A
REPORT
ON

RRT* Path Planning with a dynamic obstacle

By

Dev Rupesh Mehta	**2019B1A41084G**
Prateek Nanhorya	**2019A4PS0564G**

Under the guidance of

Dr. Abhishek Sarkar and Dr. Ashwin K.P.

At

BITS Pilani KK Birla Goa Campus

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI, KK BIRLA GOA CAMPUS

(2nd Semester, 2022-23)

INDEX

1.0 Introduction

1.1 Overview of the work

Path planning algorithms are critical for finding a safe and efficient path for a robot or autonomous vehicle to navigate from a start point to a goal point in an environment. However, in dynamic environments where obstacles are not static and may move unpredictably, it becomes even more challenging to find a safe and optimal path.
Path planning algorithms are essential in dynamic environments for several reasons, including safety, efficiency, and adaptability. In a dynamic environment, a robot or vehicle must avoid collisions with moving obstacles to ensure safety. Path planning algorithms can take into account the changing positions and velocities of obstacles and plan a path that avoids collisions.
Efficiency is also essential in dynamic environments, and finding a safe path quickly is crucial. Path planning algorithms can quickly adapt to changes in the environment and generate a new path in real-time, reducing the time required to reach the goal.
Adaptability is another critical factor in dynamic environments. The environment is unpredictable, and the robot or vehicle must be able to adapt to changes. Path planning algorithms can update the planned path based on new information, allowing the robot or vehicle to navigate through the environment smoothly.
RRT* (Rapidly-exploring Random Trees) is a popular motion planning algorithm used for finding feasible paths in robotics applications. It is an extension of the RRT algorithm that guarantees asymptotic optimality, which means that as the algorithm runs for longer, it will eventually find the optimal path between the start and goal states, if such a path exists.
Overall, using RRT* for path planning with static and dynamic obstacles requires a combination of collision detection, obstacle prediction, and path optimization techniques. While the algorithm can be computationally expensive, it is well-suited for real-time motion planning applications.

1.2 Literature

Our project has been mainly been inspired from the research paper- MOD-RRT*: A Sampling-Based Algorithm for Robot Path Planning in Dynamic Environment.
MOD-RRT* is a new sampling-based algorithm for robot path planning in dynamic environments. It is designed to quickly adapt to changes in the environment and generate collision-free paths for the robot. The algorithm uses a modified version of the Rapidly-Exploring Random Tree (RRT*) algorithm, which is a popular approach for motion planning in robotics.
The key contribution of MOD-RRT* is the introduction of a dynamic obstacle model that can predict the future positions of moving obstacles based on their current trajectory. This allows the algorithm to consider the future positions of obstacles when generating paths for the robot, which improves the reliability and safety of the generated paths.
The algorithm also includes a mechanism for detecting changes in the environment and updating the obstacle model accordingly. This ensures that the algorithm can quickly adapt to changes in the environment, such as new obstacles or changes in the trajectory of existing obstacles.

The authors of the paper conducted experiments to compare the performance of MOD-RRT* with several other state-of-the-art algorithms for dynamic path planning. The results showed that MOD-RRT* outperformed the other algorithms in terms of success rate, path length, and computation time.
Overall, MOD-RRT* is a promising new approach for robot path planning in dynamic environments, with potential applications in areas such as autonomous vehicles, robotics, and industrial automation.
In the paper, the authors propose an extension of the MOD-RRT* algorithm which is designed to generate multiple Pareto-optimal paths in dynamic environments. The Pareto dominance criterion is used to evaluate the optimality of different paths based on multiple objectives, such as path length, clearance from obstacles, and proximity to the goal.
The paper presents experimental results showing its effectiveness in generating multiple Pareto-optimal paths for different scenarios in dynamic environments. The authors also demonstrate how the algorithm can be used to generate paths with different trade-offs between the multiple objectives, allowing the robot to make decisions based on its preferences and constraints.
Overall, the paper provides a valuable contribution to the field of multi-objective path planning in dynamic environments, and highlights the potential of the Pareto dominance criterion for evaluating the optimality of different paths based on multiple objectives.
Pareto dominance is a concept used in multi-objective optimization to evaluate the optimality of different solutions based on multiple criteria or objectives. A solution A is said to dominate another solution B if it is better than or at least equal to B in all objectives, and strictly better in at least one objective.
In other words, if there are two or more objectives to be optimised, Pareto dominance is used to identify the solutions that are not dominated by any other solution. These solutions are called Pareto-optimal or non-dominated solutions.
The Pareto dominance criterion is based on the idea that in a multi-objective optimization problem, there is usually no single solution that is the best in all objectives. Instead, there may be a set of solutions that represent trade-offs between the different objectives. The Pareto-optimal solutions represent the best possible trade-offs between the objectives, and the decision maker can choose the most suitable solution from this set based on their preferences and constraints.

