Term Project Report

Path Planning in Search Space using Rapidly exploring Random Trees

ENGR-E 599 - Autonomous Robotics

Introduction

Path Planning in Robotics is a computational problem to determine the path a robot can take to reach the goal state from the start state. For example, consider a robot that has to move around in an office space to find its way to a particular predetermined destination. The robot has to avoid running into walls, desks, chairs and only navigate in the free space available. Path planning algorithms help in determining an efficient path for the robot to reach its destination. There are many different path planning algorithms that can be employed for this purpose, like DFS, A* search, Probabilistic Roadmap etc. In this term project, we explore 'Rapidly exploring Random Trees', also called as RRT, as a path planning algorithm.

RRT Algorithm

The RRT algorithm efficiently searches a high-dimensional space by constructing a random space-filling tree. The tree is constructed by drawing random samples from the available free space in the environment (also called the Search Space), and is inherently biased towards exploring unexplored areas. This algorithm helps in easily navigating around obstacles in the Search Space and efficiently obtain a path to the goal. It is a Monte-Carlo algorithm, which means that it has a certain failure probability but that can be mitigated by adding bias towards unexplored areas and moving towards the goal state. This algorithm depends on random sampling, and does not depend on discretizing the entire environment into a grid structure (as required by many algorithms like DFS, A* search etc.). This makes RRT algorithm memory efficient, as it does not have to store the entire discretized environment in memory.

Technical Details

This project simulation has been developed with the help of PyGame library. The file *main.py* is the main executable python script that accepts command line arguments to initialize what type of obstacles are to be initialized in the environment. There are multiple options included :-

- **-s or --standard**: This initializes the environment with 3 predefined obstacles.
- **-b or --blocked**: This initializes the environment with an obstacle that makes it impossible to reach the goal state.
- **-n or --narrow**: This initializes the environment with a couple of obstacles that give a very narrow opening to travel towards the goal state. This reduces the probability of finding a path to the goal and path planning takes longer to find a path.
- **-r or --random**: This initializes the environment with 35 randomly generated obstacles. It helps in gauging real world performance of the algorithm in new and unseen environments.

Sample execution script - **python3 main.py --random**

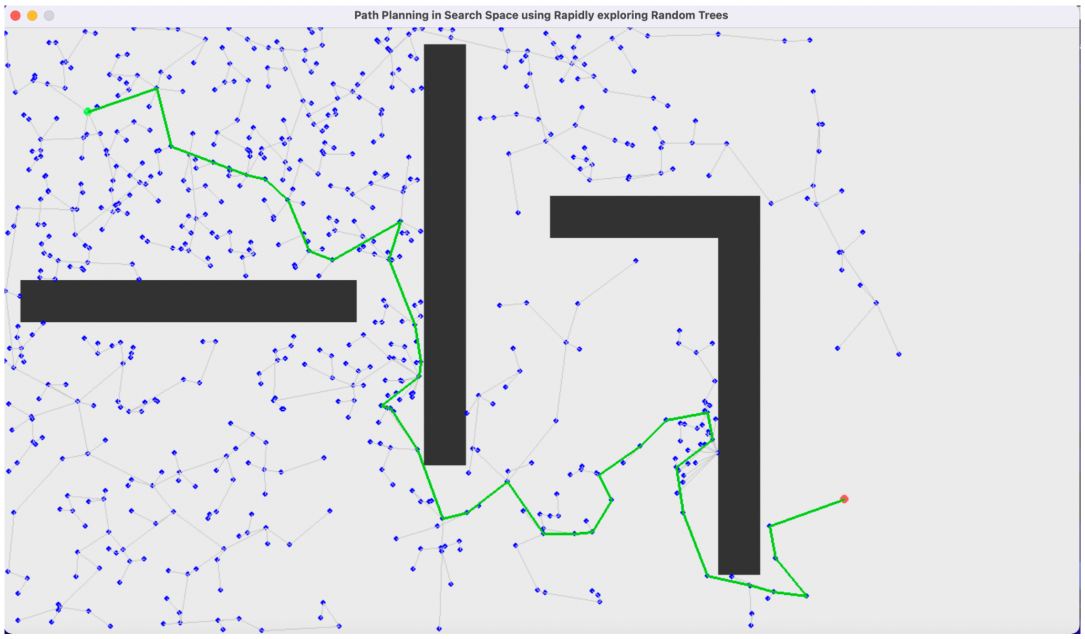
Obstacles are initialized based on the command line argument passed. The pygame environment is initialized and the RRT algorithm begins execution 1 second after the simulation is displayed. A total of 1000 iterations is considered for the environment regardless of the number and size of obstacles. If the path is not found in 2000 iterations, it is considered that the path to the goal does not exist. It may in reality exist, but this is done in order to limit the computation and show the disadvantage of Monte-Carlo based algorithms in general. Setting a higher limit for this directly results in a higher probability of a path to goal being found.

