Bachelarka

Mikhail Poludin

December 2021

Contents

- Theory
  - Sample based path planning

        * RRT algorithm
        * RRT* algorithm
    - Obstacle avoidance
        * Virtual objects with point drone - line/sphere intersects
        * Binary search of collisions
- Analysis
    - time consumption
    - optimality

1 Goals

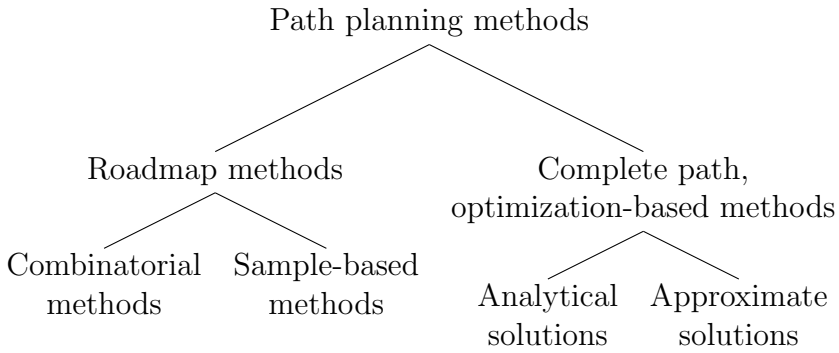
- Investigate the problem of sample based path planning approaches.
- Study the MRS software and learn all the requirements for working with it.
- Implement algorithms of 2D and 3D path planning for UAVs (Unmanned Aerial Vehicle) and test them in different scenarios or simulations.
- Compare main algorithms numerically and decide on advantages of using the ones instead of others in different scenarios.

2 Path or 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. The motion planning algorithms in particular, give a programmer a way to find this sequence of moves so that it satisfies certain constraints (such as are obstacles or computational time). Today there is a wide variety of completely different approaches already, and each one of them is best suited for it's particular purpose.

2.1 Examples of path planning approaches

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



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

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

2.2 Combinatorial path planning

The main concept of combinatorial path planning is dividing the configuration space $Cspace$ into connected regions by any sort of algorithm and then find the way from the points derived out of these regions. The configuration space of a robot, $Cspace$, is a set of all possible positions the robot may be in. For the purpose of this thesis, mainly focused on UAV-like objects, we will take all the possible positions of a drone in an Euclidean space - 3 coordinates and 3 angles of rotation. The crucial part of the methods efficiency is the algorithm to divide the search space.

The space division can be done by a lot of methods and the most naive one is to divide the space into equal square blocks. For a bit more complex example, let's take the polygon cell decomposition approach. As shown in Figure 1, the search space is being divided into polygons, using a triangulation algorithm. Then, the center of gravity is calculated and the final roadmap is built out of them.

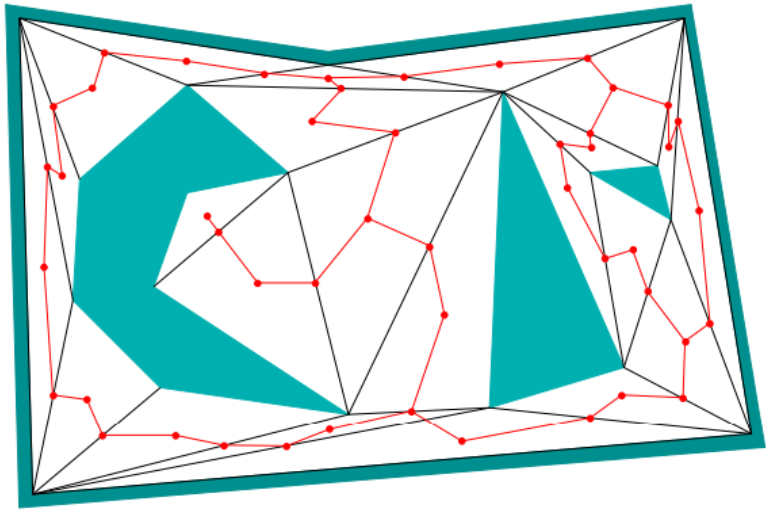


Figure 1: Polygon cell decomposition from path planning book.