1.3 Objectives

Following are the main objectives we aimed to achieve in this project:

- Creation of a map with static obstacles with known positions.
- Simulation of RRT* path planning algorithm using openFrameworks. (openFrameworks is an open source C++ toolkit for creative coding.)
- Introducing a dynamic obstacle with an unknown trajectory(randomised), wherein the trajectory is controlled by the user via the pointer tool.
- Implement path replanning using Pareto Dominance criterion due to interference by a dynamic obstacle.

2.0 Theory

To use RRT* for path planning with static and dynamic obstacles, we have followed these general steps:

1. Defining the state space: The first step is to define the state space for the robot. This typically includes the position and orientation of the robot, as well as any other relevant parameters such as joint angles or velocities.
2. Defining the cost function: The cost function determines the cost of moving from one state to another. For example, the cost might be proportional to the distance between states or the time required to move between them.
3. Generating an initial RRT: The RRT* algorithm starts by generating an initial tree with a single root node at the start state of the robot. The tree is grown by adding new nodes in the direction of randomly sampled states within the state space.
4. Adding static obstacles: To account for static obstacles, you can use collision detection to determine which nodes in the RRT are in collision with the obstacles. If a node is in collision, it is discarded and not added to the tree.
5. Adding dynamic obstacles: To account for dynamic obstacles, you can predict the future positions of the obstacles and check for collisions with the RRT nodes. If a node is in collision with a predicted obstacle, it is discarded and not added to the tree.
6. Updating the RRT: As new nodes are added to the tree, the cost of each node and its parent is updated to reflect the path cost along the tree.
7. Connecting the tree to the goal state: As the RRT grows, it is likely that a node will be added near the goal state. To connect the tree to the goal state, you can generate a short RRT from the goal state backwards towards the tree.
8. Optimising the path: Once the RRT is connected to the goal state, the algorithm can optimise the path by rewiring the tree and adjusting the costs of nodes.
9. Repeat: The algorithm continues to generate new nodes and optimise the path until a feasible path is found or a maximum number of iterations is reached.

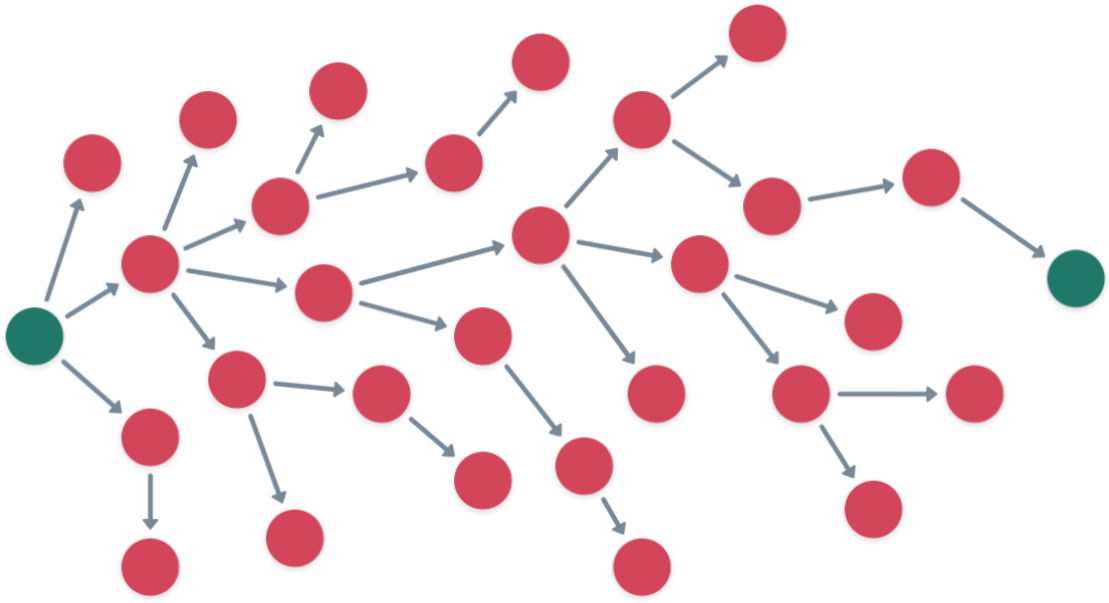

Figure 1: A schematic of the functioning of an RRT* algorithm

2.1 Descriptions

The code is written in c++ using the Open-Frameworks library.

