RRT and RRT* for Linear and Dubins Vehicles

James Holden
Department of Computer Science
University of New Hampshire
Durham, NH, 03824 USA
holdenjames96@gmail.com

Abstract

Continuous-space searching algorithms use Monte Carlo methods to generate solutions, but as a consequence suffer from a lack of optimality without further guidance. Additionally, many RRT implementations and evaluations only consider vehicles with linear motion, which don't require a minimum turning radius to change orientation, as in Dubins vehicles. This paper evaluates the gains in optimality made by implementing RRT*, and whether those gains are similar for Dubins vehicles.

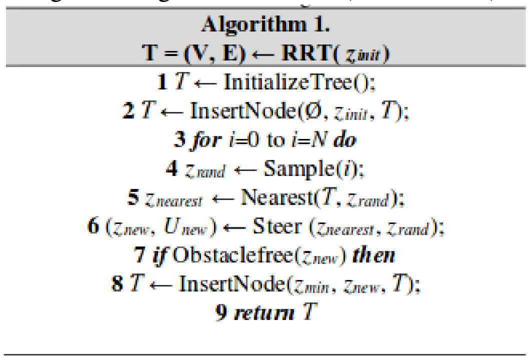
Introduction

In continuous space searching problems, algorithms that work with traditional discrete spaces are usually unable to create reasonable plans, as there lie an infinite amount of points within the workspace to consider. Without discretizing the space, some algorithms, such as RRT (LaValle 1998), solve this problem by using Monte Carlo sampling to explore and create a tree of paths to a given goal from a series of sampled points. These algorithms are able to successfully create solutions, but these trajectories are noticeably suboptimal. There exist optimized versions which seek to improve the optimality of these algorithms, such as RRT*.

RRT

RRT (Rapidly-Exploring Random Tree), developed by Lavelle, selects random samples from within the workspace and connects them to the nearest node in the existing previously explored tree, governed by euclidean distance. Following a RRT, you are guaranteed to generate a solution path to the goal, given a possible path exists and the algorithm has enough time to generate the random samples needed. However, in many solutions generated, there are actions that could be deemed too illogical or too inefficient for many real world applications.

Figure 1: Algorithm for RRT (Noreen 2016)

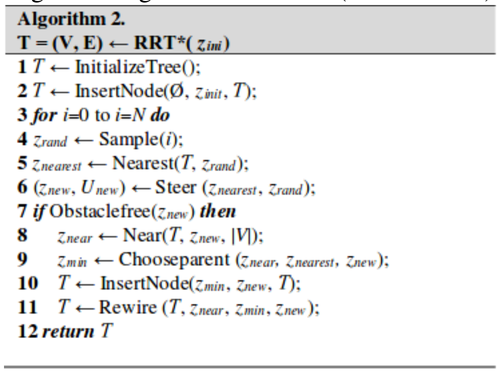


```
Algorithm 1.
T = (V, E) ← RRT( $z_{init}$ )
1 $T \leftarrow$ InitializeTree();
2 $T \leftarrow$ InsertNode($\emptyset$, $z_{init}$, $T$);
3 for i=0 to i=N do
4     $z_{rand} \leftarrow$ Sample($i$);
5     $z_{nearest} \leftarrow$ Nearest($T$, $z_{rand}$);
6     ($z_{new}$, $U_{new}$) ← Steer ($z_{nearest}$, $z_{rand}$);
7     if Obstaclefree($z_{new}$) then
8         $T \leftarrow$ InsertNode($z_{min}$, $z_{new}$, $T$);
9 return $T$
```

RRT*

RRT* seeks to improve the optimality of the solutions generated by using a different strategy to connect nodes, as well as an additional procedure to reorganize existing nodes. RRT* does this by connecting newly sampled nodes not simply to their nearest neighbor, but to their neighbor of least cost from the root node, defined as the sum of euclidean distance of all nodes from root to node. RRT* then uses the additional procedure of rewiring, in which nodes neighboring the newly added node within a given radius are rewired to optimize on the total depth from root node, considering the newly added node as a parent. The algorithm for RRT* is shown below:

Figure 2: Algorithm for RRT* (Noreen 2016)



```
Algorithm 2.
T = (V, E) ← RRT*( $z_{ini}$ )
1 $T \leftarrow$ InitializeTree();
2 $T \leftarrow$ InsertNode($\emptyset$, $z_{init}$, $T$);
3 for i=0 to i=N do
4 $z_{rand} \leftarrow$ Sample($i$);
5 $z_{nearest} \leftarrow$ Nearest($T$, $z_{rand}$);
6 ($z_{new}$, $U_{new}$) ← Steer ($z_{nearest}$, $z_{rand}$);
7 if Obstaclefree($z_{new}$) then
8     $z_{near} \leftarrow$ Near($T$, $z_{new}$, $|V|$);
9     $z_{min} \leftarrow$ Chooseparent ($z_{near}$, $z_{nearest}$, $z_{new}$);
10    $T \leftarrow$ InsertNode($z_{min}$, $z_{new}$, $T$);
11    $T \leftarrow$ Rewire ($T$, $z_{near}$, $z_{min}$, $z_{new}$);
12 return $T$
```

Many implementations of RRT and RRT* are made with the assumption that the agent that will follow the solution path has complete freedom of yaw movement. This is an inadequate assumption for vehicles that require a minimum turning radius, such as bicycles, cars, boats, etc. These vehicles have trajectories that can be modeled as in Dubins paths (Dubins 1957). Dubins vehicles are governed by a minimum turning radius, and all points along a Dubins path compose the minimum-distance trajectory between two points defined by an x, y, and theta.

The purpose of this report is to present an evaluation of the differences in performance and optimality between the RRT and RRT* algorithms, and determine if the differences are as prevalent for Dubins vehicles.

Experiment Design

To test the performance of the two algorithms, separate Python scripts were developed in order to accommodate the specific needs of linear and Dubins vehicles. Both scripts are able to select between RRT and RRT* on the command line, and accept a world on standard input, where all cells in the world are free or obstructed. The scripts use the pyFLANN library for quick nearest neighbor lookups, as it is the most computationally expensive procedure in the algorithm. A Dubins Python library was used to model and interpret Dubins paths. The minimum turning radius for Dubins paths was set to a constant of 2 units for aesthetic purposes in visualization. The scripts run the selected algorithm on their respective vehicle and output the time elapsed to calculate a solution path as well as the total distance of the solution path. Each algorithm is executed by a simple bash script to run on the same world 100 times and write the time and distance to a csv. The same process was repeated for Dubins. For debugging, visualization, and sanity purposes, the visualizer from Assignment 3 of CS730 was used to display the solution and explored paths.

Results

Below, figures 3 and 4, show typical solutions and exploration trees that are generated by RRT and RRT* for linear vehicles. Figures 5 and 6 show the same for Dubins vehicles. The agent's starting configuration is denoted by a green dot, and its goal is denoted by a red dot. Obstacles are black blocks, and white space is free to explore.

