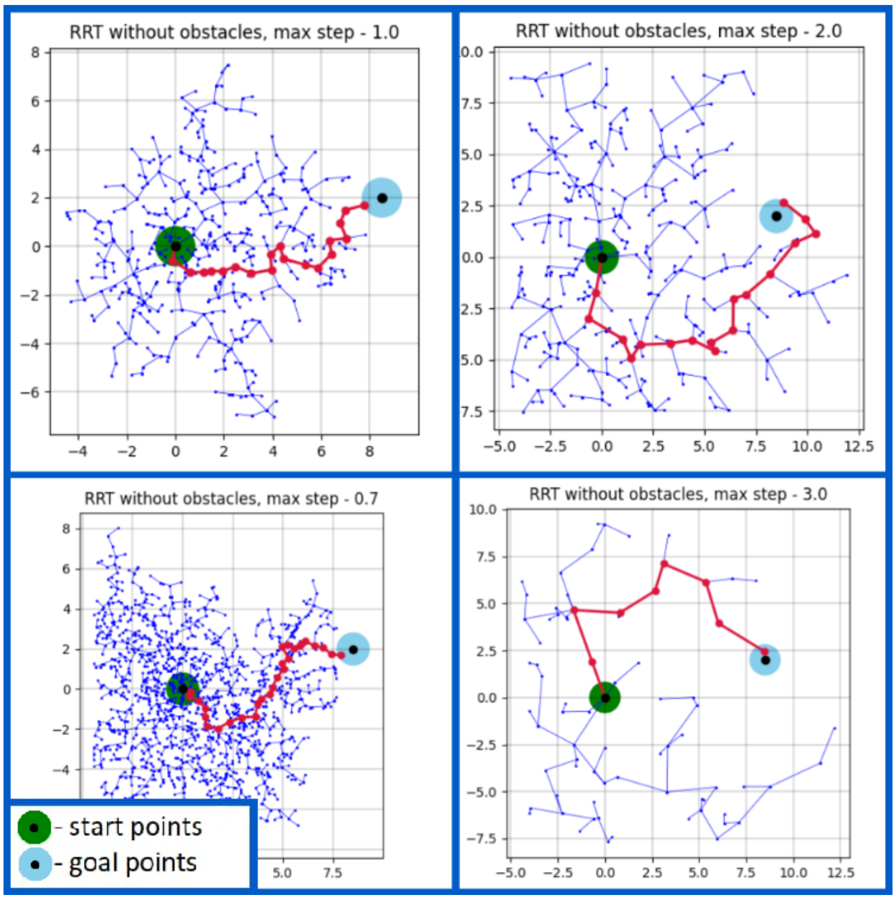


**Figure 5.3:** 2D RRT tree visualized with python and matplotlib.

### 5.3.2 RRT* algorithm implementation

The path search and random tree growth with graph optimisation and rewiring were implemented exactly as shown with pseudocode 3. In contrast to RRT, node costs need to be carried out. Now, when finding the most suitable neighbour node, besides looking at the distance to it, we also need to look at it's cost in case it is not the best one, 4.

**Algorithm 4** Find the best neighbour

```
1: Neighbours ← get_neighbour_in_radius(rnd_point, radius)
2: best_cost_to_new_node ← closest.cost + distance_to_closest;
3: for neighbour ∈ Neigbours do
4:     is_inside_an_obstacle ← false;
5:     current_cost ← neighbour.cost + distance_between_points(rnd_point, neighbour)
6:     if current_cost < best_cost_to_new_node then
7:         for obstacle ∈ obstacles do
8:             if there_is_an_intersection(neighbour, rnd_point, obstacle) then
9:                 is_inside_an_obstacle ← true
10:                Exit_for_cycle
11:            end if
12:            if is_inside_an_obstacle then Continue_for_cycle
13:            end if
14:            best_cost_to_new_node ← current_cost
15:            best_neigbour ← neighbour
16:        end for
17:    end if
18: end for
```

#### Results

Following figures show a couple of simulations without obstacles. Figure 5.4 shows the typical paths found with the RRT* algorithm. The path is almost a straight line, despite the fact that the number of iterations was set fairly low ($< 50$).

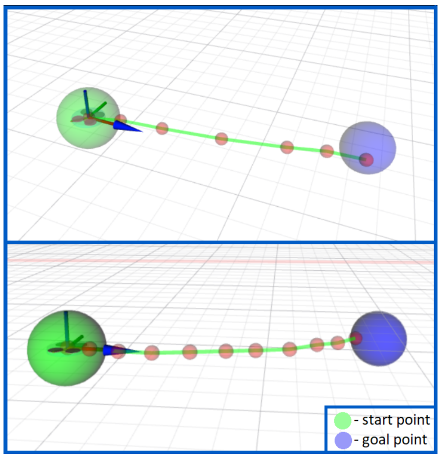


**Figure 5.4:** 3D RRT* found path without obstacles and appropriately tuned parameters, Rviz simulation.

RRT* is strongly dependent on several important parameters, and for simplicity, I will compare the results with the following (most crucial) varying criteria:

- Minimal number of iterations $N_{iters}$.
- Neighbour radius for graph rewiring $R_n$,
- Maximal distance between nodes $D_{max}$.

First, let us look at the changes made by **varying the minimal number of iterations** $N_{iters}$, Figure 5.5. As a reminder, RRT* continues to optimise the path even after it was found for a given number of iterations, unlike RRT, which stops right away. As we can see from the simulations in Figure 5.5, the path is being improved with time. But after 1000 iterations, it is almost a perfectly straight line, which means that it is not really necessary to continue. When the number of iterations is set too low (upper plots), the algorithm does not finish the optimisation. The path is not that bad, but it can definitely be better.

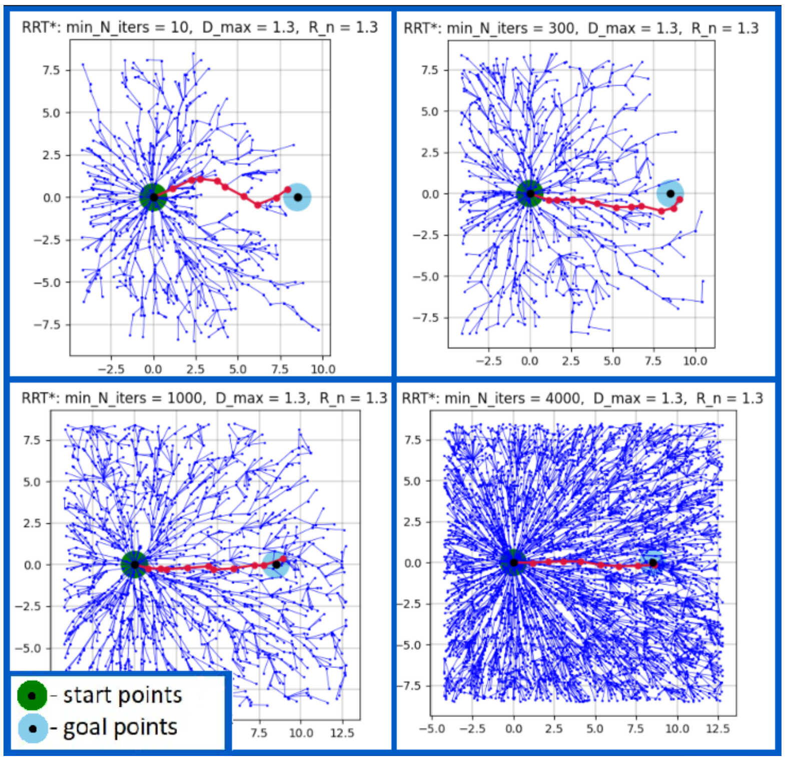


**Figure 5.5:** 2D RRT* graph with different minimal number of iterations.

The changes made by **varying the neighbour rewiring radius** $R_n$, Figure 5.6, clearly show the importance of finding the right parameters. When set too low, the tree looks more like a tree produced by the RRT algorithm (top left). That is because the algorithm only rewires the graph in a small radius around a point, which is not enough for big improvements. Similarly to that, there is a situation where the neighbour radius is set too large. However, while the final path looks great and straight, the computational time to optimise all points in a big radius can be too long for certain applications.

Lastly, **changing the maximal distance between nodes** $D_{max}$, Figure 5.7, also controls the algorithm result to performance ratio. The smaller the distance, the more precise the result will be. But the number of calculations is much higher when using a bigger distance. Interesting observation - when distance is bigger than neighbour radius, the RRT* produces a weird firework-shaped graph, which, of course, cannot be profitable for regular path-finding.

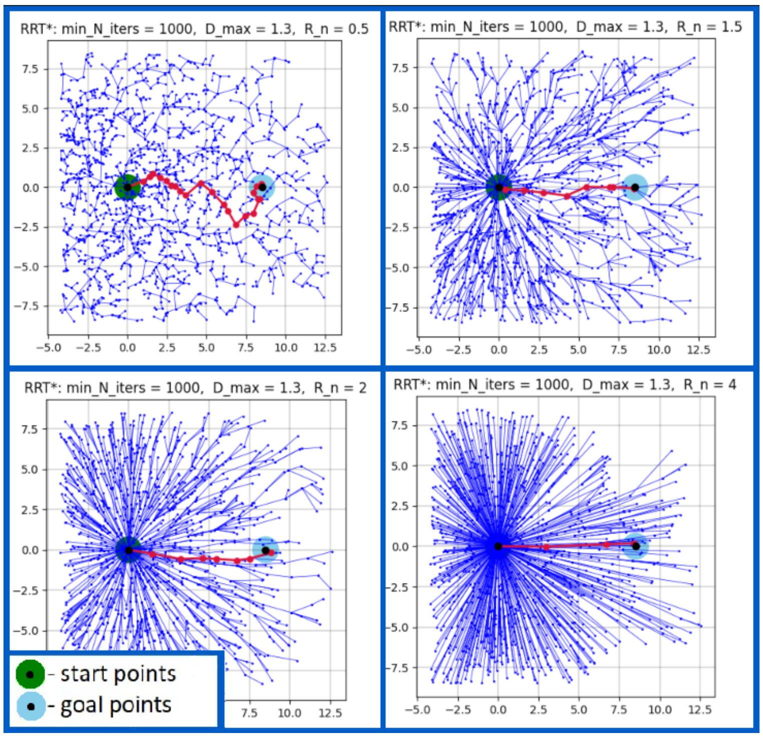


**Figure 5.6:** 2D RRT* graph with different neighbour radius for optimisation rewiring.

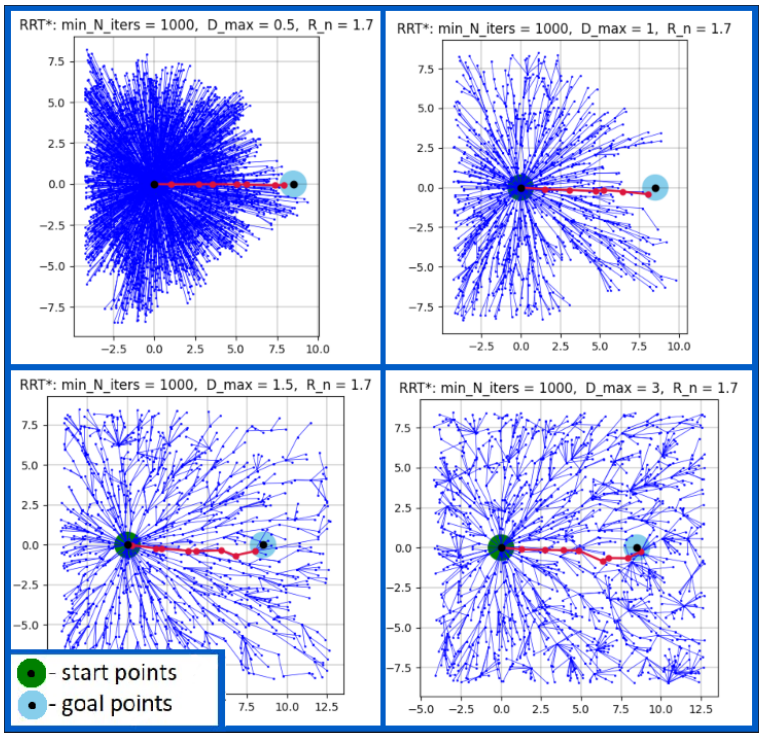


**Figure 5.7:** 2D RRT* graph with different maximal distance between nodes.

# Chapter 6
# UAV flight with Obstacle Avoidance

Obstacle avoidance can be performed in various ways. In this project, I implemented two of them. Like the *Algorithm* class that has a virtual function that is redefined in the *RRTAlgorithm* and *RRTStarAlgorithm* classes, here is a *AvoidanceAlgorithm* class with a virtual function *ThereIsIntersectionAlongThePath()* that I redefine in the *LinearAlgebraIntersection* and *BinarySearchIntersection* classes.