Here is another example, Figure 2. This approach is called cylindrical decomposition. The later 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.

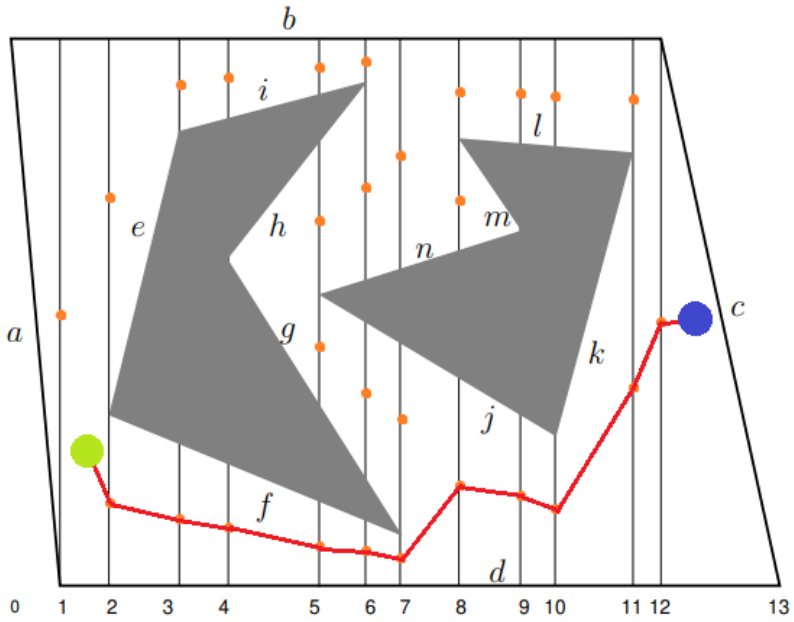


Figure 2: Cylindrical cell decomposition from path planning book with path added by me.

For more efficient search and decomposition, the graphs in forms of trees are used. The most common ones are so-called quad-trees(oc-trees in 3D path planning). The most interesting areas are divided into smaller ones recursively, Figure 3. Such trees make the roadmap smoother and therefore, the found path will be closer to the optimum.

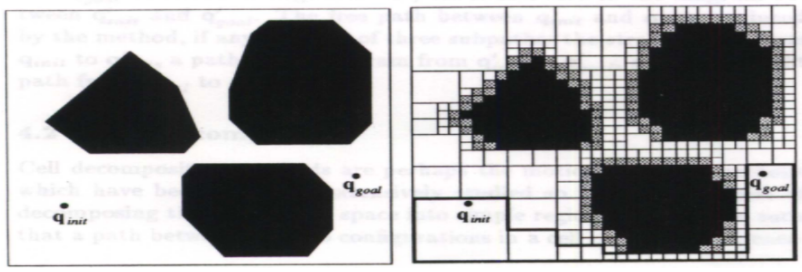


Figure 3: Quad tree decomposition from stanford web site.

2.3 Sample-based path planning

The main idea of sample-based path planning approach is to implement a search that explores the given search space $Cspace$ with a certain sampling scheme. The exploring 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, as the real-life problem is usually represented by real numbers that can be divided into smaller and smaller ones. Since the computers have 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 does 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.