Figure 3: Typical Linear RRT Exploration

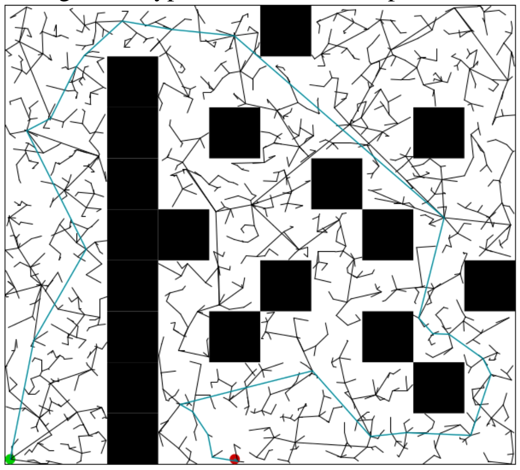


Figure 4: Typical Linear RRT* Exploration

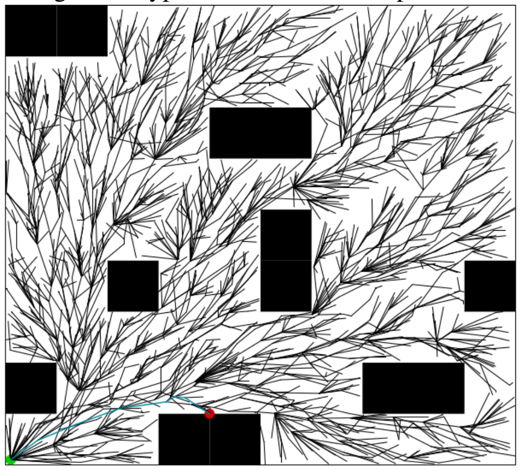


Figure 5: Typical Dubins RRT Exploration

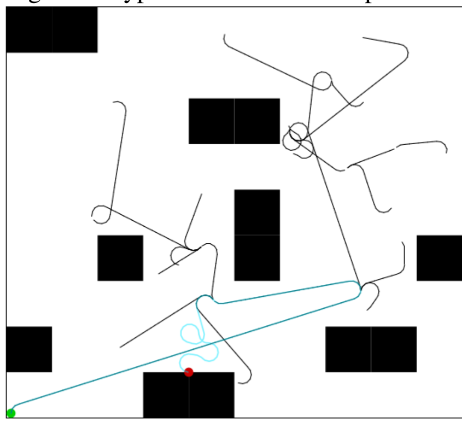
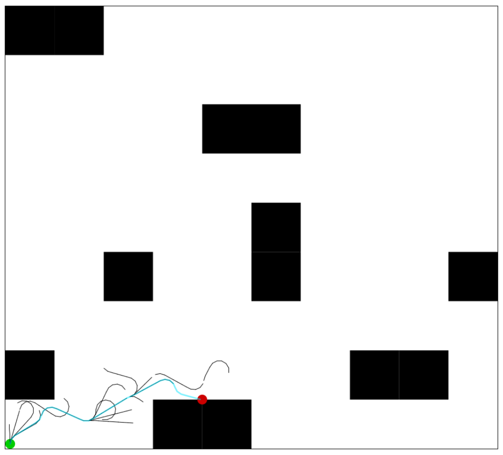


Figure 6: Dubins RRT* Exploration



See figures 7 and 8 in the Appendix for box plots that show the distributions of run times and path lengths generated for both linear and Dubins vehicles within the 100 trials of each.
This report is confident that the implementation and execution of these algorithms is correct, because the trees generated by the visualizer are valid and logical, and the results generated are well within reason.

Analysis

Solution Path Length

The empirical evidence shows that for linear vehicles, RRT* is able to generate solutions that are significantly shorter, and thus more optimal. The difference was on a scale of approximately 50% reduction in path length, from an average of 8 to 4 units. Dubins vehicles showed a very similar increased optimality; about 50% from an average of 10 to 5 units.

Time

The distributions shows that for linear vehicles, RRT* takes a few seconds longer on average to compute a valid solution than RRT, but has a tighter distribution. On this problem, RRT* is about 1 second slower. RRT and RRT* for Dubins show very similar distributions with RRT* being incrementally slower as well.

Qualitative

In visually analyzing the the exploration trees for RRT and RRT*, it is very clear that RRT* creates more organized and visually efficient solutions for both types of vehicles. RRT* is limited in that it cannot make big jumps at once, but this doesn't seem to affect its solution path lengths, as seem in the empirical results. RRT* tends to create solution trajectories that are more intuitive as well, rarely making moves that appear to be counter-productive.

Conclusion

By selecting connections that optimize for distance from root, the RRT* algorithm makes significant improvements in the optimality of the solutions it generates, at the cost of time complexity. The optimizations made in RRT* hold up for vehicles that follow Dubins paths as well. Both vehicles under these circumstances were able to reduce their path lengths by 50% and the cost of a small, but not insignificant amount of time. The paths generated seem to be much more useful in real world applications.

Future Work

As an any-time algorithm, the solution trajectory of an RRT* can be improved by taking more samples after calculating an initial solution, continuing to rewire neighboring nodes and thus optimizing the existing tree further. A future experiment could evaluate whether this improvement is a similar magnitude for Dubins vehicles as it is for linear vehicles. Additional work could compare results with other similar algorithms, such as BIT*, Informed RRT*, and RRT-Smart. A comprehensive comparison, given much more time, would involve randomly generating environments (to avoid the designer's bias) and testing each algorithm/vehicle's solution to it, recording all information and comparing.

Acknowledgements

Thanks to Wheeler Ruml for the early guidance in the development of this project.

Reference

Noreen, Khan, Habib. ”A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms” IJCSNS International Journal of Computer Science and Network Security, VOL.16 No.10, October 2016

LaValle, Steven M. (October 1998). ”Rapidly-exploring random trees: A new tool for path planning”. Technical Report. Computer Science Department, Iowa State University (TR 98-11).

Dubins, L.E. (July 1957). ”On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents”. American Journal of Mathematics.