## 6.1 Point drone - inflated/virtual obstacles, line/sphere intersects

This approach assumes that a drone is a single point. However, all the obstacles that I have represented for now as spheres have an artificially enlarged radius. In other words, a considerable amount of a safe zone is glued to the obstacles, so that the point drone can safely be in any place outside those zones without interacting with the objects.

This way of obstacle avoidance requires two conditions to be satisfied to put a new trajectory point to the tree:

**1).** New point doesn't lay inside an obstacle,

**2).** The line segment between two points does not intersect any obstacle (the line between the place of connecting to the tree $S_i$ and a new node $S_n$).

In this method, considering the obstacles of the sphere, **first condition** can easily be judged from the distance between the obstacle centre $S_o$ with radius $R$ and the new point:

$$dist(S_n, S_o) < R. \tag{6.1}$$

**Second condition** requires a little more complicated maths. It is implemented in the *LinearAlgebraIntersection* class and the idea was insiped by [31]. Let us assume that we have two points $S_1$, $S_2$ and we want to check if the line segment between them intersects a certain sphere with a coordinate vector $\mathbf{S}$ with radius $r$. First, we need to find the point closest to $\mathbf{S}$ that is on the line. Let us find the difference of $S_1$ and $S_2$,

$$\mathbf{d} = \mathbf{S_2} - \mathbf{S_1}. \tag{6.2}$$

Then find the squared length of this segment, in other words, the vectors norm squared,

$$l = \mathbf{d}^T\mathbf{d}. \tag{6.3}$$

Find the second needed vector, the vector between $S_1$ and $S$,

$$\mathbf{h} = \mathbf{S_1} - \mathbf{S}. \tag{6.4}$$

Do the dot product of $d$ and $h$. This, divided by $l$ will give us the percentage along the $S_1$-$S_2$ line - $P$. If the found number is not inside the [0, 1] interval, then make it 0 or 1.

$$P = \frac{\mathbf{d} \cdot \mathbf{h}}{l}, \tag{6.5}$$

$$P = \begin{cases} 0, & \text{if } P < 0 \\ 1, & \text{if } P > 1 \end{cases}. \tag{6.6}$$

The closest point to the centre of the sphere **C** can be calculated as follows:

$$C = \mathbf{S_1} + P \cdot \mathbf{d}. \tag{6.7}$$

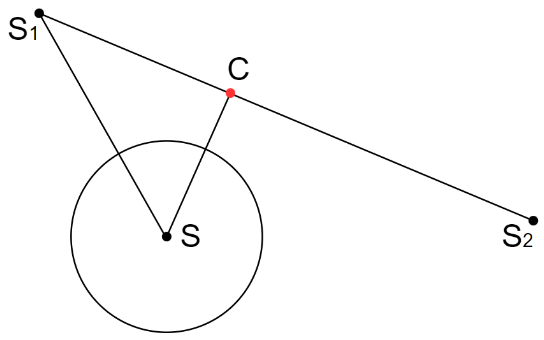


**Figure 6.1:** 2D example of line segment with circle intersection, **C** isn't inside of a sphere

Now that we have **C**, all that remains is to check whether it is inside this sphere, which is almost trivial (first condition of this avoidance approach).

### 6.1.1 Results

This avoidance algorithm was implemented and tested successfully. Figure 6.2 shows RRT and RRT* working in two different 2D environments. RRT finds a solution with a very small number of iterations, but RRT* tends to have much better optimised trajectories.

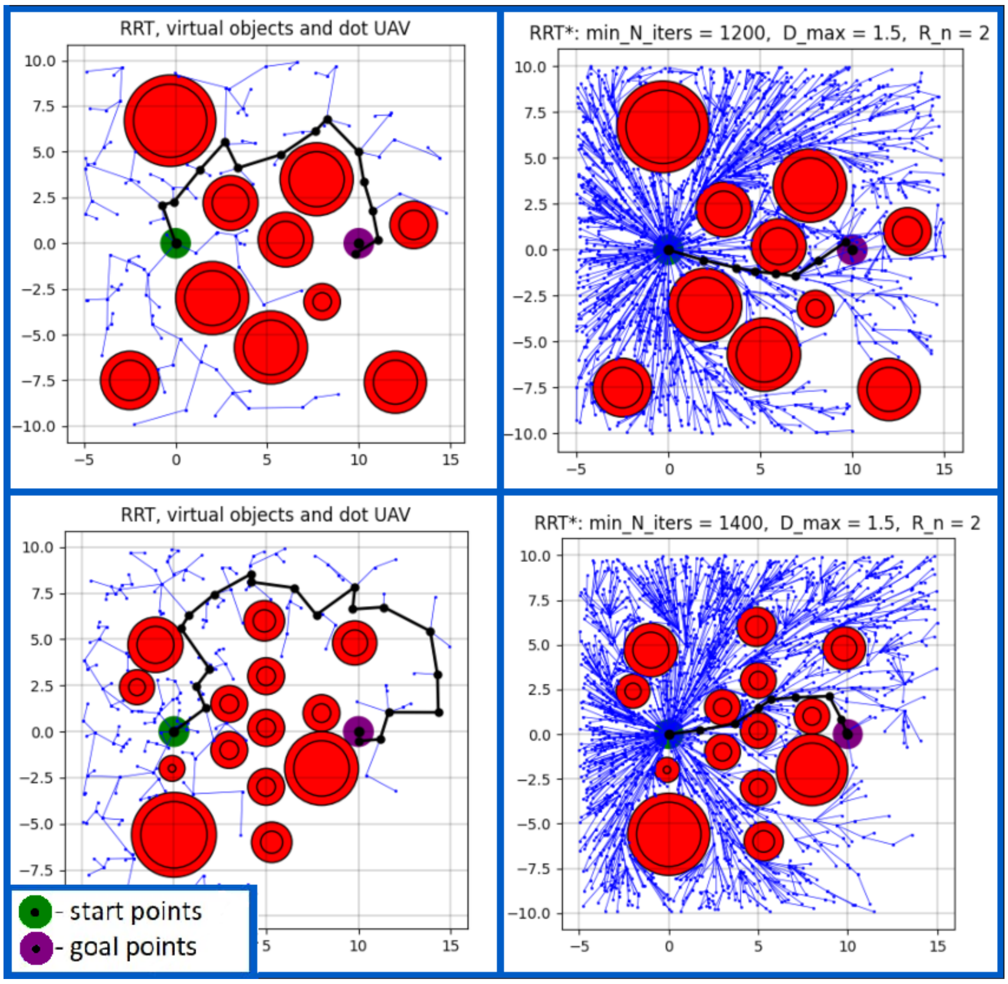


**Figure 6.2:** 2D simulation of inflated objects and point UAV obstacle avoidance.

## 6.2 Sphere drone - binary search of collisions

In this approach, it is assumed that the drone is a sphere of appropriate radius. This means that no safety indent needs to be added to the obstacles, unlike the case with a point drone. First, we need to have a function to check for two-sphere intersections (in project, located in the AvoidanceAlgorithm class). For an intersection, the following condition must be satisfied:

$$dist(c_1, c_2) \leq R_1 + R_2, \tag{6.8}$$

where $c_1$ and $c_2$ are the sphere centres and $R_1$, $R_2$ are their radii.
The collision search algorithm is as follows:

1. Define a line between start $S1$ and goal $S2$ points.
2. Find the centre of this line.
3. Assume that the UAV is at that central point and look for an intersection with the obstacle $O$.
4. If the intersection has not been detected, select a new line segment. Line segment between the centre point and the end point of the previous line that is closer to the obstacle $O$ centre.
   If there is an intersection, then the algorithm stops, the result is found.
5. Continue from Step 2.

This is an iterative method and the depth of the collision search should be defined. It can be defined as a number of steps to take or as a minimum step length. Minimal step length is a more robust way because of situations where the search line is too big and a fixed number of steps will not control all the needed points.

As a 2D example, consider the obstacle $O$ and two points $S_1$ and $S_2$, Figure 6.3. The first step is to find the middle point between $S_1$ and $S_2$ (I). Then check if there is a collision, assuming that a UAV is in position (II). No, there is no collision; define the next line segment between that centre and $S_2$. Check for an intersection at middle point 2 (III). Continue in the same way and finally find a collision after 3 iterations (IV).

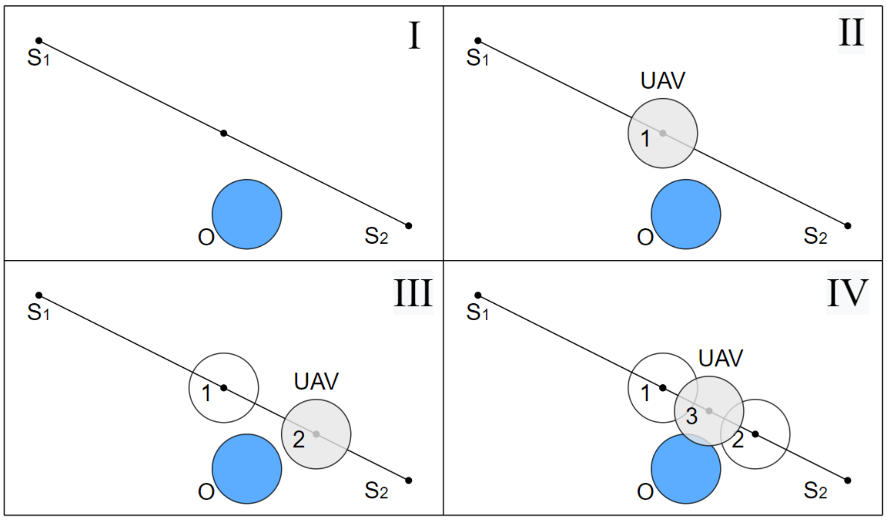


**Figure 6.3:** 2D example of binary collision search.

### 6.2.1 Results

3D simulation of binary search avoidance is shown in Figure 6.4. Figure 6.5 represents RRT and RRT* working in two different environments. From the gap around objects, we can see how the drone will not approach them too closely. The lower row of the pictures shows that RRT* is capable of finding a way even through a very narrow bottleneck-shaped place. Also, the fan-shaped twigs of the RRT* tree are perfectly seen compared to a kind of rectangular RRT.

