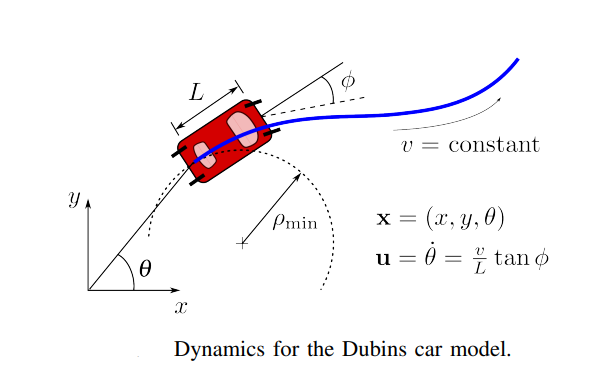
**CPSC 8810: Motion Planning - Final Project Report**

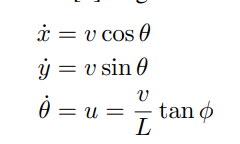
**Project Title:** Nonholonomic Motion Planning for a Car in 2D Environments using RRT\*

**Team Members:** Kalpit Madhusudan Vadnerkar, Vasanth Seethapathi

**Overview of the Project:**

The objective is to plan the motion of a car in 2D environment that is not confined to the roads,i.e an off road environment. The car is considered to be a Dubins Car which is subject to non-holonomic constraints where RRT\* algorithm is implemented for path planning, which can effectively generate paths to navigate through the 2D environment.

****

****

Dubins car is a type of non-holonomic vehicle used in the research field of autonomous navigation that is constrained to move along circular arcs with a fixed turning radius. The Dubins car model is constrained in such a way that the maximum steering angle imposes a minimum turning radius on the vehicle.

**Description of your implementation:**

In this project, we implemented a path planning algorithm using RRT\* (Rapidly-exploring Random Trees) combined with Dubins paths for a simple car model. The goal of the implementation is to efficiently find a feasible and safe path for the car to navigate from a starting point to an endpoint in an environment with obstacles.

* Car Model: We used the SimpleCar class to represent the car in the environment. This model considers the car's kinematic constraints such as maximum steering angle, turning radius, and position information. The object also takes in the start, and end positions of the car.
* Environment: The Environment class represents the 2D environment containing the obstacles, and the car. The environment is defined with a given length and width, as well as a list of obstacles represented as rectangles.
* RRTStar: We implemented the RRT\* algorithm in the RRTStar class. RRT\* is an extension of the RRT algorithm, with the added benefit of being asymptotically optimal. The algorithm uses a tree data structure, where each node in the tree represents a valid configuration of the car. During the search process, the algorithm samples random configurations and connects them to the tree by finding the nearest node and extending the tree towards the sampled point.
* Dubins Path: To ensure that the car's motion constraints are considered in the path planning, we used Dubins paths to connect the nodes in the RRT\* tree. The DubinsPath class calculates the shortest path between two configurations in the plane, considering the car's turning radius constraints.
* RRTStarExploration: We created a new class, RRTStarExploration, which is a modified version of the RRTStar class to allow for complete map exploration, using a search path method that explores the map and returns a list of nodes.

The main script brings together all the components to find a safe and feasible path from the start to the goal position in the environment. We used the TestCase class to define the environment and the car's start and end positions. The main script then initializes the RRTStarExploration object with the car and relevant parameters such as maximum steps, target picking probability, Dubins check interval, and maximum iterations. The search process is executed by calling the search\_path method on the RRTStarExploration object. This method returns the path and nodes in the RRT\* tree. Finally, we used the Matplotlib library to visualize the environment, obstacles, car, RRT\* tree, and the resulting path. The visualization is animated to show the search process and the path followed by the car. Overall, our implementation effectively combines RRT\* and Dubins paths to provide an efficient and safe path planning solution for a simple car model navigating through an environment with obstacles.

