Computational Geometry - Rapidly-Exploring Random* Trees for Path Planning in 2D Environments

Adrian Bisberg
bisberg.a@northeastern.edu
August 15, 2023

ABSTRACT

One significant application for several of the techniques derived from the field of computational geometry is the motion planning problem. In the case of Rapidly-Exploring Random Tree*, a continuous space with potential obstacles and movement constraints is explored to find shortest length paths between regions of the space. In implementing the RRT* algorithm I explore several features of computational geometry and detail the specific implementation requirements of this algorithm.

INTRODUCTION

The field of computational geometry spans multiple disciplines, all of which have in common the need for geometric representations and algorithmic procedures to solve their unique set of problems. Some fields that feature usages of computational geometry include computer graphics, robotics, computer aided design, geographics information systems, and several others. Geometry problems in this domain can begin in more simple two dimensional spaces and can into higher dimensionality. After discussing some background work in the computational geometry field, I will describe the implementation of a 2D motion planner in a continuous space, including a set of cluttered obstacles, using the sample-based method of RRT*.

RELATED WORK

Following is a brief summary of several topics in computational geometry, some of which will be of particular importance when discussing my project for RRT*.

I. *Object Representation*. One of the important building blocks of computational geometry is efficient and accurate representation of different geometric objects. The basic building blocks of a geometric object are composed of vertices, edges, and their faces.
   Polygons and Triangles: Dividing most polygons into a set of edge connected triangles can be done in a process called triangulation. Using triangles can simplify many problems such as object rendering and graphics, path planning, collision detection, and mesh generation and modifications.
   Meshes: Meshes are great for representing a host of complex objects. Meshes consist of a graph structure of vertices, edges, and faces, and depending on the intended use can be stored in a multitude of ways. One common and very efficient means is called the half-edge data structure. Mesh simplification is a useful class of algorithm that reduces complexity of a mesh by removing vertices while prioritizing as much retention of key features as possible.
   Curves and Surfaces: Not all objects can be represented in discrete terms. Splines are curved paths that are formed using techniques that include using linear interpolation to create chains of Bezier curves. Curves can have different levels of continuity which at highest levels can be important for modeling and designing 3D surfaces.
   Voronoi Diagrams: An additional topic for representing space information is the concept of Voronoi diagrams, wherein space is regionalized based on a set of points, such that each region represents all points in the space closest to each point compared to any other point in the set.
II. *Object Detection.* In addition to representing objects, detecting the

positions and structure of objects is important for certain applications.

Bounding Boxes: Objects can be approximated using geometrically simplified versions, such as a cube or sphere, that are then wrapped around the object so that it is entirely contained within.

Convex Hull: The convex hull of an object is the smallest convex polygon which encloses all points of a set. Algorithms like gift wrapping and quick hull can efficiently be performed on the vertices of a polygon in order to create a convex approximation that can be used for faster detection or simpler representation.

Collisions and Intersections: In several applications of computational geometry, such as game design or path planning, determining the collision between two objects can be performed using the above mentioned techniques, so that not every small detail needs to be checked.

Shape Analysis: More complex applications of computational geometry include analyzing objects for attributes about the shapes themselves, determining if objects are symmetrical or classifying them into a detected category.

III. Motion Planning. Another useful application of computational geometry concepts is the field of motion planning. Some problems can be solved in discrete spaces, like a grid or a roadmap of an area which represents the space in a graph structure. Algorithms such as Dijkstra's and A* are used to solve motion planning problems in these graphs.

Alternately to these discrete based problems are motion planning in continuous space. These problems use sample based methods for planning, which will further be discussed as the topic of this paper in the next section.

IV. Transformations. Once objects can be stored and displayed, transformations can be applied to shapes to modify their position and appearance. These transformations include scaling, shearing, rotations, and reflection.

PROJECT INTRODUCTION

In order to further explore some of these topics in computational geometry, I have implemented a path planning algorithm called the Rapidly-Exploring Random Tree* (RRT*), where the asterisk or "star" indicates additions made to the algorithm to find the shortest path solution to a search query in the space. The RRT* search will take place in a continuous 2D environment with polygonal obstacles scattered throughout. The best path will then be computed and visualized from a starting position into a goal region, defined by a goal point and radius of acceptable locations.