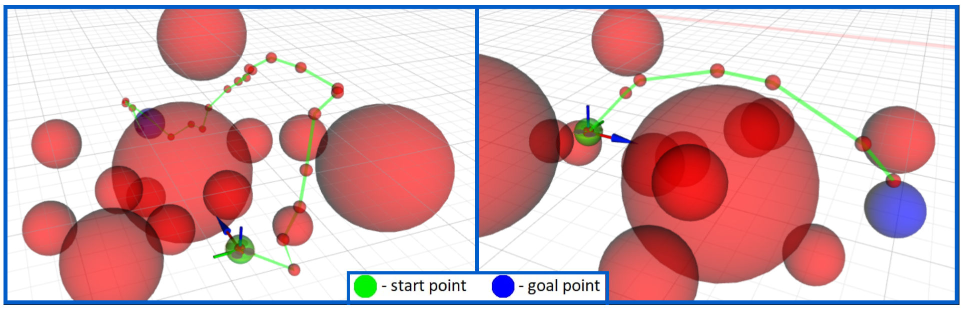


**Figure 6.4:** 3D simulation of spherical UAV and binary search avoidance in Rviz with RRT and RRT*.

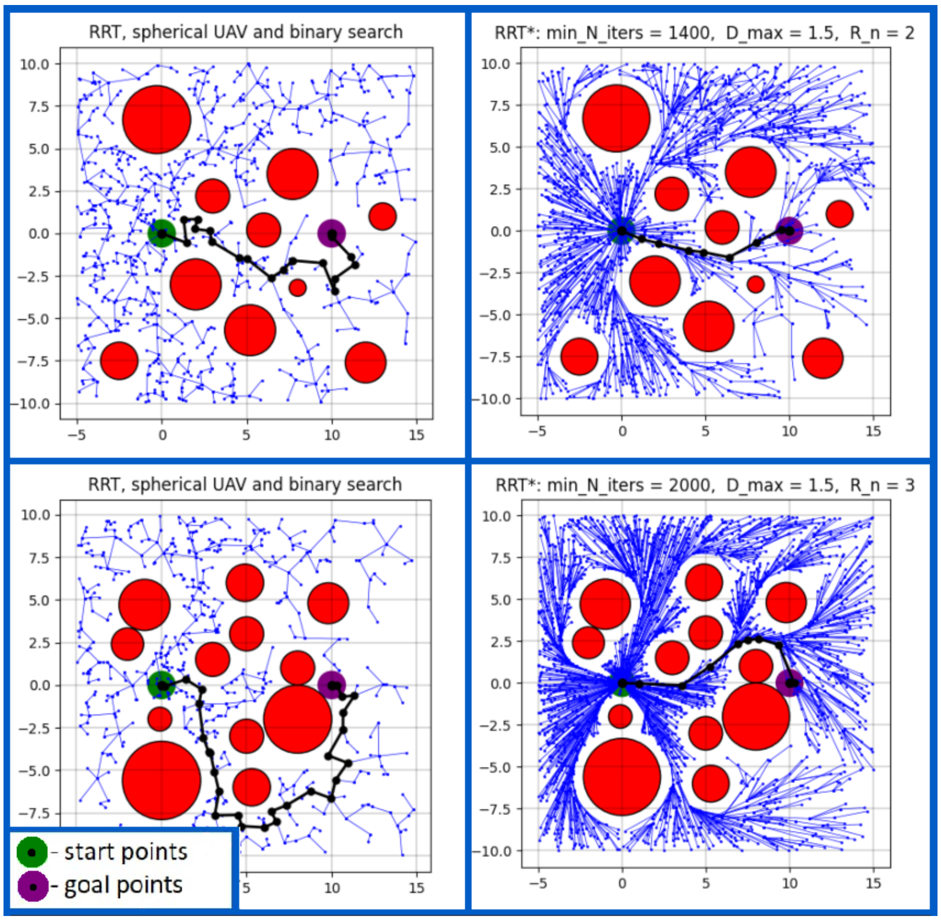


**Figure 6.5:** 2D simulation of spherical UAV and binary search obstacle avoidance.

### Straight line solution

Of course, when there are no obstacles along the straight line between a current node and a target, this line can be added to the tree as the best trajectory possible. This step helps RRT reduce the number of nodes added even further, Figure 6.6.

However, for the purpose of this project, I decided to avoid using this feature, because the main goal is to see the exact RRT and RRT* behaviour, without interrogating. Moreover, in a complex environment, such as a forest, this may take more computational time (after adding every node, check for any intersections with any obstacle).

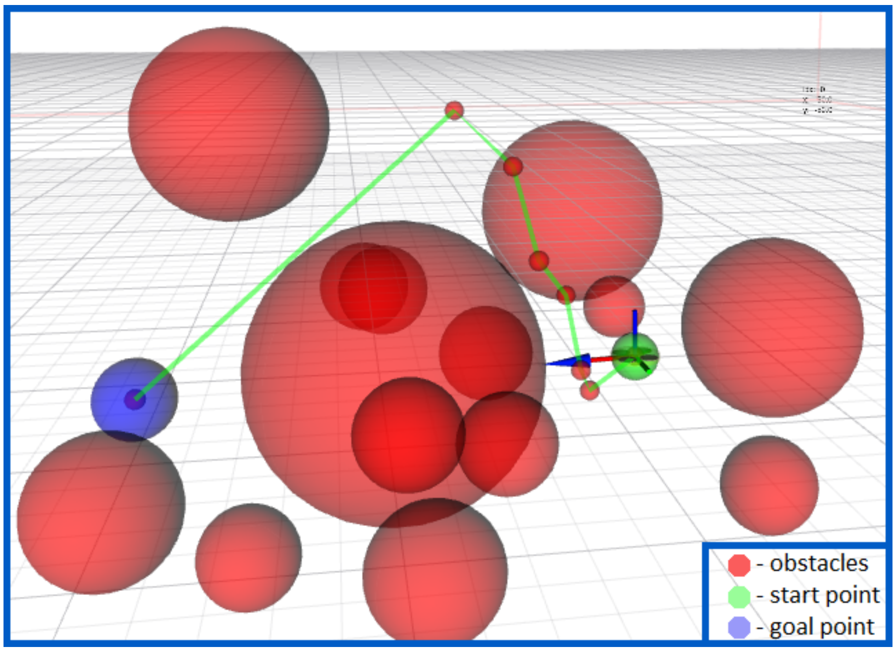


**Figure 6.6:** Straight line solution example.

## 6.3 Avoidance of cylindrical obstacles

Sometimes, it would not be efficient to represent certain obstacles as bare spheres. For example, in a simple forest environment, where trees are essentially the only obstacles to deal with. Trees are mostly in the shape of a vertical barrel. So, I also made the decision to introduce cylinders as an obstacle type in this project. The essence of avoidance algorithms will remain similar to spheres.

The **Point drone - inflated/virtual obstacles** concept was modified to separate two types of objects with a *dynamic_cast* operator. The calculations were reorganised in such a way that the method would work with cylinders. It was achieved by transforming the problem into a 2D plane, where the cylinder is mapped to a circle and the line section is projected to the plane. First, the algorithm ensures that the line segment does not intersect a circle 6.1. If it does not indeed cross it or lay inside of it, then it is known that the line would not intersect the cylinder, and the algorithm returns the answer. However, if it does, on this occasion, we still cannot know the answer; the problem is shown in Fig. 6.7.

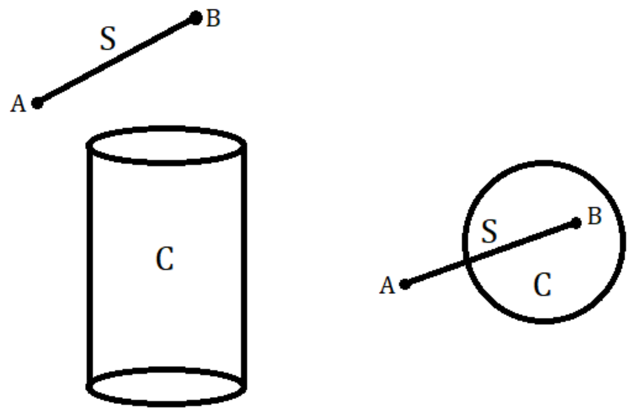


**Figure 6.7:** Explanation of the intersection problem.

This problem was solved transparently by investigating the end points of a line segment: $A$ and $B$. If both points are placed above or both points lie below the cylinder top point, then the intersection is not present. Despite the simplicity of this rule, the algorithm presented itself as very robust. In fact, this made UAV flying even safer, because the drone would be slightly further from the corners of the cylinders. The **Sphere drone - binary search of collisions** way of avoiding obstacles now also differentiates between two types. In this case, the dodging of the cylinders is performed by finding the closest point $P$ on the axis of the cylinder $O$ to the line that we are investigating $A - B$. The algorithm then executes the sphere-to-sphere intersection at that point, Fig. 6.8.

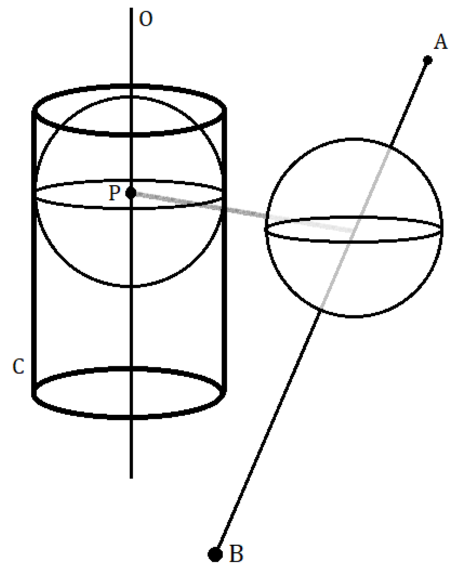


**Figure 6.8:** Explanation of the intersection with a cylinder search.

In case the intersection is not found, the algorithm returns the result. But if the intersection seems to occur, at this point it does not tell us anything. The first thing the algorithm does after that is controlling the location of a $P$ point. The problem here is that it is not possible to approximate the cylinder with a sphere at its top or bottom point. The code executes control of two boundaries above and below the cylinder keeping in mind the radius of the UAV and the "rectangular" shape of the cylinder.

### 6.3.1 Results

Both avoidance algorithms were modified to work with barrel-shaped objects. The RRT* is used in every picture in this results section, so that a man could properly see the avoidance working.

In Fig. 6.9, it can be seen that the algorithms perform impeccably with inordinately big cylinders. This test allows us to see that there are no mistakes when planning above the cylinder. It could be crucial to change the parameters of RRTs, i.e., the maximal step length.

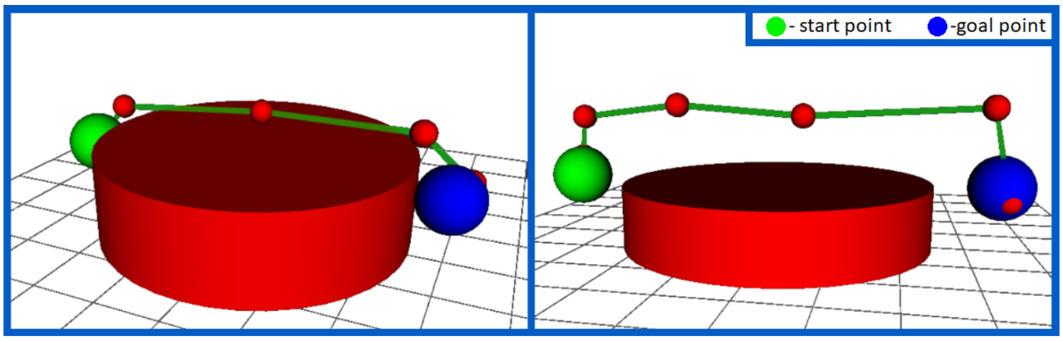


**Figure 6.9:** Wide single cylinder avoidance: left - point drone, right - binary search algorithm.

Fig. 6.10 shows how both avoidance algorithms keep their fundamental properties with cylindrical obstacles. The binary search approach keeps all the necessary distances for the UAV to hover around the obstacle without the danger of collision.