2.2 Map representation

We thought it would be better to have a map which can be represented with any kind of shapes. To achieve this, a class hierarchy was made.
The given class hierarchy showcases the concept of inheritance and polymorphism in object-oriented programming using C++. The Obstacle class is the base class, and it is inherited by the Polygon and Conic classes. The Polygon class further derives Triangle and Quadrilateral, and Quadrilateral has Rectangle as its derived class. The Rectangle class then derives Square. On the conic side, Ellipse is a derived class of Conic, and Circle is a derivative of Ellipse. This hierarchy demonstrates the principles of polymorphism where derived classes can have their own specialised implementation of functions and data members, which can be accessed through a pointer or reference of the base class type. For example, a pointer to an Obstacle object can be used to point to an object of any derived class type and call its functions polymorphically. This hierarchical structure also allows code reuse by avoiding redundant implementation of common properties and methods in multiple classes.
Each edge in a polygon type has a starting and an ending point. The obstacle region is defined as the region lying towards the left side of the respective edge. One can make instances of Polygon type (input is a vector of coordinate points), or any of its derivative classes. Note that we restricted the obstacles formation to closed shapes only. That is the point coordinates are taken in counter clockwise format
To incorporate the region outside the window frame efficiently, we have made it an obstacle of rectangle type.

2.3 Algorithm steps and Description

The map that will be provided as a preliminary source contains description of the static obstacles. A description of start and goal state is also provided alongside. Typical RRT star algorithm works by sampling nodes by random selection. Post this, a rewiring operation finishes the loop by optimising the distance from the start using the known nodes.
The pseudocode for the RRT-star algorithm is given in *figure 2*.
Initially, the *Sample* function randomly selects a state from the free space. Now we select the *Nearest* node to this sample node from the already existing tree. A node is made by driving in the direction of the sample node from the nearest node to a specific distance with the aid of the *steering* function. If the path from the nearest node to the new node formed is obstacle free, we can undoubtedly add the new node to the tree. Further we determine the nodes near to the new node and perform *rewiring.* The loop termination criterions can be the maximum number of nodes in the tree or the path cost less than equal to pre decided optimal cost. The code provided in this paper assumes the prior termination.

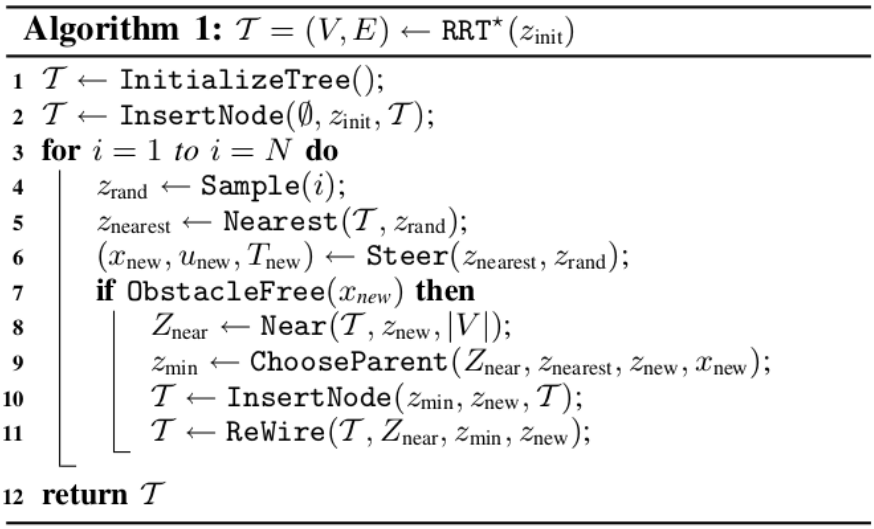

Figure 2: Pseudocode for the RRT* algorithm

One of the main challenges in motion planning arises when dynamic obstacles are introduced into the environment. The uncertainty caused by the presence of moving obstacles can be addressed by adapting the RRT* algorithm to rearrange the tree structure or prune nodes and edges.