**Status of Code:**

The RRT\* algorithm is created and integrated to run with the 2D environment. The code performance is compared with the other available path planning algorithms to find out the which is the optimized algorithm for Dubins Path planning.

By executing the RRT\* algorithm, we obtain an optimized path from start location to goal location of the vehicle based on the lowest cost of the generated paths as per the sampling and iteration.

**Usage of Third-Party Library:**

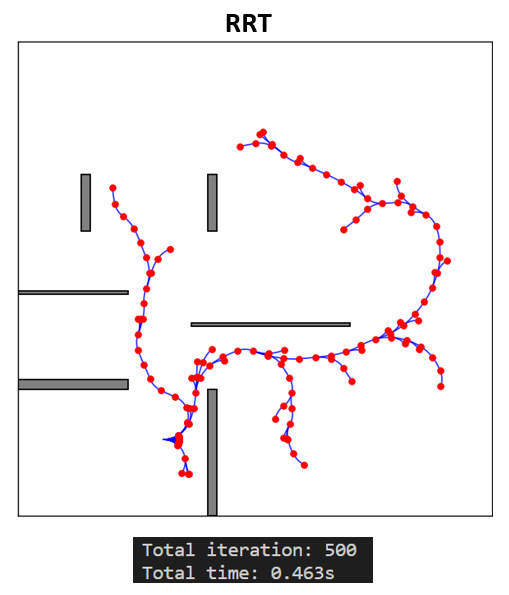
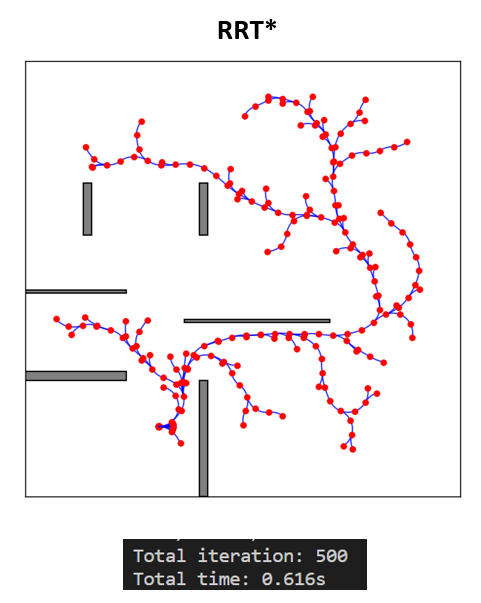
For the baseline version of the project, we have considered the environment and utility function code from this below Git path -[**https://github.com/jhan15/dubins\_path\_planning**](https://github.com/jhan15/dubins_path_planning)

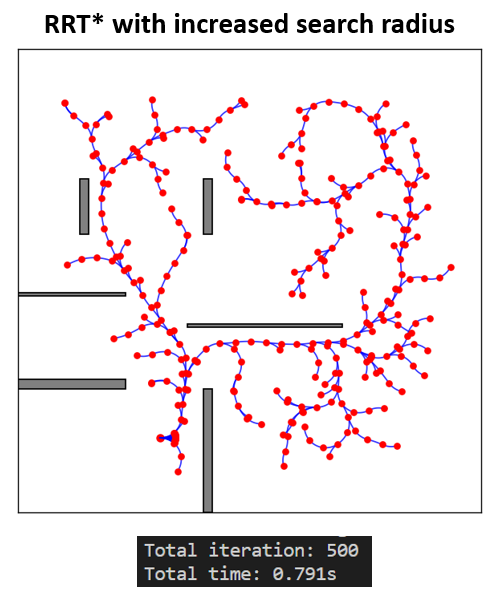
The other algorithms in the existing directory of the Git folder will be used for comparison purpose.

**Results:**

In a 2D environment, the RRT\* algorithm has been shown to produce better path and search times, making it a popular choice for path planning tasks in a variety of applications, such as robotics and autonomous vehicles. However, the performance of the algorithm can be affected by a number of factors such as the complexity of the environment and the parameters used in the algorithm, and thus requires careful tuning to achieve optimal results.