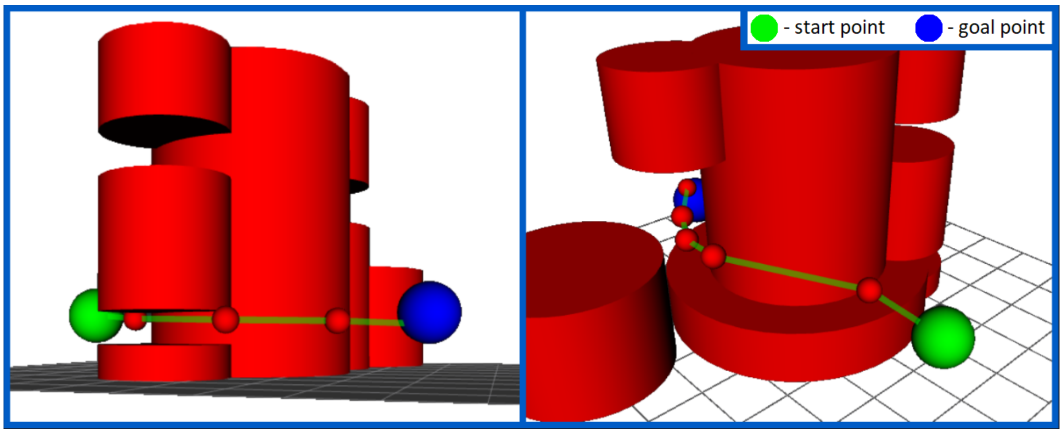


**Figure 6.10:** Cylinder arrangement testing: left - point drone, right - binary search algorithm.

# Chapter 7

# Statistical analysis of RRT*, RRT and obstacle avoidance algorithms

Numerical analysis is an essential part of project implementation. It provides a better understanding of the program and the details of its efficiency. As a first experiment to obtain algorithms' performance data, I created a script to randomly generate a given number of obstacles in a certain area, Fig. 7.1.

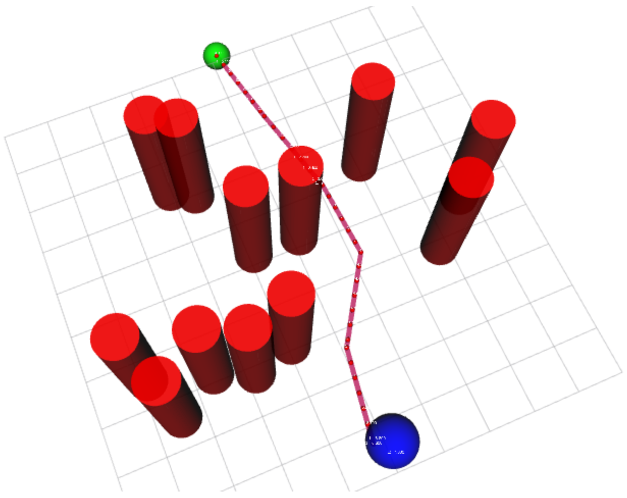


**Figure 7.1:** Example of 12 randomly generated obstacles.

Then the start and goal positions were set to be exactly 10 metres apart. These positions will not change throughout the experiments. After that, there is a for cycle in the script, which will run the provided path planning algorithms $N$ number of times. The data to be gathered from $N$ iterations are:

- Number of nodes in a search tree - $\kappa$,
- Total found path length - $L$,
- Execution time of a planning algorithm - $\tau$.

The collected data will be used to calculate the arithmetic mean $\mu$ and sample standard deviation $\sigma$,

$$\mu = \frac{\sum_{n=1}^{N} x_i}{N}, \tag{7.1}$$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}(x_i-\mu)^2}. \tag{7.2}$$

| **RRT* algorithm** | | | **RRT algortihm** | | |
|---|---|---|---|---|---|
| $\kappa$ [-] | $L$ [m] | $\tau$ [ns] | $\kappa$ [-] | $L$ [m] | $\tau$ [ns] |
| 4617 | 10.7688 | 412.381 | 2363 | 30.4182 | 62.2736 |
| 5612 | 10.4753 | 593.579 | 823 | 13.9266 | 24.4124 |
| 1446 | 10.979 | 72.850 | 1075 | 14.1057 | 32.6869 |
| 2509 | 11.5304 | 154.439 | 1709 | 20.5328 | 41.8258 |
| 762 | 10.497 | 32.801 | 2191 | 26.8105 | 59.656 |
| 601 | 11.5318 | 27.4014 | 2523 | 14.7748 | 65.6948 |
| 1368 | 13.631 | 65.322 | 2042 | 20.16 | 49.4019 |
| 697 | 11.472 | 30.059 | 1182 | 18.9987 | 37.4006 |
| 2483 | 10.6871 | 156.736 | 2441 | 21.6002 | 62.5806 |
| 6399 | 10.5717 | 782.126 | 1327 | 26.7231 | 34.6009 |
| ... | | | | | |

**Table 7.1:** Measured values of the first 10 iterations in a randomly generated environment.

Table 7.1 shows examples of $\kappa$, $L$ and $\tau$ measured on both algorithms. The obstacle avoidance algorithm that was used for this experiment was binary search of collisions, because it allows a more transparent view on the difficulty of the environment with randomly generated obstacles (Fig. 7.1).

| | **RRT* algorithm** | | | **RRT algortihm** | | |
|---|---|---|---|---|---|---|
| | $\kappa$ [-] | $L$ [m] | $\tau$ [ms] | $\kappa$ [-] | $L$ [m] | $\tau$ [ms] |
| $\mu$ | 2439.99 | 11.1201 | 185.265 | 2784.33 | 22.0501 | 87.2244 |
| $\sigma$ | 1524.832 | 0.716886 | 186.816 | 1654.965 | 5.32939 | 75.3477 |

**Table 7.2:** Calculated arithmetic mean $\mu$ and sample standard deviation $\sigma$ in RRT tests.

Table 7.2 contains calculated $\mu$ and $\sigma$ from 100 generated data samples. Looking at the arithmetic mean $\mu$, the reader can observe that the mean number of tree nodes $\kappa$ is quite similar for both RRT* and RRT. It was intentional because the minimum number of nodes for RRT * was set to 0, making it end after finding the first valid trajectory, just like RRT.

More interestingly, average path length $L$ of RRT is twice larger than RRT *, making the flight twice less efficient in the sense of distance. However, on the other hand, the average time required for RRT to find that longer path is more than twice as long as RRT *, which also confirms that the algorithms were designed correctly.

When analysing the sample standard deviation $\sigma$, the path length $L$ stands out significantly. $\sigma$ is a measure of the level of dispersion of the data with respect to the mean. The low standard deviation means that the data are clustered around the mean, and the high standard deviation indicates that the data are more spread out. Even when obstacles are generated randomly, which means that they can be spawned even in a wall-like structure, RRT * manages to have $\sigma$ to be 0.716886 metres. Taking into account the Euclidean distance between start and finish (10 m), the standard deviation of a metre of distance is $0.716886/10 = 0.0716886m = 7.16886cm$ for this scenario. And assuming the normal distribution and using the $68 - 95 - 99.7$ rule [32], in approximately 95% of the path planning in this obstacle density, the algorithm will find a trajectory shorter than $\mu + 2 \cdot \sigma = 12.553872$ metres.

The RRT algorithm, however, has much bigger $\sigma$, which means that the found path usually oscillates greatly, between 27.379 and 16.72 metres with only 0.68 probability.

To compare the speed between two approaches to avoid obstacles, the program was run two times: once with a point drone and obstacles with a radius increased by 0.3 m and second with a drone of radius 0.3 m and obstacles of real size. As the parameters $\kappa$ and $L$ will not be affected, $\tau$ is a single variable to be measured. Again, $N = 100$ iterations with randomly generated obstacles provide the following.

| | **RRT* algorithm** | |
|---|---|---|
| | **Binary search $\tau$ [ms]** | **Point drone $\tau$ [ms]** |
| $\mu$ | 243.961 | 181.99 |
| $\sigma$ | 350.323 | 323.487 |

**Table 7.3:** Calculated arithmetic mean $\mu$ and sample standard deviation $\sigma$ for obstacle avoidance execution time tests.

Table 7.3 shows the calculated $\mu$ and $\sigma$, which tell us that the point drone-inflated object algorithm works faster than the binary search for collisions. But the difference is not that significant, considering that, for precision, the number of steps for the binary search is set to 8, which can be reduced to 4, 5, or 6, when dealing with small maximal distance between nodes.

Sample standard deviation shows, that the time $\tau$ is greatly influenced by the complexity of the generated environment, due to the fact that $\sigma$ is larger than the average execution time.

# Chapter 8
# Multiple UAV path planning

The concluding implementation goal of this work is to design an algorithm that could be used to plan the paths of multiple UAVs. Unlike the case with a single drone, this problem requires the introduction of new terminology. A **path** $P$ is essentially a set of points in a high-dimensional space. For a 3D case, each path $P$ would contain 3D points/vectors $P_i$, where $i = 0, 1, \dots$ . A **trajectory** $T$ is a set of points in space and a schedule for reaching each particular point. Just like that, each trajectory $T$ will contain points and the exact time stamps on which the drone will be flying by this point.

All the planning will take place in a single coordinate system, that is, the main frame of the UAV. Furthermore, the whole calculation process will run on a single drone. This simplifies the problem to focus on trajectories coordination, rather than communicating between different drones to find out all their trajectories just to adjust one's own.

## 8.1 Converting a path to a trajectory

Investigating the project source files, a reader can observe the Trajectory class that was created to work with trajectories. The main attribute of the class is a vector of pairs, representing the trajectory $T$ itself. The method *equally_divide_path_in_time* splits the given path $P$ into segments of a given length $ds$ and assigns a time stamp so that $t_{i+1} = t_i + dt$. It does this by finding the direction vector between each pair of neighbouring points in a path and gradually crawls along it, adding new points to the trajectory, Fig. 8.1. It can be done only with the assumption that the drone will fly with a constant speed; which in our case is assumed. The results can be seen in Fig. 8.2.