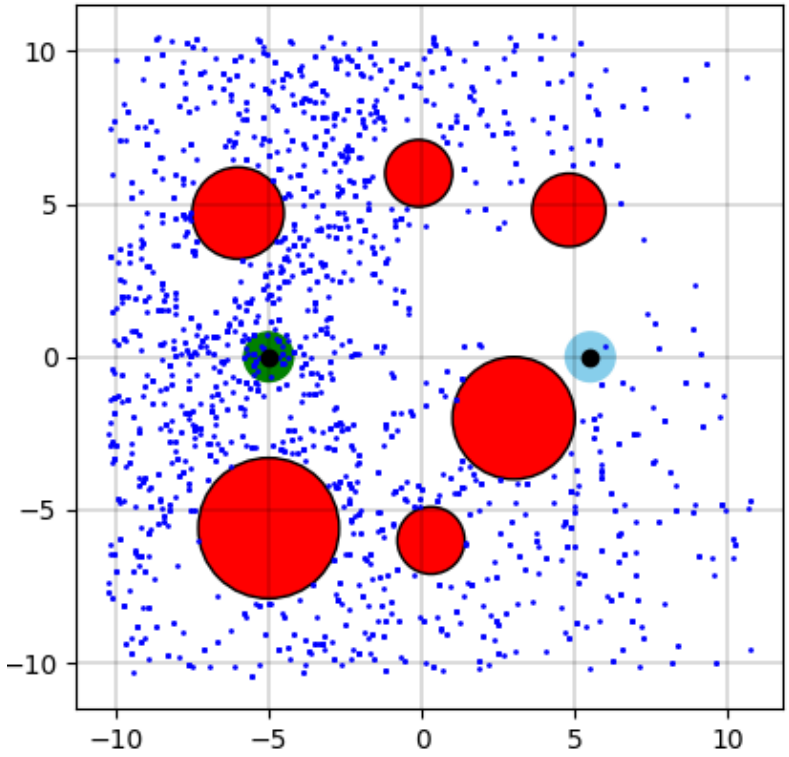


Figure 4: Random point sampling in 2D search space example. Obstacles are shown in red.

As an example for a better understanding of sampling idea consider a 2D rectangle search space (Figure. 4), that 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. The generation of points of interest is what is called "sampling". The sampling can be done differently. The space can be split uniformly, exponentially form the start position or even randomly. The random sampling is the main idea of so called **R**apidly Exploring **R**andom **T**rees, that are the main focus of this chapter.

3 RRT algorithm theory

Rapidly exploring random tree (RRT) is a sample based approach algorithm. It is a widely used algorithm in autonomous robotics, mostly because it is pretty intuitive. It is an algorithm to quickly scan any high-dimensional spaces by constructing a search tree graph, Figure 5, out of randomly selected points from obstacle free areas. RRT generates very rectangular graphs, because the new random state is always attached to it's nearest neighbour.

It is a probabilistic complete algorithm, which means that if the algorithm is run for a long enough time the graph will have a solution - found path, if one exists. It is also a pretty 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 zig-zag 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 comes, the distance between them is verified to be smaller than d. If it is not, then this connection cannot be made.

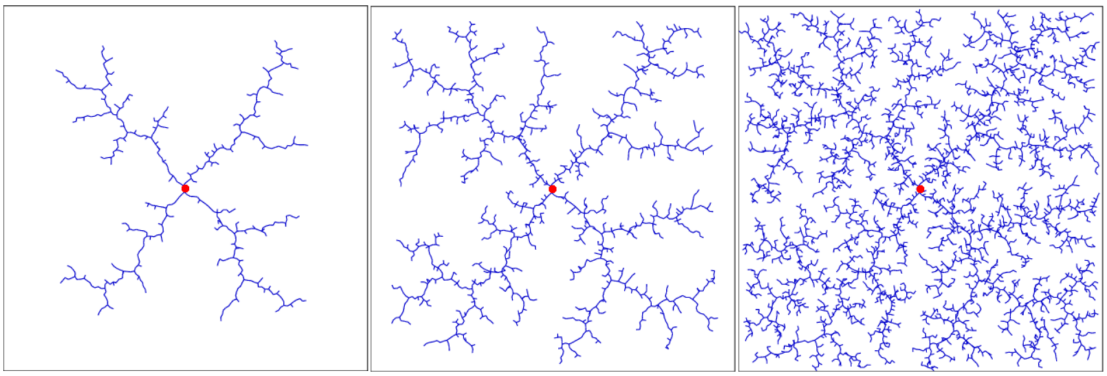


Figure 5: Rapidly exploring random tree expansion, from BC rrt gif.

3.1 Explanation of the algorithm

The search is started with given starting and goal point in high-dimensional 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 area around S_g, in which the path is considered found. Tree graph T is initialised with only one node - S_0. A random point $S_{new} \in C_{space}$ is generated with every iteration of the algorithm. Then S_{new} is investigated to obey the given constraints. Firstly, it is controlled on whether it lays inside any obstacle with obstacle intersection 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 a parent node. Lastly, if S_{new} lays inside the r_g, then the path is considered found and the only thing left to do is to extract the path from T, which can be done by going back by parents to S_0 from S_g.