***Comparison between RRT and RRT\* exploration –***

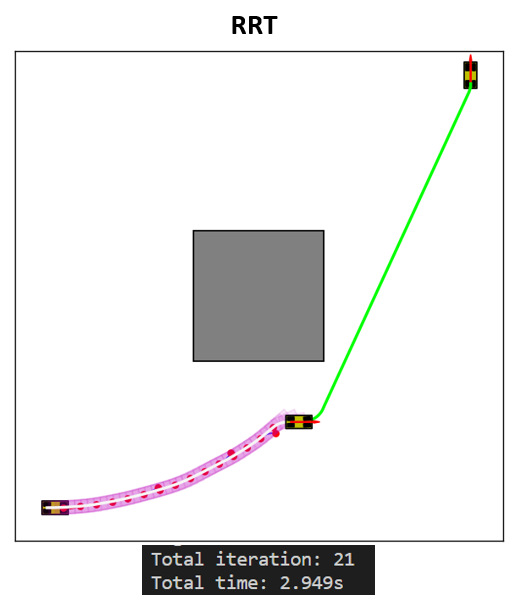
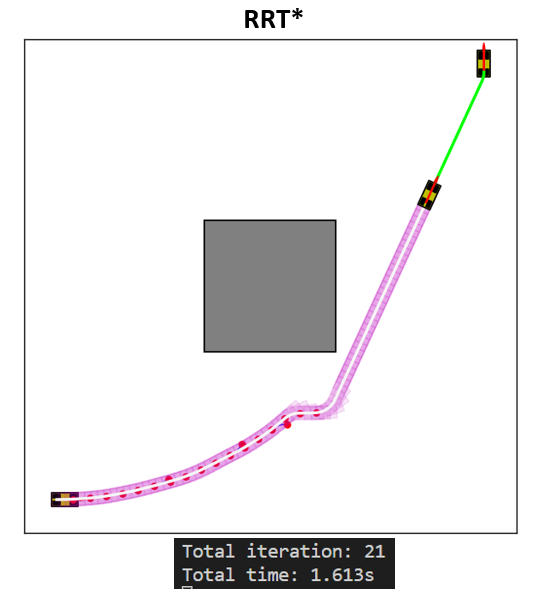
** **

****

**Figure (1): Comparison between RRT and RRT\* algorithm exploration**

|  |  |  |
| --- | --- | --- |
|  | RRT | RRT\* |
| Optimality | RRT prioritizes finding feasible paths quickly without ensuring optimality in terms of length or cost. | RRT\* improves upon RRT by using rewiring to minimize path cost as the number of samples increases, resulting in an asymptotically optimal algorithm. |
| Rewiring | In RRT, new node is connected to the nearest node without considering other nearby nodes. | RRT\* introduces the rewiring step, which updates connections to nearby nodes if the new node offers a lower-cost path. |
| Search Radius | RRT connects new nodes without a search radius | RRT\* optimizes the tree structure and minimizes path cost by using a search radius during the rewiring step. |
| Convergence | RRT quickly finds feasible solutions, but not necessarily optimal ones | RRT\* can converge to optimal solutions with sufficient samples or iterations, a bit slower than RRT due to the added rewiring step. |
| Computational Complexity | Lower computational complexity as it skips the rewiring step, making it faster but possibly leading to suboptimal paths. | Higher computational complexity due to the rewiring step, resulting in more optimal paths at the cost of increased processing time. |
| Path Quality | Produces non-smooth, longer, or higher-cost paths compared to optimal ones. | Minimizes the cost function by adding more samples, resulting in smoother and more optimal paths. |