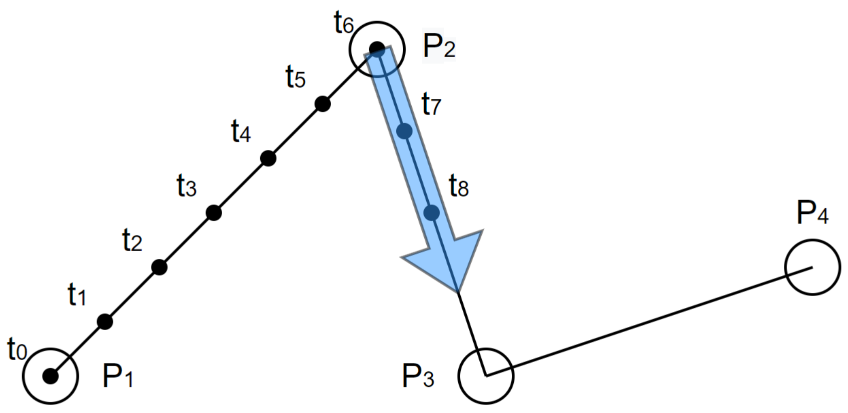


**Figure 8.1:** Path to trajectory conversion visualisation.

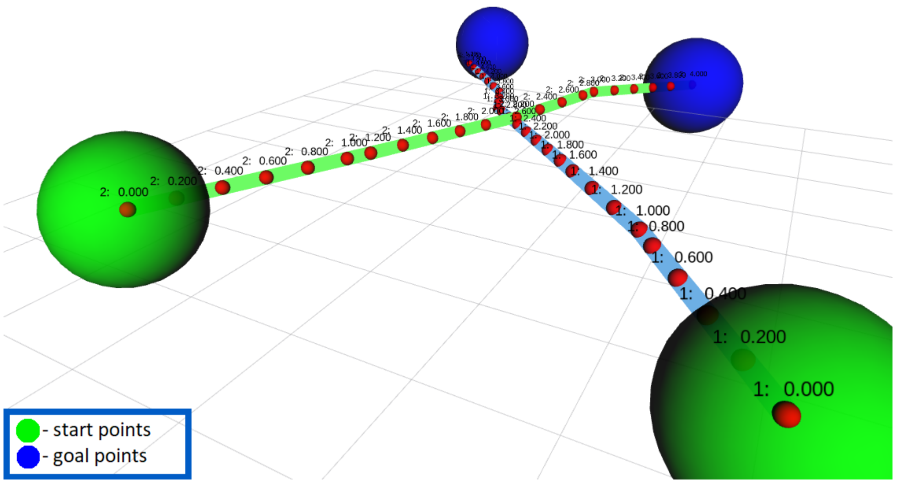


**Figure 8.2:** Time stamps on trajectories in RViz.

## 8.2 Spotting collision dangers

While having $N$ number of trajectories, we need to make sure that there is no pair of UAVs that are going to be in the same place at the same time. Furthermore, for safety, $\xi$ was introduced - time, which can be described as the maximum amount of delay or postponement for a drone to be located in the current position. It serves as an additional time-domain buffer.

Now, to find potential crash areas, two trajectories need to be compared with each other and to see if any points are too close to each other (intersection of two spheres, 6.8) in a time interval $< t_i - \xi, t_i + \xi >$. In addition, the last point that does not intersect $p_i$ of each trajectory is stored for the upcoming construction of a new trajectory. Fig. 8.3 shows the simulation in RViz.

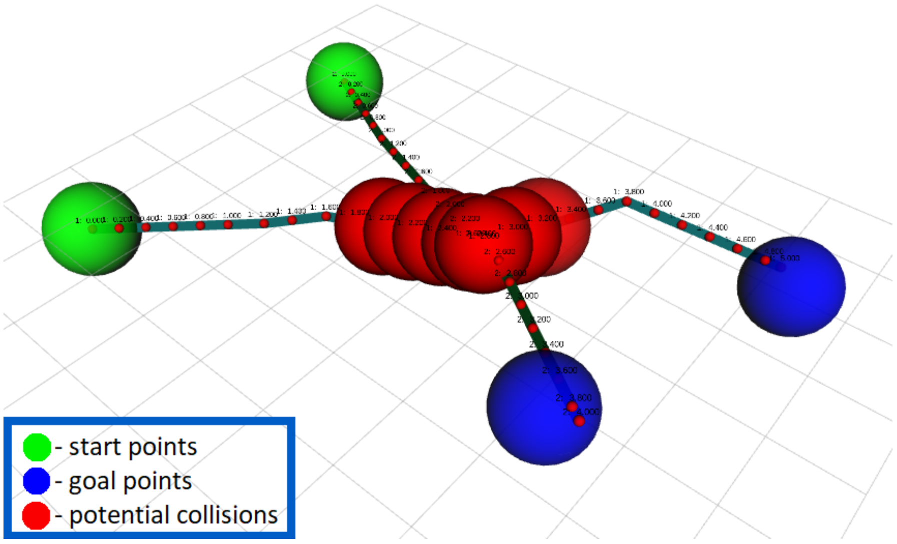


**Figure 8.3:** Potential collision domains between two trajectories.

## 8.3 Constructing not interfering trajectories

Now, it is necessary to build $N$ trajectories, each of them having its own starting-$S_0$ and goal-$S_G$ point. The drones will have their **priority** level. In the event of an arguable situation, a drone with higher priority will have a shorter and smoother path than another. The higher-priority drones will have their trajectories found before generating trajectories for less prioritised drones. The algorithm starts from the highest-priority drone and searches a path without any other drones' constraints, except their starting and goal positions. When the path is found, the second drone will have its trajectory generated. After that, two trajectories will be investigated for collisions. If collisions were found, the path for a second drone should be rebuilt, keeping in mind the collision areas.

**Algorithm 5** Planning multiple trajectories

**Input: An array of drones, each having its $S_0$ and $S_G$ - uav_array.**
**Output: Each drones trajectory that doesn't intersect other ones.**

```
 1: uav_array.sort_by_priority()
 2: for $uav_i \in$ uav_array do
 3:     while collisions_were_detected do
 4:         if performance_mode then
 5:             $uav_i$.change_starting_point_to_last_without_intersects()
 6:         end if
 7:         $uav_i$.find_path().convert_to_trajectory()
 8:         for $uav_j \in$ uav_array[:$i$] do
 9:             find_trajectories_intersections($uav_i$, $uav_j$)
10:             store_last_point_without_collision()
11:             $uav_i$.add_obstacles_in_places_of_collisions()
12:         end for
13:     end while
14: end for
```

Parameter **performance_mode** is introduced to classify two possible path re-building ways, Fig. 8.4. The first one is to forget about the previously constructed trajectory and find a totally new path with newly added obstacles. The second one (performance mode) will take the last point on a previous trajectory without a collision and start building a tree from there. When found, it connects the found path to the tail of the last one.

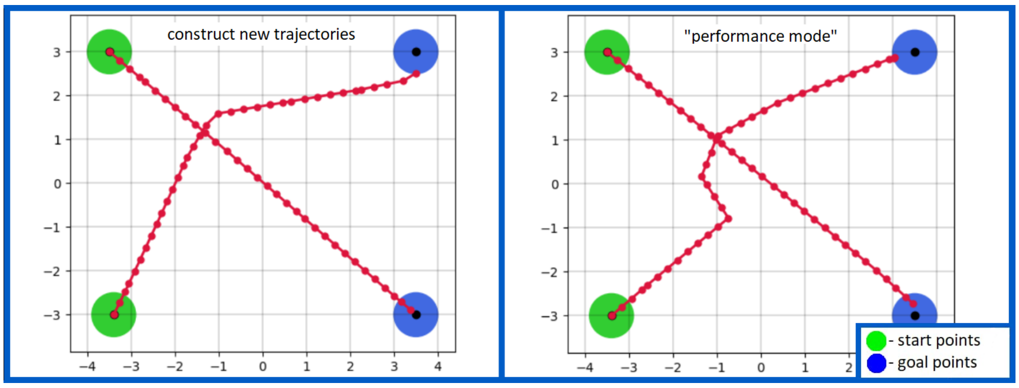


**Figure 8.4:** Difference between the regrowing tree possibilities, right - performance mode.

While using the performance mode, the size of the new search tree can be significantly reduced. Especially when planning in a 3D space, the algorithm will find a short way around a certain collision domain without building a massive tree from the start again. However, the probability that the resulting trajectory will be longer than without the performance mode is high and strongly depends on a given scenario. As for most cases, the difference is not that critical, so the performance mode is more than acceptable for its usage.

## 8.4 Results

Let us have a look at a 2D problem. First of all, in the situation of crossing trajectories, when drones will not collide because of the timings, the algorithm performs as it should, can be seen in Fig. 8.5. One UAV will fly through the crossing faster and there will be no collision.

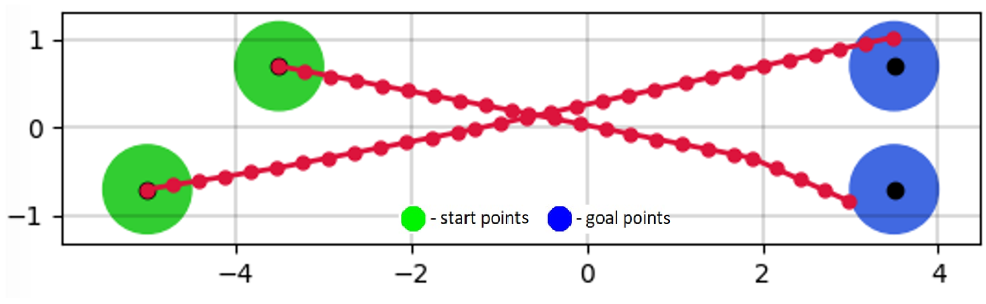


**Figure 8.5:** Time delayed crossing trajectories 2D.

Or more complex examples of the algorithm working with four drones and three obstacles in a 2D plane, see Fig. 8.6.

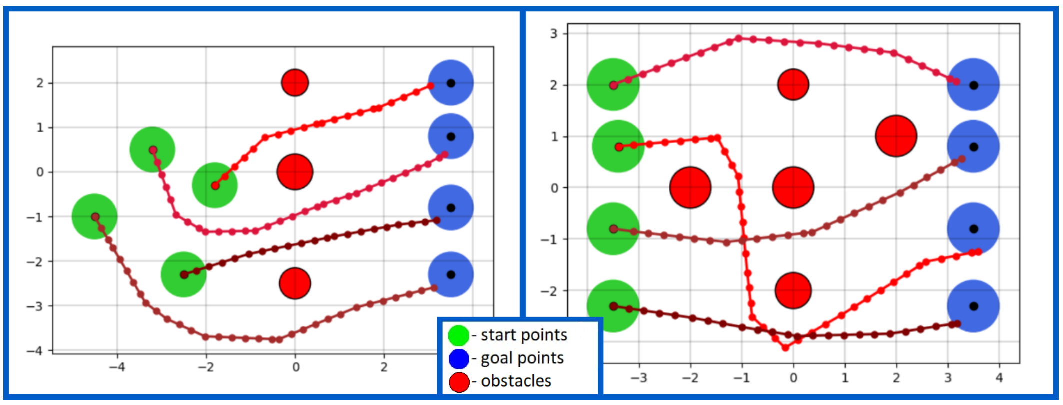


**Figure 8.6:** Time delayed crossing trajectories.

Relocating to the three-dimensional world, the algorithm performs well on two crossing trajectories, Fig. 8.7. When a group of drones and obstacles is introduced, the speed of finding the paths decreases moderately. Even in a dense environment (Fig. 8.8, 8.9), the calculated trajectories will not allow drones to have a collision midflight. In summary, the multiple trajectories generation algorithm works efficiently and reliably.