Appendix

Figure 7: Length of Paths Generated over 100 Trials

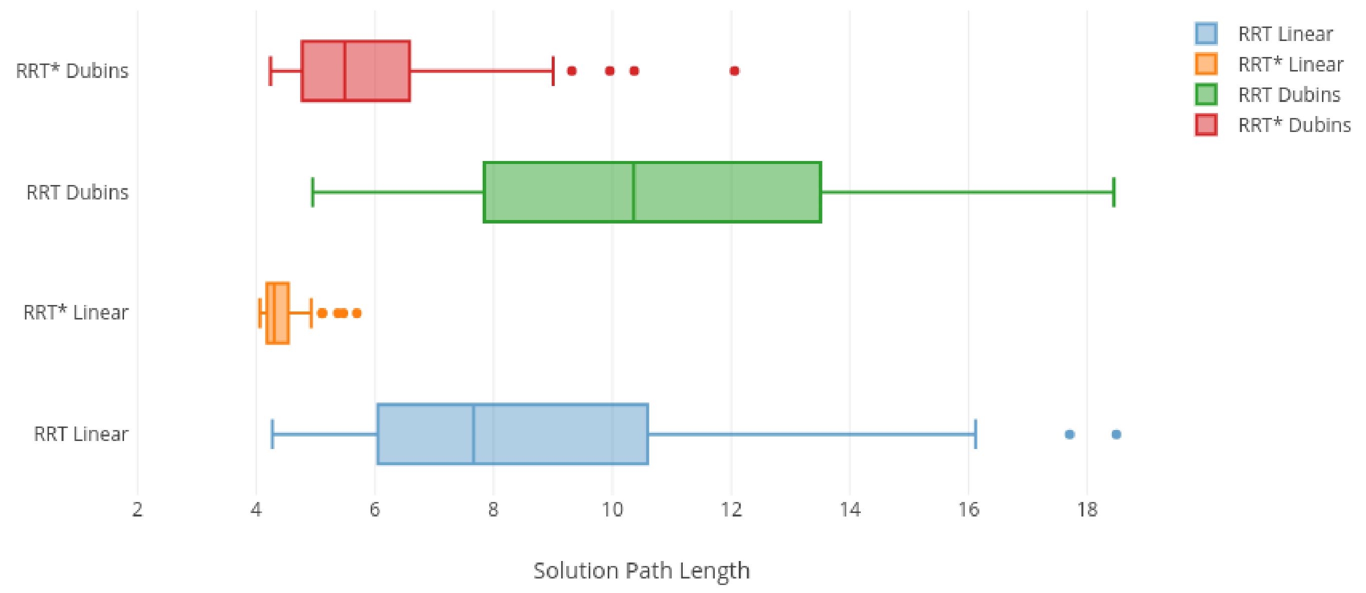
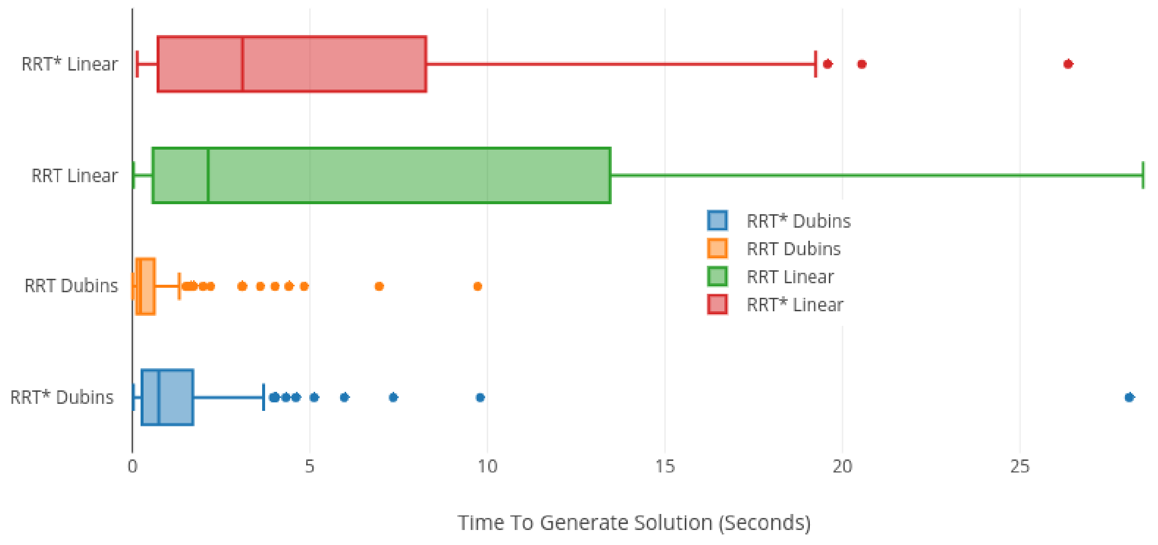


Figure 8: Time to Generate Solutions over 100 Trials


RRT* Linear
RRT Linear
RRT Dubins
RRT* Dubins
Time To Generate Solution (Seconds)
RRT* Dubins
RRT Dubins
RRT Linear
RRT* Linear