To handle dynamic obstacles, we continuously traverse the existing tree structure at regular intervals. If a new obstacle is encountered that collides with existing nodes or edges, we isolate the corresponding node and make it an orphan, appending it to a list of dangling nodes.

We then choose a new parent node from the remaining tree structure that is nearest to the orphaned node. However, if the edge joining the dangling node and parent node collides with any obstacle, indicating that the nearest node is not feasible, the entire branch is pruned.

Challenge arises when removing nodes in the branch corresponding to a dangling node. In traditional implementations, each node in the branch must be removed from the node list and then appended back to the list after determining a proper parent. To overcome this challenge, we have implemented dynamic memory allocation, which operates by using a pointer to a node. With this approach, removing nodes from the node list and appending them back after determining a new parent is unnecessary. Instead, we simply modify the pointers of the affected nodes to point to the new parent node.

This implementation requires restructuring the entire RRT* algorithm to accommodate the use of pointers and dynamic memory allocation. However, this modification allows for more efficient pruning of nodes in response to dynamic obstacles, improving the robustness and adaptability of the algorithm for real-world applications.

1. Description of a Node (rrtNode):

A class for node having data members as follows:

- A coordinate representing data member `ofVec2f` provided by open-frameworks library.
- A pointer to another node behaving as parent. It is NULL for the start node.
- A vector of pointers to nodes behaving as children of the current node. Empty initially.
- The cost to the current node. Initially zero.

2. Description of RRT star (rrtStar):

A class implementing rrtStar for the given inputs. The data members and methods include:

- A pointer to the start node.
- The coordinate to the goal state.
- A bool value representing whether the goal state is reached or not
- A pointer to the node which is within the desired vicinity of the goal state.

➔ A method `update()` to implement a single node of RRT star algorithm. A continuous loop of this results in a complete algorithm.
➔ A method `dynamicUpdate()` to incorporate the updates imposed due to collision with dynamic obstacles.
➔ Other methods to support algorithm implementation.

Since we have dynamically allocated memory, it would be efficient to use recursion based implementations using dynamic programming.
The recursive call to find the dangling obstacles taking start node as input works as follows:
If the given node contains a parent, and the edge connecting them is colliding, we first change the parent of the children of the current node to NULL and then give the recursive call to each child. And if the node is not colliding, we just recursively call each child.
If the given node does not contain a parent, i.e. it is either a start node or it got removed from the main branch. If it is a start node, we simply call its children recursively. If the node is colliding, again we call its children recursively. Finally, if the node is not colliding and it does not have a parent means that it defines a dangling node. We append this node to the dangling nodes list.
The list generated is then traversed to find a parent of each node and connect it back to the existing graph. Note that we do not need to add all the nodes back to the graph; instead we just add one node and all the dynamically allocated branch nodes follow. This reduces the algorithm time significantly. In case, we do not get a nearest node which signifies that the edge to the nearest node is not feasible, we prune the branch completely.

The pseudo-code is expressed below:

---

dynamicUpdate():

```
    list danglingNodes
    dynamicUpdate_generateDangling(danglingNodes)

    for(node in danglingNodes):
        parent <- nearest(node)
        if(node to parent edge is collision free):
            parent.children + node
            node.parent <- parent
        else:
            delete node
```

```
dynamicUpdate_generateDangling(node):
    if (node has parent):
        if(line joining node and parent collide):
            node.parent.children - node
            if(node collides):
                for child in node.children:
                    child.parent <- NULL
                    dynamicUpdate_generateDangling(child)
            else:
                potential_parent <- nearest(node)
                if(potential_parent exists):
                    parent.children + node
                    node.parent = potential_parent
                else:
                    for child in node.children:
                        child.parent <- NULL
                        dynamicUpdate_generateDangling(child)
        else(line joining node and parent does not collide):
            for child in node.children:
                child.parent <- NULL
                dynamicUpdate_generateDangling(child)

    else(node does not have a parent):
        if(node is equal to start):
            for child in node.children:
                child.parent <- NULL
                dynamicUpdate_generateDangling(child)
        else if (node is colliding):
            for child in node.children:
                child.parent <- NULL
                dynamicUpdate_generateDangling(child)
        else:
            dangling_nodes + node
```

Now, as we obtain a single path towards the goal, a robot can start following the path. But since the path is variable subject to dynamic interruptions, the robot needs to make a decision while encountering a node with multiple children. This problem can be overcome by using the Pareto dominance criterion as stated by Qi, Yang, and Sun (1). Briefly, the criterion aids in making an optimal decision when the agent has multiple options by taking into account all sets of variables.