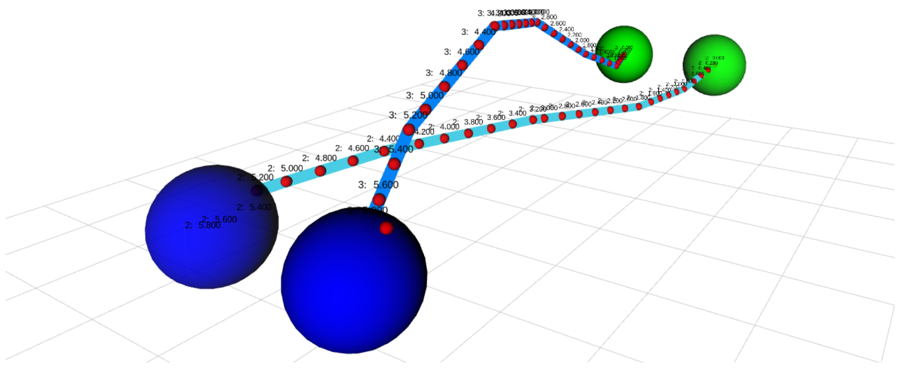


**Figure 8.7:** Two generated trajectories in 3D space.

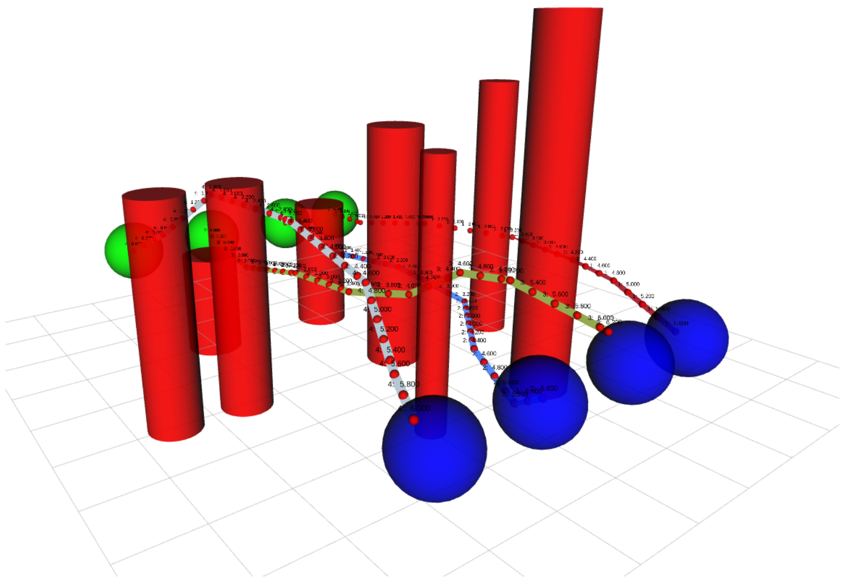


**Figure 8.8:** Example of the algorithm working in 3D, forest like environment.

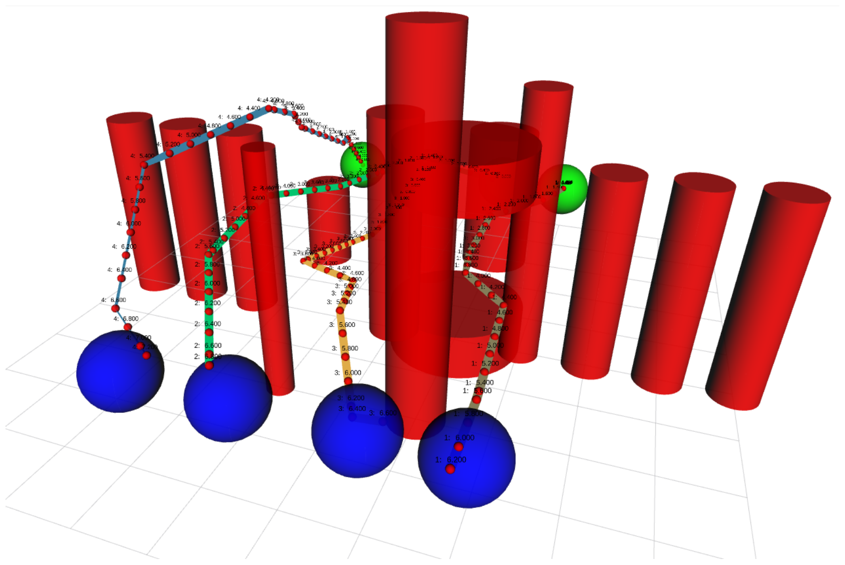


**Figure 8.9:** Example of the algorithm working in 3D with a narrow entrance.

# Chapter 9

# Autonomously flying UAV

To prove the usability of the implemented path planning, it was decided to conduct an experiment with a real UAV. The experiment is meant to be as follows: When giving the drone coordinates in its coordinate system, with the UAV in $(0, 0, 0)$, it must fly to that location, avoiding all obstacles on its way. For instance, giving the drone location $(12, 0, 3)$ must result in it being 12 metres forward in the x-axis direction and 3 metres above the point it was at before.

## 9.1 Sense and avoid principle

For our purposes, the algorithm will use the so-called *S&A - sense and avoid* concept, that is discussed in [33]. The main concept is that an agent/UAV moves step by step with a defined distance; between steps, it detects the obstacles that may cause collisions and replans its trajectory for the next step, Fig. 9.1.

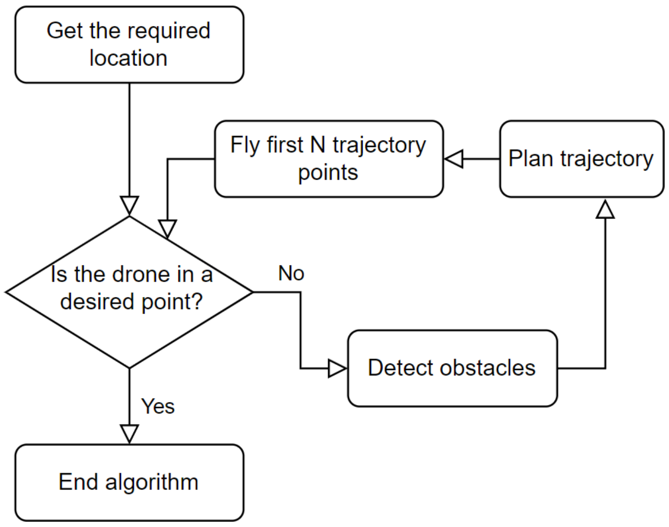

Get the required location
Fly first N trajectory points
Plan trajectory
Is the drone in a desired point?
No
Detect obstacles
Yes
End algorithm

**Figure 9.1:** Sense and avoid algorithm.

This approach allows the experiment to be carried out in safest manner possible due to the different adjustable parameters, such as the time delay between steps or the length of a single step. Also, to be more confident in the safety of an experiment, the RRT * will be used, as it creates much more predictable paths, which is better for the operator holding the emergency stop button. The trajectory waypoints are 0.25 metres apart, and the $N$ number in Fig. 9.1 will be set to 4, so that the step ends to be exactly $0.25 \cdot 4 = 1$m long.

## 9.2 Required hardware

The autonomous flying vehicle needs to have certain components for moving and obstacle detection. All hardware was provided by the MRS group.

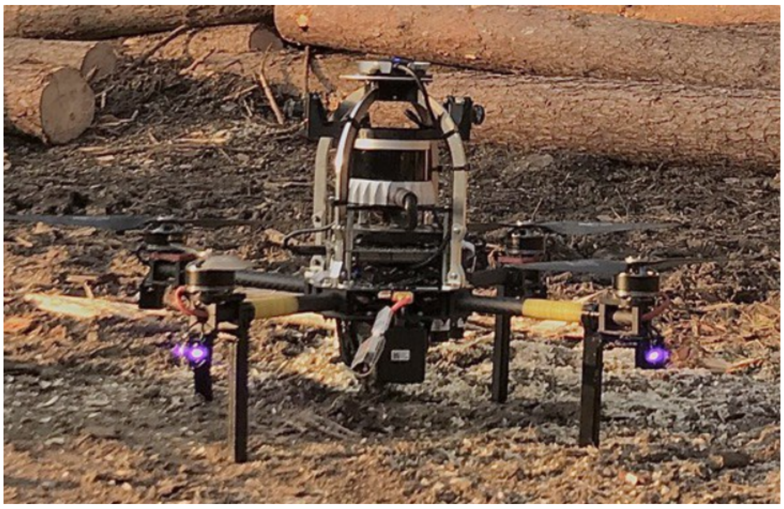


**Figure 9.2:** Fully equipped UAV, that was used during experiments.

- Tarot T650 is the name of the UAV used. Its parameters are as follows.
  - 650 mm frame
  - 15 inch carbon fibre propellers
  - 4x 4114 320kV motors
  - 6S 8000mAh LiPo battery
  - 3.5 kg lift-off weight

  It is a big, redundant drone, with a long-lasting battery.
- GPS receiver. For general purposes, it is necessary to define a safety area in which the UAV will operate.
- 45 degrees Ouster Lidar and metal safety bars around it. Used for reliable obstacle detection.
- Garmin distance sensor. Used for altitude measurements.

## 9.3 Obstacle detection

To avoid obstacles, you must first have an obstacle detection module. For the purpose of this experiment, it was decided to fly in a forest-like environment and perform path planning in a 2D plane. Because cylinder avoidance was already implemented in Section 6.3, trees can be represented as ideal cylinders.
To detect them all around the drone, we need a tool to scan the environment. The ROS node of the MRS bumper, Fig. 9.3, uses a 2D or 3D Lidar sensor and divides the $360°$ plane around the drone into $n$ identical sectors. Then it publishes an array of size $n + 2$ on a *uav_name/bumper/obstacle_sectors* ROS topic. Each element of this array is a float

number representing the distance to the closest point in the current sector. 2 additional items represent the measured height and distance to the ceiling if flying inside.

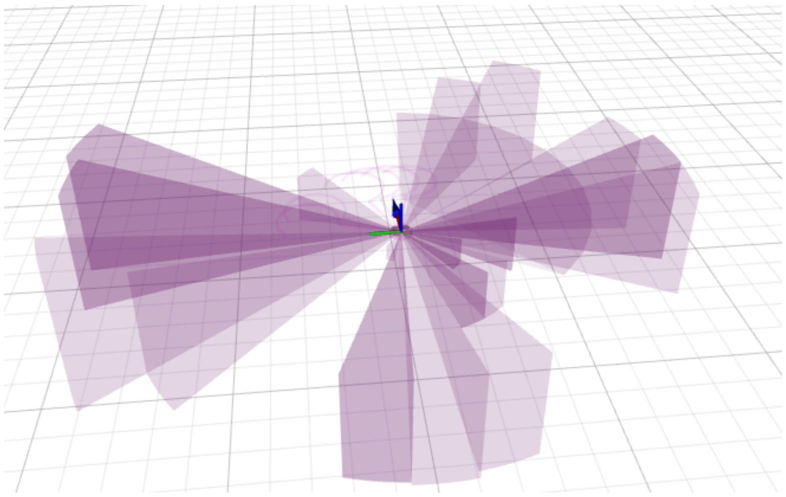


**Figure 9.3:** MRS bumper tool visualised in RViz. Number of sectors $n = 30$.

Using these $n$ sectors, it is possible to map cylindrical obstacles using simple trigonometry. The first sector is aligned with the x-axis of the drone, having an angle of 0°. Other sectors have their angle incremented counterclockwise if viewed from the top. In Fig. 9.4, 4 sectors are shown (I, II, II and IV). Vectors $a, b, c$ represent the distance to the closest point in the corresponding sector. Sector II has a zero vector, which means that the Lidar has not detected any cloud-points in this sector direction. The cylindrical obstacle positions can be obtained from each sector angle $\theta_i$ and distance $d_i$ as such 9.1,

$$x_i = d_i \cdot cos(\frac{\theta_i \cdot \pi}{180}),\ y_i = d_i \cdot sin(\frac{\theta_i \cdot \pi}{180}). \tag{9.1}$$

The radius of the current cylinder is calculated as 9.2. The constant $r_{min}$ is the minimum obstacle radius allowed, which prevents dangerous situations close to obstacles,

$$r_i = min\{r_{min},\ d_i \cdot sin(\frac{\pi}{n})\}. \tag{9.2}$$

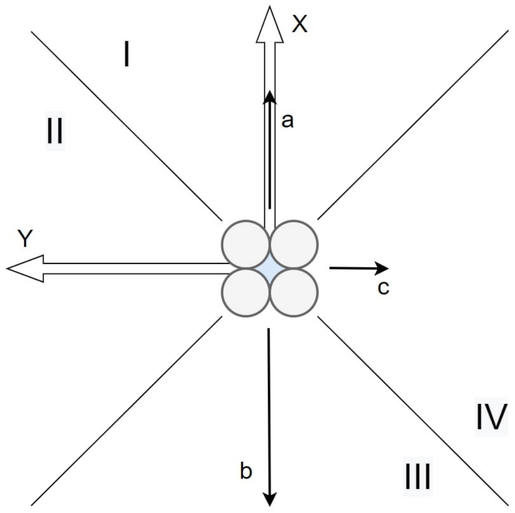


**Figure 9.4:** Diagram explaining MRS bumper principle. View from above.

### 9.3.1 Simulation tests

The resulting tree-to-cylinder mapping was first tested in a simulation. The adjustable minimal obstacle radius allowed to tune the algorithm to be robust and ensure the maximal safety of the UAV. Adding cylinder-type markers to RViz displays a real-time mapping of the trees, Fig. 9.5.