3.2 Pseudocode

Here are the pseudocodes of the RRT path-finding algorithms that I 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 in RRT -** n**.**
Output: Array of states/path - P**.**

```
T ← tree_init(S_0)
number_of_iters ← 0
while number_of_iters < n do
    S_rand ← get_random_state()
    S_nearest ← get_nearest_state(S_rand)
    if path_is_clear(S_nearest, S_rand) then
        T.add_new_edge(S_nearest, S_rand)
    end if
    if S_rand ∈ S_goal then                ▷ If new state is inside the goal, consider path found
        T.add_new_edge(S_rand, S_goal)
        return extract_path_from_RRT(T)
    end if
    number_of_iters ← number_of_iters +1
end while
```

Algorithm 2 Function to extract path from an RRT tree

Output: array of states/path - P

```
function EXTRACT_PATH_FROM_RRT(Tree T)
    P ← S_g                                ▷ P - path/array of following each other states
    S_tmp ← S_g
    while S_tmp ≠ S_0 do
        P.push_back(S_tmp)
        S_tmp ← S_parent_of_tmp
    end while
    return P
end function
```

4 RRT* path planning algorithm theory

To find a path that is shorter than a common path found with 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. That means that theoretically, when the number of iterations approaches infinity, the algorithm finds the shortest way possible. The growth of an RRT* is shown in Figure.6.

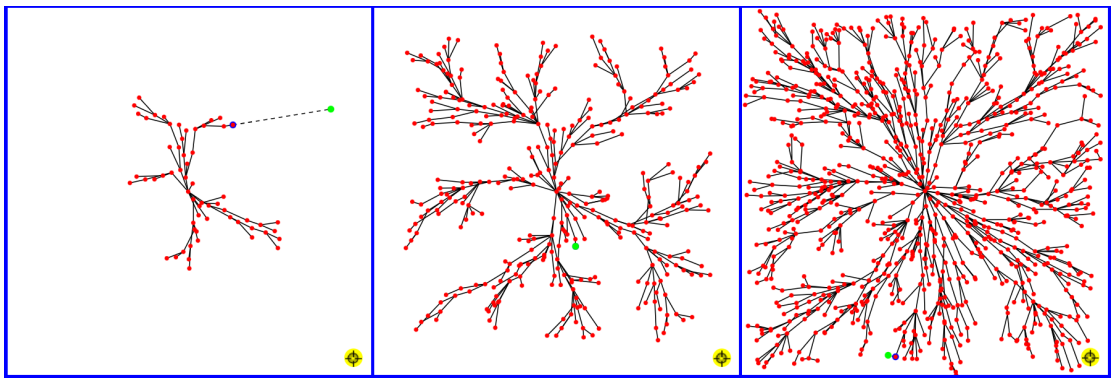


Figure 6: RRT* expansion, rrt online demo.

4.1 Contrast to RRT

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

- Tracking the distance from S_0 to each vertex in a tree. Each vertex would now have a **cost**, which is substantially a sum of distances along current branch in the tree. Now, executing the step with finding the nearest state, the algorithm will also look for a state S_{best} that provides the min{cost(S_{best}) + 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 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 eliminates rectangular shapes in a graph.

- So called **rewiring** of the graph. It happens right after the new state S_{new} has been connected to S_{best}. All the nodes S_i within a defined neighbour radius r_n are inspected weather their cost would decrease if their parent vertex was S_{new}. If it is true, then the graph 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 Pseudocode

Pseudocode representing RRT* algorithm is shown in algorithm 3:

Algorithm 3 RRT* algorithm for path finding

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

$T \leftarrow$ tree_init(S_0)
number_of_iters $\leftarrow 0$
while number_of_iters $< n$ **do**
    $S_{rand} \leftarrow$ get_random_state()
    $S_{best} \leftarrow$ get_best_cost_state(S_{rand}, r_n) ▷ Find the cheapest neighbour in a given radius r_n
    if path_is_clear(S_{best}, S_{rand}) **then**
        T.add_new_edge(S_{best}, S_{rand})
        S_{rand}.cost $\leftarrow$ dist(S_{rand}, S_{best}) + cost(S_{best})
    end if
    for $S_i \in$ Neighbours **do** ▷ Go through all neighbours and **rewire** the tree
        $S_{parent} \leftarrow S_i$.parent
        if dist(S_{new}, S_i) + cost(S_{new}) $<$ dist(S_{parent}, S_i) + cost(S_{parent}) **then**
            S_i.parent.remove_child(S_i)
            S_i.parent $\leftarrow S_{rand}$
        end if
    end for
    if $S_{rand} \in S_{goal}$ **then** ▷ If new state is inside the goal, consider path found
        T.add_new_edge(S_{rand}, S_{goal})
        return extract_path_from_RRT(T)
    end if
    number_of_iters $\leftarrow$ number_of_iters +1
end while

5 Implementation of RRT and RRT*

5.1 Preparation and tool investigation

First thing to do was to investigate software for simulations provided by Multi Robot Systems.

For easy UAVs simulations, **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. User can create so called nodes, that can be subscribed to or publish to a specific topic. Such structure is also good for debug and mistake control. It is a shell application, so to see all the messages being sent through the given topic you can simply write $\$$*rostopic echo /topic_name*. The official tutorials *ROS tutorial link*, were studied to learn how to correctly use the ROS.

Graphical visualisation is done 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 such as are polygon meshes. It also accepts arrays of points, that can be used as a way to graphically visualise tree shaped figures, exactly what we want.

To monitor different running processes and ROS servers I used **htop**. This tool also helps killing applications such that nothing unnecessary remains while next simulations are run. **Tmux** - terminal multiplexer is a tool that is also useful for multiple processes execution. 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 are 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 the project that will contain big amounts of numbers, searches and computations C++ suits better.

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

5.2 Writing environment classes and usage of the main function

I decided to go full C++ and with it being an object oriented language I've written many classes to simplify future work.

5.2.1 World and Object classes

These classes are meant to be used to create and store different objects in 3D space. This ensures easy working with visualisation of objects and obstacles in simulators. Some World methods accept a ROS publisher and send all object data to Rviz in a specified form. Rviz then shows it nicely. Object class is used to contain information about a single object, such as it's 3D position.

5.2.2 Vec3 class

Instances of this class are essentially 3D points that only contain their x, y and z coordinates. 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 generator of a random point in space implemented, that is used drastically in random search trees.

5.2.3 Main function

First thing to do before testing set up algorithms and their classes is to initialise a new ROS node. The ros::Publisher and ros::Subscriber object types will be used to handle desired ROS topics. To see 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 $visualization\_marker$. To correctly use the data from a subscription topic and ensure that data is not changed simultaneously, $std :: mutex$ is introduced.

5.3 Tree structure implementation

I decided to represent the whole tree as a bunch of independent nodes that all have a pointer to their parents instance. And each of them contains an array of pointers to it's own children. // TODO maybe formulate better

5.3.1 Node class

Instance of this class contains 3 coordinates in a form of Vec3 object, cost of this node, raw pointer to a parent node and a vector of shared pointers to all of it's children.

Listing 1: Node object attributes

```
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. Which are frequently used along the whole RRT algorithm execution.

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

5.3.2 RRT_tree class

The pointer to a tree's root is stored in a RRT_tree instance. Besides the constructor and a couple of usefull methods such as writing the whole tree to a JSON file, it involves *find_path* method, that accepts the algorithm to be used and grows a tree according to that algorithm.

5.4 Algorithms implementation

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 base class Algorithm's *find_path_according_to_alg* virtual function is redefined in derived classes RRTAlgorithm and RRTStarAlgorithm.

Listing 2: Algorithm virtual class

```
class Algorithm {
public:
    virtual std::vector<Vec3> find_path_according_to_alg(const World *world_ptr,
                                                         const AvoidanceAlgorithm &avoid_alg,
                                                         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 simultaneously the path has been successfully found, this path requires to be extracted from that tree to a separate array of nodes. For this purpose, there is a static *find_way_from_goal_to_root* method.