2.4 Comparison with other methods

Modular RRT* is an extension of the RRT* algorithm that decomposes the problem into smaller subproblems or modules, each of which is responsible for handling a specific aspect of the planning process. For example, one module might be responsible for obstacle avoidance, while another module might be responsible for optimising the path.
Genetic Algorithm, on the other hand, is an optimization technique inspired by natural selection. It works by maintaining a population of candidate solutions and evolving them over time by applying genetic operators such as mutation and crossover.
We tried to understand and explore all the three algorithms- RRT*, mod RRT* and genetic algorithm but finalised RRT* due to relevance to the problem and complexity in code.

3.0 Simulation and Results

3.1 Inputs

The input is in the form of a map which is user defined. As stated in the Map class description, one can produce any kind of map using it.
Other than this, we define some dynamic obstacles using the same class which will be user controlled. Next we need a start goal and end goal for path planning. To make the program user friendly, we make the goal state user controlled. The code can be found in the GitHub repository stated in resources(4).

3.2 Outputs - Plots, Graphs, Tables

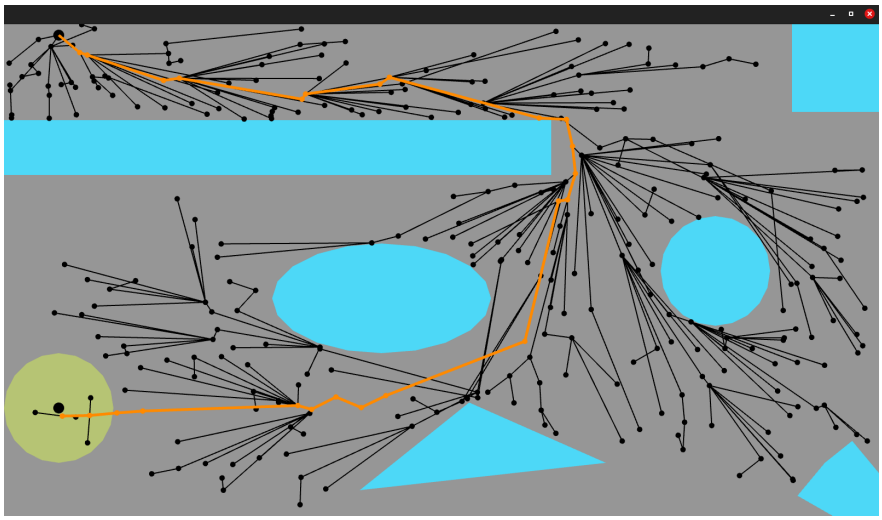

Figure 3: An implementation of RRT star algorithm with static obstacles

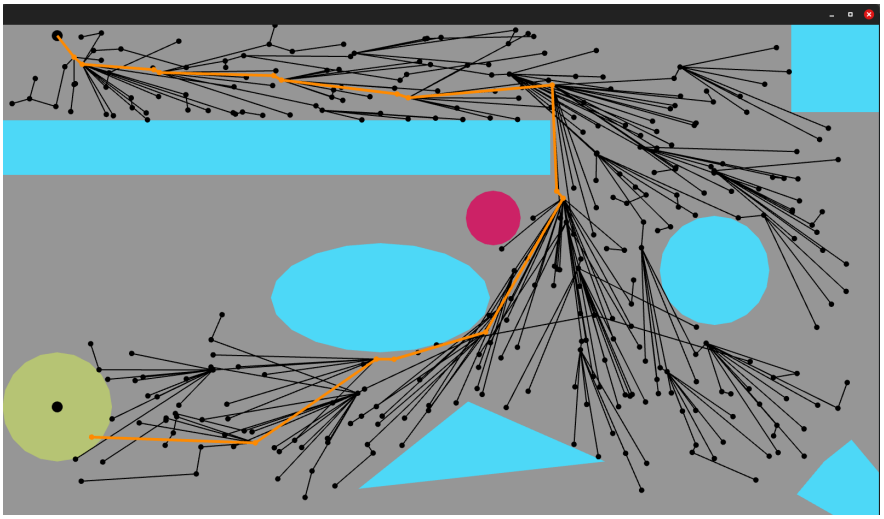

Figure 4 : The red circle is user controlled and behave as a dynamic obstacle

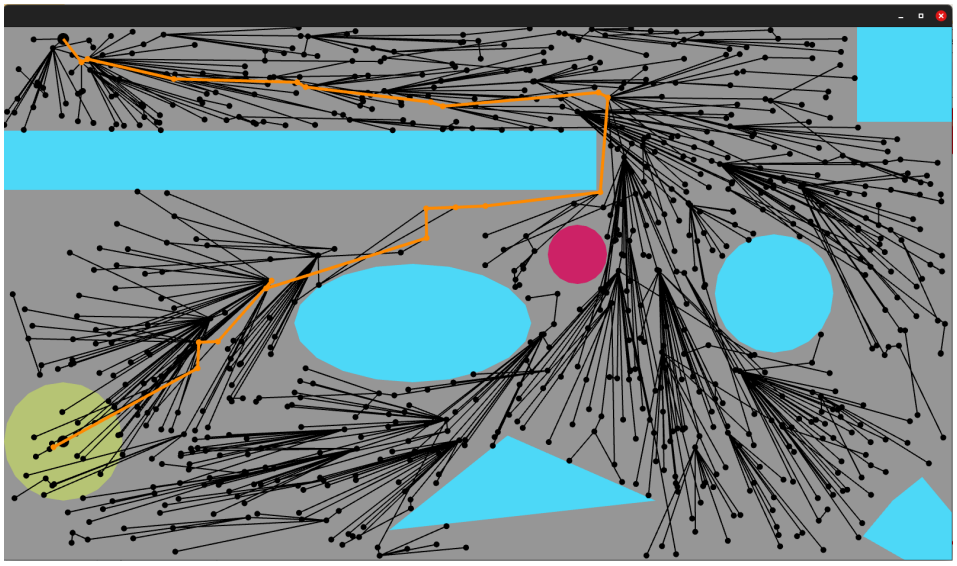

Figure 5: Path updated on moving obstacle

4.0 Conclusion

In conclusion, the objectives of this report were to create a map with static obstacles and simulate the RRT* path planning algorithm using openFrameworks. Additionally, the report aimed to introduce a dynamic obstacle that was unknown and randomised, which could be controlled by the user, and to implement path replanning due to interference by the dynamic obstacle.