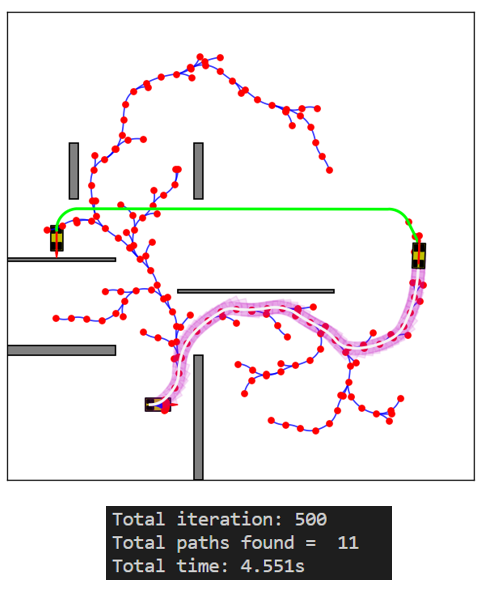
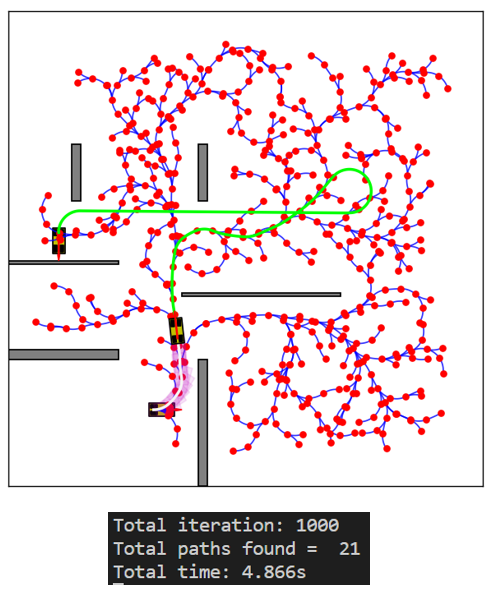
In the figure (2), we can see RRT\* outperforms RRT algorithm where the environment is a bit simple i.e, the vehicle had to move from one vertex of square to another diagonally.

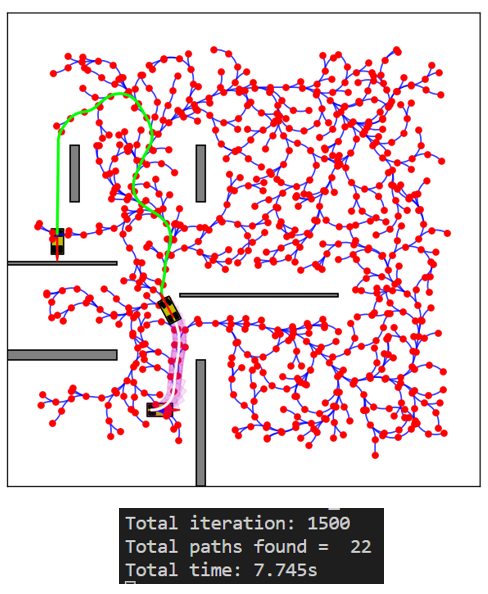
** **

**Figure (2): Comparison between RRT and RRT\* on a simple environment**

The path exploration of algorithm is mainly based on the number of iterations. It has a great influence on the percentage of the map being explored for trajectory generation.