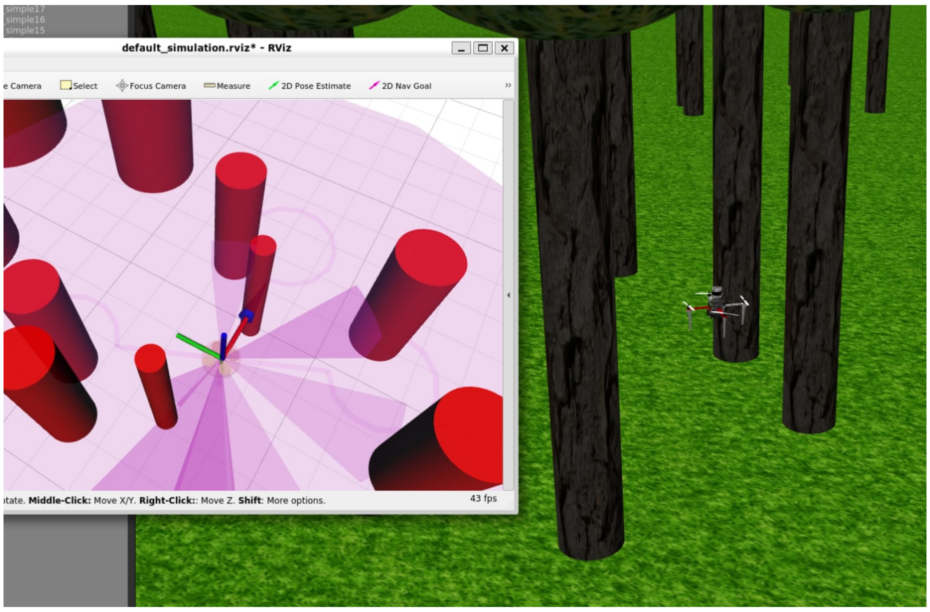

default_simulation.rviz* - RViz
e Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
otate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.
43 fps

**Figure 9.5:** Simulation test of the trees mapping in RViz and Gazebo example.

The fact that the radii of the trees are calculated in such a way that the farther the obstacle, the bigger the radius, introduces a certain amount of uncertainty for the planning algorithm to cope with. But because we only use a small number of bumper sectors $n$, the planner will not control more than $n$ obstacles in every iteration, resulting in a decrease in the execution time of the trajectory planner.

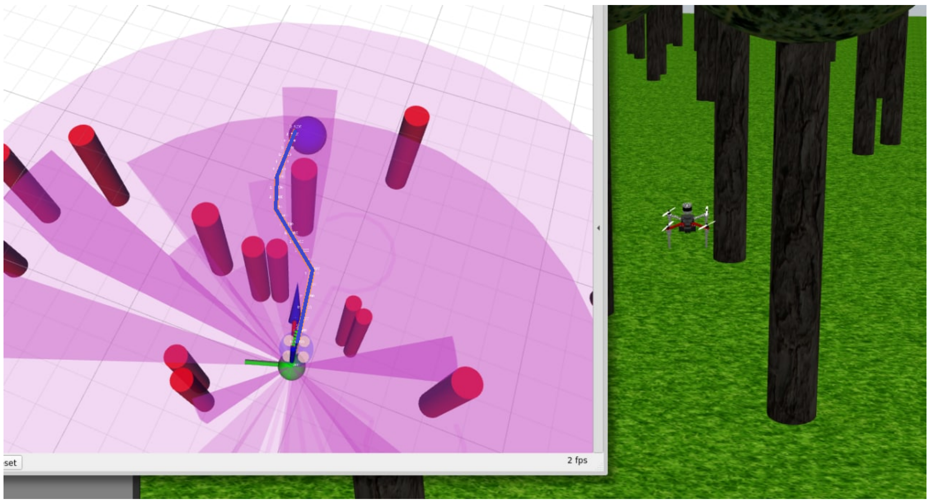


**Figure 9.6:** Simulation test of trajectory planning.

Fig. 9.6 shows the ability of the algorithm to construct a trajectory through the mapped obstacles and follow it. A video of a simulation test is given in **this youtube link**.

# Chapter 10

# Real world experiment

The goal of the real experiment is to demonstrate the usability of implemented algorithms. The task sounds the same as the one sounded in Chapter number 9: autonomously fly through the forest-like environment to a given position relative to the UAV. To carry out the experiment safely, soft artificial tunnel-like obstacles were built in the middle of a field, Fig. 10.1.

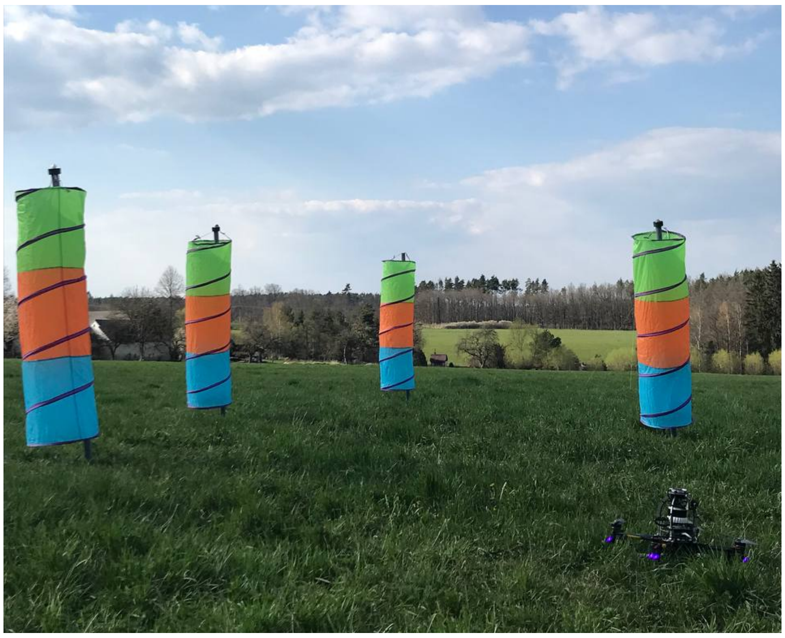


**Figure 10.1:** Artificial experimental forest.

Several tests of flying between tunnels were performed and in this paper only the last one will be described. Many data were recorded during the experiment. ROS feature, called **rosbag**, was used to store all logs and published topics. This allows us to monitor every aspect of the flight at any time after the experiment.

After a successful take-off of the UAV, it is required to give it a command to allow the execution of a program. For this purpose, ROS service */StartExecution* was created (located in the /srv directory of the project).

The coordinates (0, 15, 0) were the input for the experiment, which means that the UAV is wanted to fly 15 metres in the local Y-axis direction avoiding the tunnels. The starting and goal points are shown in Fig. 10.2.

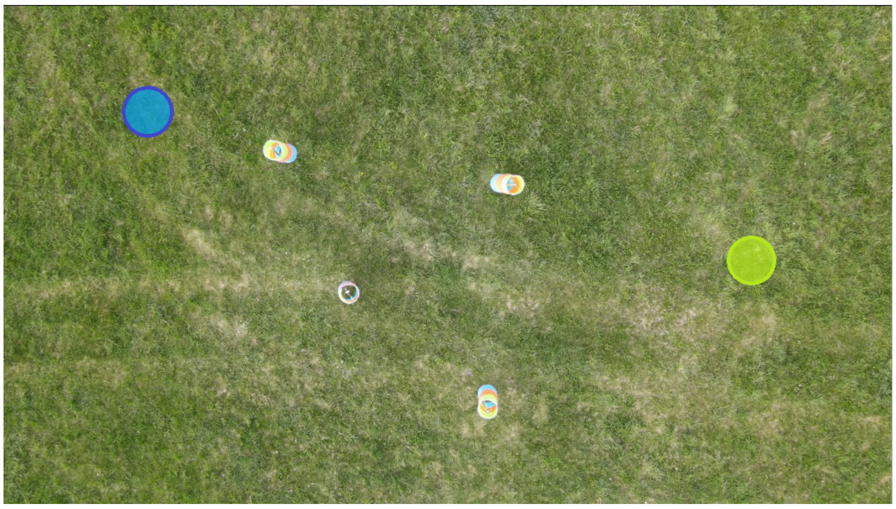


**Figure 10.2:** Starting (green) and goal (blue) positions for the drone.

The altitude of the flight was unchangable (1.5 m); otherwise, the drone could create a trajectory above all the tunnels, because it would be more efficient than flying in a 2D plane.

Flyight visualisation can be recreated using the generated rosbag file, Fig. 10.3. It can be clearly seen that the lidar cloud points and mapped cylinder obstacles around are precisely correlating, so the drone is safe to create trajectories around them. The video of the flight (top view, side view and RViz data) is accessible in **this youtube link**.

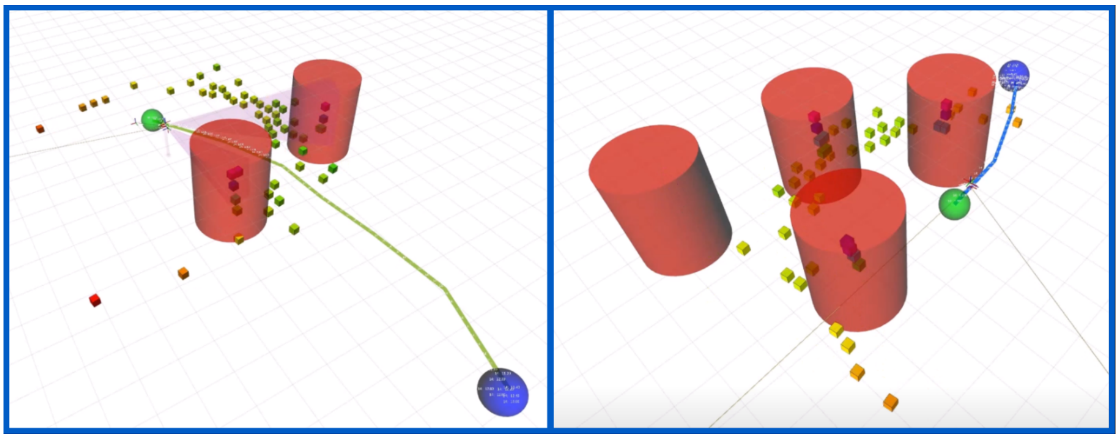


**Figure 10.3:** Rviz visualisation of gathered during the experimental flight data.

To analise the precision of reaching the goal, it is possible to use a global GPS frame. After take-off, the drone is located at $(-57.53, -20.54, 1.52)$ and at the end of the flight its coordinates are $(-67.37, -8.82, 1.45)$. The distance between these two points is 15.3 metres, which, keeping in mind the goal radius of 0.5 metres, is an achieved goal.