The proposed modification to the RRT* algorithm is a promising approach for generating collision-free paths for robots in dynamic environments. By using a dynamic obstacle model and a mechanism for detecting changes in the environment, the algorithm can quickly adapt to changes in the environment and generate paths that consider the future positions of obstacles. Moreover, it enables the generation of multiple Pareto-optimal paths for multi-objective path planning in dynamic environments.

The experimental results of this report demonstrate the effectiveness of the algorithm in generating collision-free paths in dynamic environments. The algorithm outperforms several state-of-the-art algorithms in terms of success rate, path length, and computation time. The algorithm can be used to develop autonomous systems with enhanced safety and reliability in environments with both static and dynamic obstacles.

Overall, the proposed algorithm is relevant to the objectives of this report. It provides a solution for generating collision-free paths for robots in dynamic environments, even with unknown and user-controlled obstacles. The path replanning mechanism of the algorithm is effective in avoiding collisions and ensuring the safety of the robot. Therefore, the modified RRT* algorithm we have developed could be a valuable contribution to the field of path planning in dynamic environments, and can be used in various real-world applications, including autonomous vehicles and industrial automation.

5.0 References

1. Qi, Yang, and Sun (2021).MOD-RRT*: A Sampling-Based Algorithm for Robot Path Planning in Dynamic Environment. IEEE Transactions on Industrial Electronics.
2. Karaman, Walter, Perez, Farazzoli, Teller (May 2011). Anytime Motion Planning using the RRT*. IEEE Conference on Robotics and Automation. MIT. http://hdl.handle.net/1721.1/63170
3. Ant Colony Optimization Algorithms. (August 2018). Ant colony optimization algorithms - Wikipedia
4. **Resource**: https://github.com/xD-prateek/Path-Planning-in-dynamic-environments-using-RRT-star.git