5.4.1 RRT algorithm implementation

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

The code is designed the way 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. A start state S_0 is located on the one side and the goal S_g laying on the other, as shown in the Figure. 7.

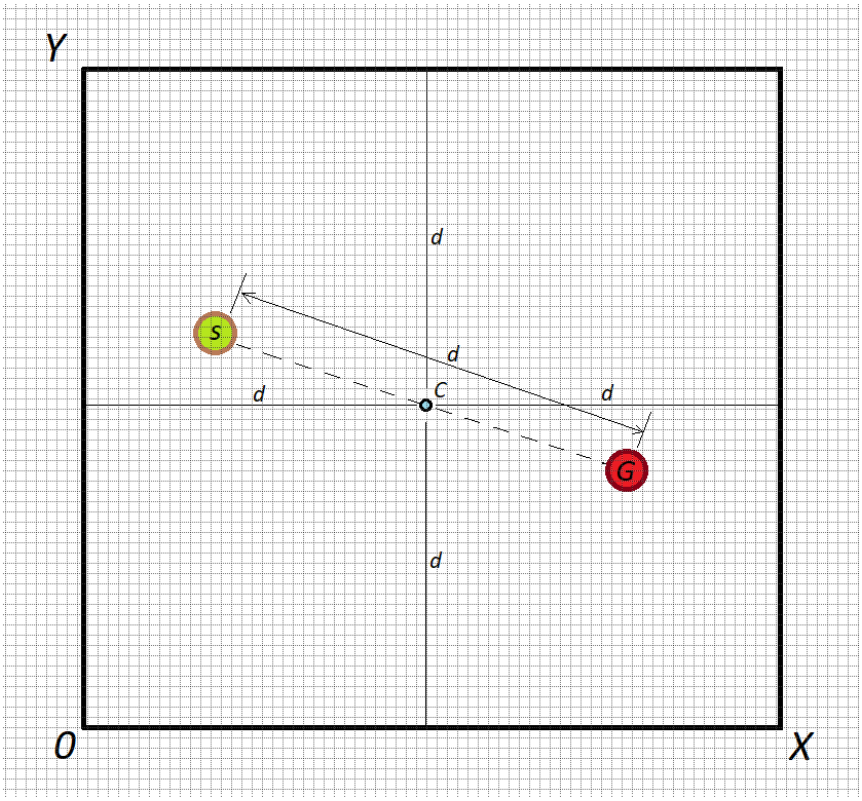


Figure 7: Trivial search area definition, 2D example.

More sophisticated implementations define this area as an ellipse with two focuses in S_0, S_g.
TODO implement maybe

I decided to represent obstacles as spheres, because they are the easiest to work with. After generating a random point, this point needs to be checked on whether it lays inside any obstacle or 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 5.5.

5.4.2 RRT* algorithm implementation

The path-finding and a random tree growth with optimisation and rewiring of the graph was implemented exactly as displayed with pseudocode 3. In contrast with RRT, the track of node costs need to be carried out. Now, when finding the most suitable neighbour node, beside the looking at the distance to it, we also need to look at it's cost in case that it is not the best one.

Listing 3: RRT* best neighbour search code snippet

```
...
std::vector<Node *> neighbors = Node::get_neighbors_in_radius(root, rnd_point,
                                                              neighbor_radius);
Node *best_neighbor = nullptr;
double best_cost_to_new_node = closest->cost + distance_to_closest;
double current_cost;

for (const auto& neighbor:neighbors) {
    is_inside_an_obstacle = false;
    current_cost = neighbor->cost + Vec3::distance_between_two_vec3(rnd_point,
                                                                    neighbor->coords);
    if (current_cost < best_cost_to_new_node) {
        // Check potential best neighbors
        for (const auto& obst : world_ptr->obstacles) {
            if (avoid_alg.ThereIsIntersectionAlongThePath(...)){
                is_inside_an_obstacle = true;
                break;
            }
        }
        if (is_inside_an_obstacle) continue;
        best_cost_to_new_node = current_cost;
        best_neighbor = neighbor;
   }
}
...
```

5.5 Obstacle avoidance algorithms

Obstacle avoidance can be done in a couple of fashions. I implemented two of them, similarly as the *Algorithm* class has a virtual function that is redefined in *RRTAlgorithm* and *RRTStarAlgorithm* classes, here I have *AvoidanceAlgorithm* class with a virtual function *ThereIsIntersectionAlongThePath()* that I redefine in *LinearAlgebraIntersection* and *BinarySearchIntersection* classes.