From the local coordinates incremented during the flight - from $(0, 0, 0)$ to $(0.11, 14.54, 0)$, it can also be confirmed that the goal was reached correctly. The algorithm stopped its execution when the distance to the goal became 0.46 metres, which is less than the radius of the goal.

It is also possible to access the relative positions of the goal. After each trajectory step, the drone was publishing them on Rvizs subscriber topics. These data were visualised using Matlab, Fig. 10.4. There are a couple of interesting points in this scatter plot.

For instance, the third blue point on the left is slightly shifted from other points on a seemingly straight trajectory. This happened because the UAV replanned its trajectory to the other side of the tunnel because it discovered the fourth tunnel in the distance, which messed with the previous plan of the drone.

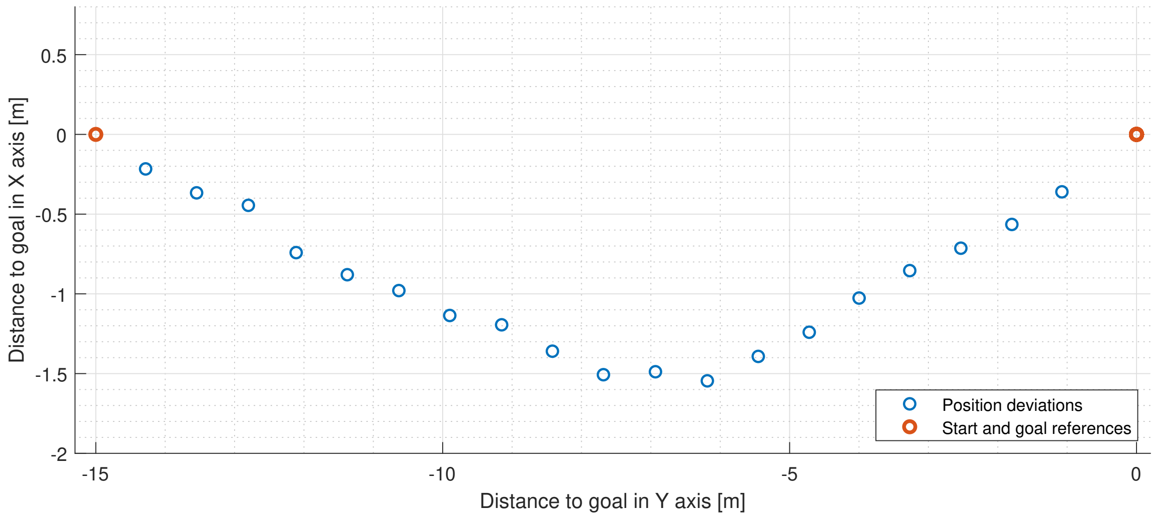


**Figure 10.4:** Distance that was left to overcome after each trajectory step in both axis.

Also, the bumpy character of the trajectory in its maximal X coordinate deviation (lowest points on the graph) tells us an uncertainty about the distance to the closest obstacle. This can be easily explained by the strong wind that blew that day, shaking the tunnels and periodically tilting them for ±20 cm.

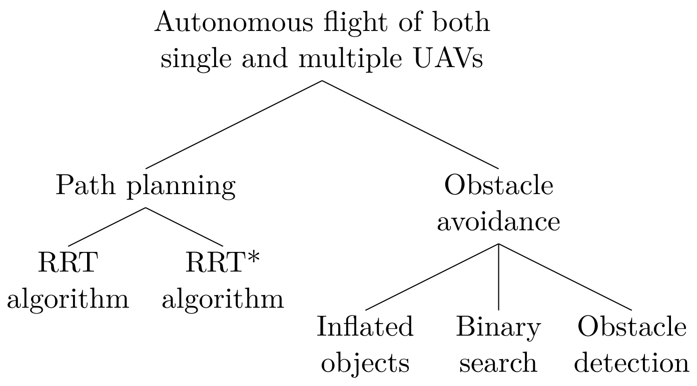
## 10.1 Summary of the experiment

The real world experiment with tunnel obstacles confirmed the possible usability of the algorithms implemented during this project. Algorithms are capable to work not only on ideal simulations with perfectly still trees and zero wind, but also in the environment with imperfections.

# Chapter 11
# Conclusion

Studying the literature and other public sources of information, the author of this thesis obtained basic knowledge on the path and motion planning problem and its aspects, such as obstacle avoidance and possible algorithm optimisation. This knowledge allowed structurising the implementation process in the optimal way, and it can also be used for later contributions to the research of this field.

The whole practical side of this project can be described using a tree graph with each branch implemented, creating a solid foundation for future development:



Due to the fact that the project was starting from scratch, a lot of path-planning algorithms were programmed to test the actual RRT and RRT*. For instance, the obstacle avoidance and obstacle visualisation must have been handled. For this purpose, two completely different approaches were implemented: inflated objects and binary search.

The formal implementation objective of this thesis was to implement and modify standard RRT and RRT* algorithms to consider route planning for multiple robotic agents. The goal was achieved; designed algorithms allow one to plan not intersecting trajectories for a swarm of drones avoiding the obstacles of the provided environment.

Several experiments were carried out both in simulation and in the real world to show the actual usability of the designed path-planning algorithms. But first obstacle detection and mapping needs to be introduced. It was decided to conduct experiments in a forest-like environment, so a lidar sensor served to measure distances to the trees around the UAV. Simulation experiments demonstrated that a single drone is capable of a slow autonomous flight in a moderately dense forest (trees standing 2 metres apart). The real-world flight experiment confirmed the possible usage of implemented path and motion planning even in an imperfect environment with shaking obstacles.

## 11.1 Future work

The design of the project allows us to extend it during future work. Future work can be, i.e., speeding up the drone moving through the forest by applying serious automatic control features or conducting the experiment in the forest-like environment for multiple agents (command each UAV its own trajectory to fly through). Or, of course, implementing other interesting path and motion planning algorithms, for example those described in Chapter 3. Just as important is the ability to introduce programmed algorithms to other projects' pipelines. RRT and RRT* would be a solid extension to a project without them.

# Bibliography

[1] Steven M. LaValle 'Planning algorithms', University of Illinois 2006, p. 269.

[2] Steven M. LaValle 'Planning algorithms', University of Illinois 2006, p. 259.

[3] Roberts Eric, Motion Planning in Robotics course 1998-99, https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html

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

[5] C.L.Nielsen, Lydia E. Kavraki, 'A Two Level Fuzzy PRM for Manipulation Planning', Proceedings of IROS 2000, In Press

[6] Robin Bohlin, Lydia E. Kavraki 'Path Planning Using Lazy PRM', Proceedings od the 2000 IEEE International Conference on Robotics & Automation

[7] Franz Aurenhammer. 1991. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Comput. Surv. 23, 3 (Sept. 1991), 345–405.

[8] S. K. Ghosh and D. M. Mount, "An output-sensitive algorithm for computing visibility graphs", SIAM J. Comput., vol. 20, no. 5, pp. 888-910, 1991.

[9] P. Bhattacharya and M. L. Gavrilova, "Roadmap-Based Path Planning - Using the Voronoi Diagram for a Clearance-Based Shortest Path," in IEEE Robotics & Automation Magazine, vol. 15, no. 2, pp. 58-66, June 2008.

[10] Canny, John F.. 'A Voronoi method for the piano move problem.' Proceedings. 1985 IEEE International Conference on Robotics and Automation 2 (1985): 530-535.

[11] Oussama Khatib, 'Real-Time Obstacle Avoidance for Manipulators and Mobile Robots', The International Journal of Robobtica Research, Vol. 5, No. 1, Spring 1986, MIT.

[12] Robert Kwiatkowski, 'Gradient Descent Algorithm — a deep dive', https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21.

[13] Howie Choset [et al.], 'Principles of robot motion: theory, algorithms, and implementation', "A Bradford book".

[14] Y. Kuwata, G. A. Fiore, J. Teo, E. Frazzoli and J. P. How, "Motion planning for urban driving using RRT," 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 1681-1686.

[15] S. M. LaValle and J. J. Kuffner. Rapidly Exploring Random Trees: Progress and Prospects. In Algorithmic and Computational Robotics: New Directions, pages 293–308, 2000.

[16] Kuffner, James & LaValle, Steven. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning.. Proceedings - IEEE International Conference on Robotics and Automation. 2. 995-1001.

[17] Vito Trianni, 'Evolutionary Swarm Robotics, Evolving Self-Organising Behaviours in Groups of Autonomous Robots', 2008 Springer-Verlag Berlin Heidelberg.

[18] S. James, R. Raheb and A. Hudak, "UAV Swarm Path Planning," 2020 Integrated Communications Navigation and Surveillance Conference (ICNS), 2020, pp. 2G3-1-2G3-12.

[19] A. Hudak, S. James and R. Raheb, "Impact of Communication Path Loss to Unmanned Aircraft Swarm Coherency," 2021 Integrated Communications Navigation and Surveillance Conference (ICNS), 2021.

[20] M. Saska, T. Baca, J. Thomas, J. Chudoba, L. Preucil, T. Krajnik, J. Faigl, G. Loianno and V. Kumar. System for deployment of groups of unmanned micro aerial vehicles in GPS-denied environments using onboard visual relative localization. Autonomous Robots 41(4):919–944, 2017.

[21] V. Walter, N. Staub, M. Saska and A. Franchi. Mutual localization of uavs based on blinking ultraviolet markers and 3d time-position Hough transform. In 2018 International Conference on Automation Science and Engineering (CASE). 2018.

[22] Puente-Castro, A., Rivero, D., Pazos, A. et al. UAV swarm path planning with reinforcement learning for field prospecting. Appl Intell (2022).

[23] Gageik, Nils & Benz, Paul & Montenegro, Sergio, 2015, 'Obstacle Detection and Collision Avoidance for a UAV With Complementary Low-Cost Sensors'. IEEE Access.

[24] Dashuai Wang, Wei Li, Xiaoguang Liu, Nan Li, Chunlong Zhang, UAV environmental perception and autonomous obstacle avoidance: A deep learning and depth camera combined solution, Computers and Electronics in Agriculture, Volume 175, 2020, 105523.

[25] Jiandong Guo, Chenyu Liang, Kang Wang, Biao Sang, Yulin Wu, "Three-Dimensional Autonomous Obstacle Avoidance Algorithm for UAV Based on Circular Arc Trajectory", International Journal of Aerospace Engineering, vol. 2021, Article ID 8819618, 2021.

[26] J. N. Yasin, S. A. S. Mohamed, M. Haghbayan, J. Heikkonen, H. Tenhunen and J. Plosila, "Unmanned Aerial Vehicles (UAVs): Collision Avoidance Systems and Approaches," in IEEE Access, vol. 8, pp. 105139-105155, 2020.

[27] Steven M. LaValle, Rapidly exploring Random Trees (RRTs), http://lavalle.pl/rrtpubs.html

[28] Aaron T. Becker and Li Huang, Rapidly Exploring Random Tree (RRT) and RRT*, January 2018
https://demonstrations.wolfram.com/RapidlyExploringRandomTreeRRTAndRRT

[29] http://wiki.ros.org/ROS/Tutorials

[30] https://gitlab.fel.cvut.cz/poludmik/uav-usv-path-planning

[31] Using Vector Mathematics, Line segment against sphere intersection test, http://nic-gamedev.blogspot.com/2011/11/using-vector-mathematics-and-bit-of_09.html, November 9, 2011.

[32] Michael Galarnyk *The 68-95-99.7 rule.* Towards Data Science, 2018, Jun 4.

[33] Xiang Yu, Youmin Zhang, Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects, Progress in Aerospace Sciences, 2015.