RRTs are a sample-based planning method introduced by LaValle and Kuffner which are a subset of Rapidly-Exploring Dense Trees.[1] In the case of RRT, they use random distribution as their form of sampling new points during exploration. Unlike path planning algorithms like A* which work on discrete graphs,an RRT searches directly in a continuous space. Due to the nature of RRT uniformly sampling the search space, the search becomes biased into the largest Voronoi regions of the space, a helpful property that naturally occurs in the tree without having to explicitly construct a Voronoi diagram.

The basic RRT search consists of a simple procedure: sample the search space for a new location, find the nearest node in the existing tree to that location (if obstacles and or constraints are present it is ensured that this path is obstacle free and reachable within constraints), then the new location is connected to the existing node in the tree. Often, the growth factor of the tree is limited through a sub-procedure called "steering" which ensures that newly sampled points are within a set step size away from existing points

in the tree, allowing for more incremental growth.

The RRT* algorithm adds two additional steps to optimize for shorter path lengths, both involving inspecting a set of vertices that can be called a “neighborhood”, which are existing vertices in the tree that are within a certain radius from the newly selected point. When adding a new point, each neighbor vertex is checked to see which provides the shortest cost (cumulative distance from the start) then that vertex is chosen as the parent. Following this, the RRT is rewired, checking each neighbor again this time to see if a shorter path can be achieved by traveling through the newly added node, and each vertex that has a shorter possible path is updated (along with its children’s path costs).

EXPERIMENTAL SETUP

To implement the RRT* I wrote code in C++ using the SDL2 library to draw the obstacle space and visualize the determined tree. For each obstacle space, the program considers a text file with a list of polygons specified by each point in a counter-clockwise orientation. For the sample environments I utilized ChatGPT v. 3.5 to generate a list of random polygons in the 2D space to provide a diverse set of obstacles for the tree to navigate around.[3] Configurations of the start, goal, space size, and specific tuning parameters for the algorithm can be set in the main.cpp file for alternate problems to be solved using the RRTStar class.

IMPLEMENTATION DETAILS

First, for creation and collision detection of the obstacles, each polygon is triangulated using the ear clipping algorithm, then each triangle gets checked for intersection with a point or segment depending on the required test. The triangulation is only performed once then the associated triangles are stored for collision checking.

There are also several sub-procedures that the RRT* algorithm depends upon:

I. Point sampling: Each sampled point was determined by a random integer in the range between 0 and the maximum x or y coordinate of the search space. it is then tested to be outside any obstacles.
II. Nearest-neighbor detection: Given the generally smaller scope of this project, the nearest neighbor detection is performed in linear time to the number of vertices, where each vertex is checked for closest distance to the new point. The same method is used for neighborhood detection, where each vertex is checked to see if it is within the neighborhood radius of the new point. Optimized methods for this procedure are discussed in the future work section.
III. Steering: Adding a steering function ensures that newly added points are within a set step (or ‘rho’) distance away from the existing tree. To implement, the directional component is found between a random point in free space and the nearest point to that point in the tree. Then a new point is found at distance rho from the nearest point in the chosen direction.This will be the new vertex added to the tree, provided there is an obstacle free path.

One additional implementation choice was how to store the RRT tree structure itself. For this, I chose to create a custom Node type that stores auxiliary information for the parent, children, and cost at the vertex, then stored the vertices in a vector structure. In the Future Work section I discuss more optimal data

structures for storing the tree that would be more efficient for higher complexity spaces.

Aside from these subprocedures the rest of the RRT* algorithm follows this basic pseudo code.[2]

```
RRT*(start) returns T(V,E)
T ← InitialzeTree()
T ← InsertNode(start, 0, ∅)
for i in 0 to m:
  v_rand ← Sample();
  v_nearest ← Nearest(T, v_rand)
  v_new ← Steer(v_rand, v_rand)
  if ObstacleFree(v_new):
      N ← NearestNeighbors(v_new)
      parent ← ChooseParent(N,
        v_nearest, new)
      T ← InsertNode(v_new, parent.cost
        + Distance(parent, new),
        parent)
      T ← ReWire(T, N, v_new)
 return T
```

EVALUATION

The final algorithm achieved by my implementation is able to find very efficient means of traversing the free space between the start and goal region. In the tests presented below on optimal path length, after tuning parameters, the best path cost was able to come very close to the euclidean distance between the start point and goal region, where in the test I ran the euclidean distance (ignoring obstacles) was 738 and the algorithm was able to traverse a path in an average of 757 units.

There are two influential parameters that can be tuned that will affect the efficiency of the path found and the time it takes to do so: the step (rho) value and the radius of the neighborhood for consideration.