5.5.1 Point drone - inflated/virtual obstacles, line/sphere intersects

This approach assumes a drone being a single point. However all the obstacles, that I have represented as spheres, have 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.
TODO pictures of the algorithm working with point drone (highlight the obstacles and the radius around them) This way of obstacle avoidance requires three conditions to be satisfied to put a new trajectory point to the graph:

1). If a new point lays inside an obstacle,

2). If a line segment between two points intersects 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 sphere obstacles, the **first condition** can easily be judged from the distance between the obstacle centre S_o with radius R and the new point:

$$dist(S_n, S_o) < R$$

Nonetheless, the **second condition** requires a little bit more complicated maths. It's implemented in *LinearAlgebraIntersection* class. Let's assume we have two points S_1, S_2 and we want to check if the

line segment between them intersects the certain sphere with a coordinate vector $\mathbf{S}$ with radius r. Firstly, we need to find the closest point to $\mathbf{S}$ laying on the line. Let's find the difference of S_1 and S_2,

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

Then find the squared length of this segment, in other words vector's norm squared,

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

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

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

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 isn't in between of [0, 1] interval, then make it 0 or 1.

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

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

Closest point to the centre of sphere $\mathbf{C}$ can be calculated as following:

$$C = \mathbf{S_1} + P * \mathbf{d}.$$

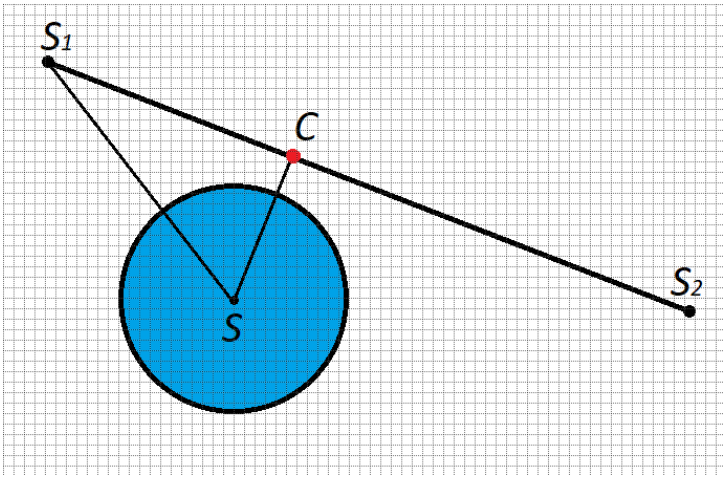


Figure 8: 2D example of line segment with circle intersection, $\mathbf{C}$ isn't inside of a sphere

Now that we have $\mathbf{C}$, all that remains is to check whether it lays inside this sphere, with is almost trivial(first condition of this avoidance approach).

5.5.2 Sphere drone - binary search of collisions

In this approach, the drone is assumed to be a sphere of the appropriate radius. That means that no safety indent needs to be added to the obstacles, unlike the case with a point drone.

6 Multi robot systems environment

The implementation of methods and algorithms will be done with usage of the software that is used by MRS system.

6.1 Programming languages

Programming was performed in **C++** language, with usage of **python** to plot data. All the scripts are available on GitLab.

6.2 Simulation and communication tools

To simulate the UAVs and visualise the algorithms, a lot of modules need to be communicating. The **ROS** was used to bind drone's and some of the following software nodes together.

- RViz. Environment for easy visualisation of algorithms. Being a ROS subscriber node, collects the data being sent to it and shows it in 3D.
- Gazebo. Simulator for real-life graphics drone simulation.
- htop. Application to monitor running ROS servers.
- tmux. Terminal assistant.