The algorithm works with two functions, one for exploration and one for bias. 90% of the iterations are dedicated towards exploration of the search space, and 10% of the iterations are dedicated towards biasing exploration towards the goal state. A random point is considered using Python's *random.uniform()* function and it is checked for collision with any obstacles. The next step is to find the nearest point to this random point and check if it can be connected to this point without collisions with any of the obstacles. Once both these checks are passed, this vertex is added along with the connecting edge to the random tree. The bias function checks for the point nearest to the goal and generates a random sample within a certain maximum distance from it (set to 50 in this case) and adds it to the random tree and after performing the aforementioned collision checks.

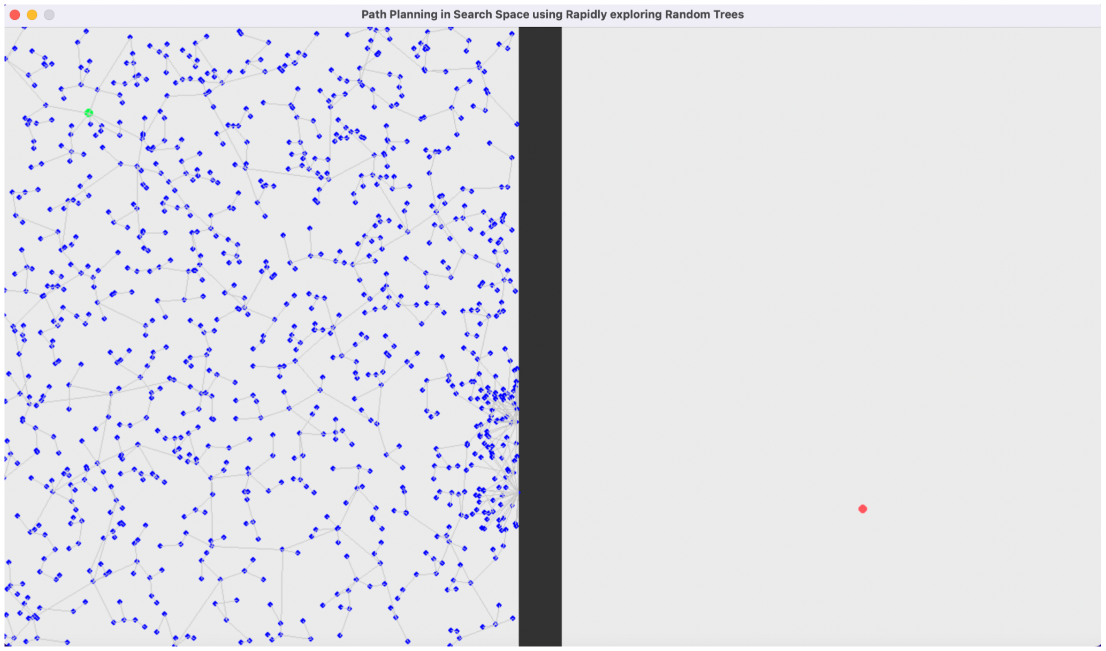
In the end, if the algorithm succeeds in finding a path to the goal, the path is highlighted in green. Else, it is left as is and the entire exploration path is displayed. The whole simulation can be observed as it happens in real time.