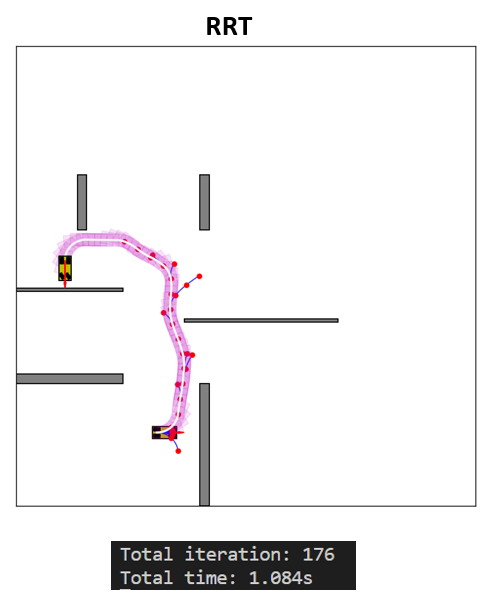
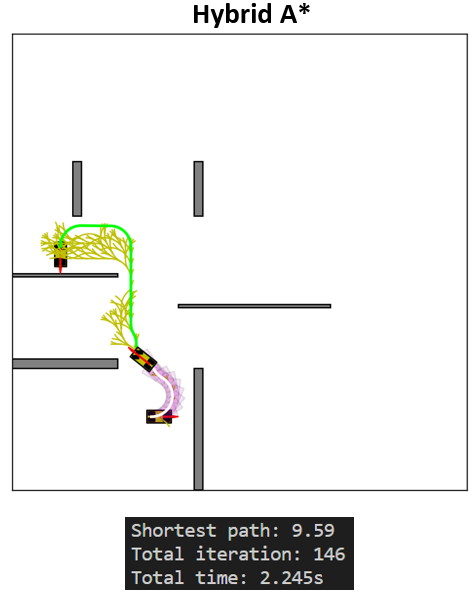
The figure (3) shows how the paths are getting optimized and efficient based on the number of iterations the algorithm is executed. *As the number of iterations increase, the algorithm has better chance to find a shorter path with the cost of computation and time.*

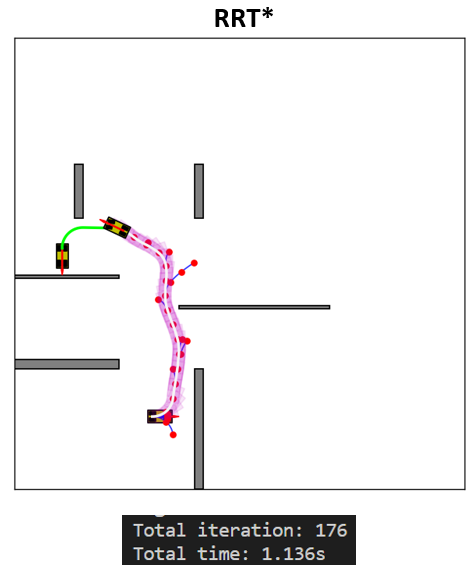
 



**Figure (3): Effect of Iteration on the execution Time and Tree exploration**

The figure (4) shows how various algorithms perform on the same scenario. As per the data collected on this instance – we are able to see Hybrid A\* having an execution time twice more than the RRT and RRT\* algorithm.



**Figure (4): Comparing Performance with other algorithms**

**Inference:**

In conclusion, the RRT\* path finding algorithm offers a great solution for motion planning in complex environments with obstacles. However, it is important to note that randomness plays a crucial role in the algorithm's performance, and removing the seed value can result in variations in the path. Additionally, running the algorithm with the same input can lead to different outputs, which has a significant effect on the vehicle's path. Despite these limitations, the RRT\* algorithm offers an effective way to find optimized paths. It is essential to consider the expense of computation and time required to achieve the optimal path, as these factors can impact the overall performance of the algorithm.

**Future Work:**

1. **Multi-Objective optimization –** RRT\* implementation should have multiple optimization constraints – such as Time to reach the goal, distance traveled and the execution time for algorithm.
2. **Dynamic Obstacle Avoidance -** To make the algorithm capable enough to handle dynamic obstacles into the environment as RRT\* is capable for adapting to changing environments.
3. **Non-Holonomic Variants Experiment -** We have planned to implement the similar algorithm in the Reeds-Shepp Car to understand how the Reverse direction have a positive impact on the final Trajectory generation.
4. **Extend to 3D environments –** Update the code to include the 3D space for the environment and expand the tree accordingly.

**References:**

1. Howard TM, Kelly A. Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots. The International Journal of Robotics Research. 2007;26(2):141-166. doi:10.1177/0278364906075328
2. Ross E. Allen, Ashley A. Clark, and Joseph A. Starek. Machine Learning Techniques for Optimal Sampling-Based Motion Planning