Results

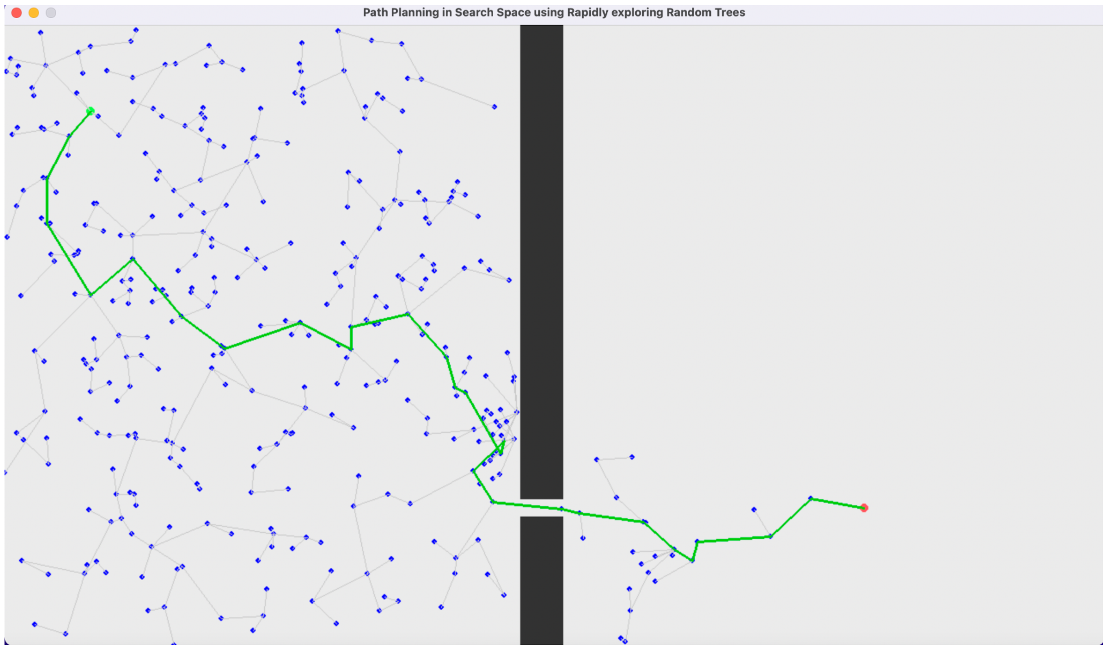
- Standard Obstacles


Path Planning in Search Space using Rapidly exploring Random Trees

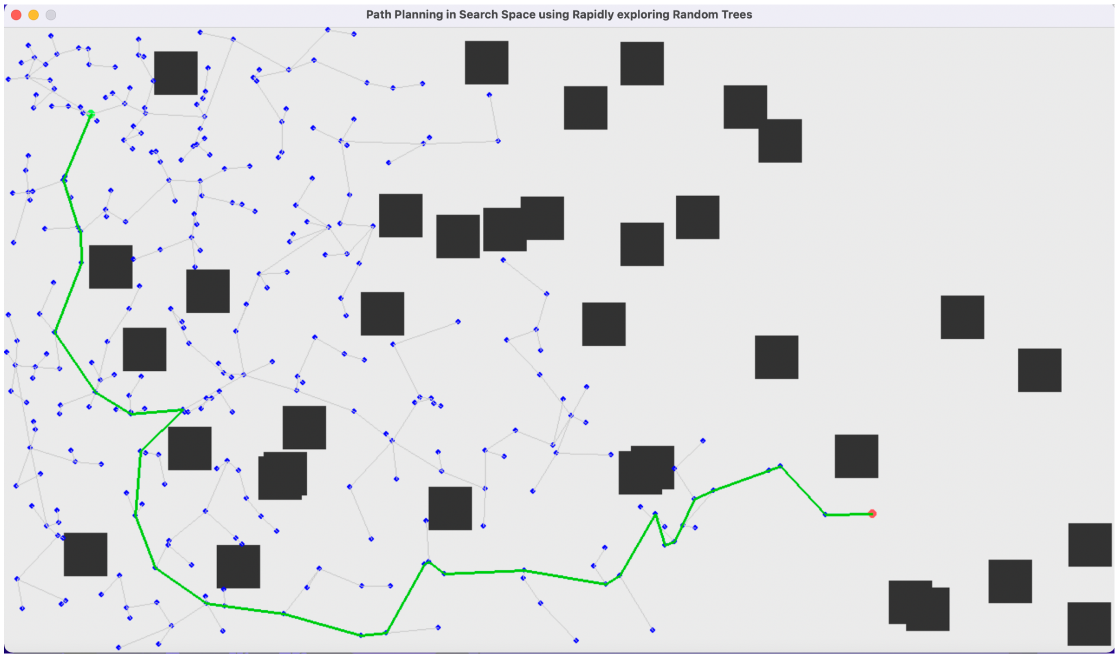
- Blocked Obstacle


Path Planning in Search Space using Rapidly exploring Random Trees

- Narrow Obstacle


Path Planning in Search Space using Rapidly exploring Random Trees

- Random Obstacles


Path Planning in Search Space using Rapidly exploring Random Trees

Discussion

The obtained results are not exactly optimal, but they come quite close. Given additional time to develop this, I would primarily work on two changes:-

1. Implementing bias towards free space between obstacles. This would ensure that obstacles that allow only a narrow space of passing would be explored efficiently since vertices that have exposure towards ends of obstacles would be prioritized for exploration.
2. Explore multiple paths to the goal with a certain hard limit on the number of computations or computation time and return the best path. This would make sure that the generated path is as close to an optimal path as possible.

Optimality cannot be theoretically guaranteed when using RRT Algorithm as it is a Monte-Carlo algorithm that works with random sampling and probabilities. But when implemented along with the aforementioned two points, it makes sure that a path that is close to optimal is returned in a very short period of time compared to other algorithms that may guarantee optimality but take much longer to find a path.