I performed some parameter tuning reported in Tables 1 and 2. The tests performed used obstacles defined from the 'large_spread_out.txt' file and traveled from the start point at 10,10 to the goal region centered at 620, 460 with a radius of 20. The RRT* pathfinding was run 1,000 times and averaged across all trials for the values reported in the tables.

In table 1, it can be seen that increasing the neighborhood radius incrementally shortens the best path found. The property is due to an increased number of tree nodes being considered for an update at each new addition, making it much more likely for new and improved connections to be made where shorter paths are possible. Where possible, the algorithm will usually choose a parent node closer to the start location, so a more direct diagonal can be traversed to create a shorter cost path.

Generally, the expansion of the neighborhood radius increases the execution time of the algorithm quite significantly, up until a certain threshold around the size of 70 where having the higher neighborhood radius quickly decreases the execution time. The decrease in time coincides with produced trees that are much smaller (less exploration had to be made in space) by the time the goal region was found which most likely accounts for the dramatic decrease in execution time. This is likely due to the fact the tree is able to explore the space in fewer iterations when the influence of the existing tree has less limitations on its growth. While not accounted for in the analysis I performed, if the tree were to grow to a set size specified by a max iteration, as opposed to running until a goal state was reached, I suspect the overall time spent on execution would remain linear as the neighborhood radius increases, as the amount of checks and updates are likewise increasing linearly.

Tuning the step size similarly has a trade off of shorter paths versus longer execution times, where this time the shorter step sizes find

significantly better cost paths, with high incursions on the time to complete. The larger step size prioritizes exploration of the space, helping the goal region be identified much more quickly, however this comes at the cost of a less efficient path length, as larger jumps are being made around the space.

Step Value	Neighborhood Radius	Path Length*	Time Spent* (μs)
30	40	847	76,402
30	50	812	86,338
30	60	790	92,643
30	70	779	25,236
30	80	773	28,126

Table 1: Parameter tuning of the neighborhood radius that is examined for the RRT* shortest path solver.

* Averaged over 1,000 trials

Step Value	Neighborhood Radius	Path Length*	Time Spent* (μs)
10	60	757	130,461
20	60	772	126,201
30	60	792	72,640
40	60	812	71,346
50	60	840	34,998

Table 2: Parameter tuning of the step size that is used for new sampled points.

* Averaged over 1,000 trials

In tuning the parameters, it can also be seen in the shapes of the trees produced how changing the step size and neighborhood radius affect the overall shape of the tree produced. In Figures 1 and 2, the lowest and highest values for neighborhood radius tested are compared. The smaller neighborhood radius in Figure 1 looks much more similar to a non-optimized RRT with more jagged branches, while the larger neighborhood radius produces a tree that looks significantly more linear, showing how the optimized distances influenced the paths to produce more direct branching from further nodes. Likewise the step size comparison in Figures 3 and 4 shows the trade off between efficient paths found with shorter step sizes, and less iterations required with larger step sizes where exploration is prioritized.

The lower step size does not appear as highly correlated with more efficient path lengths as the increased neighborhood radius achieves. There are many regions that are not as well explored and given goals in those areas, the path length found would be worse than that with the higher step size.

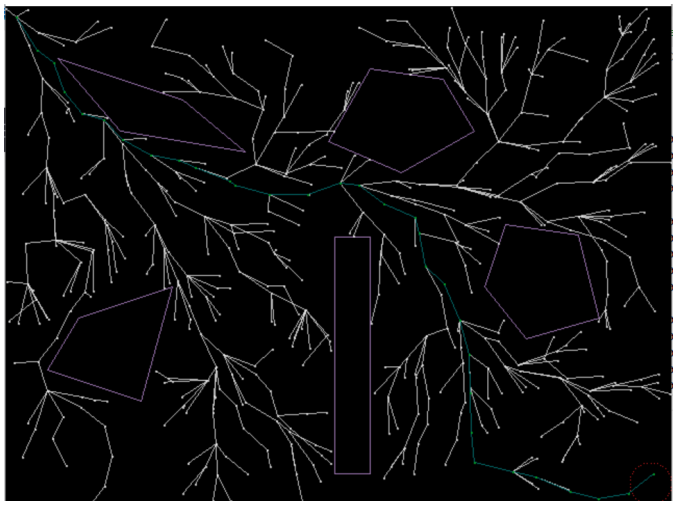

Figure 1: Tree generated with neighborhood radius of 40 (and step size of 30)

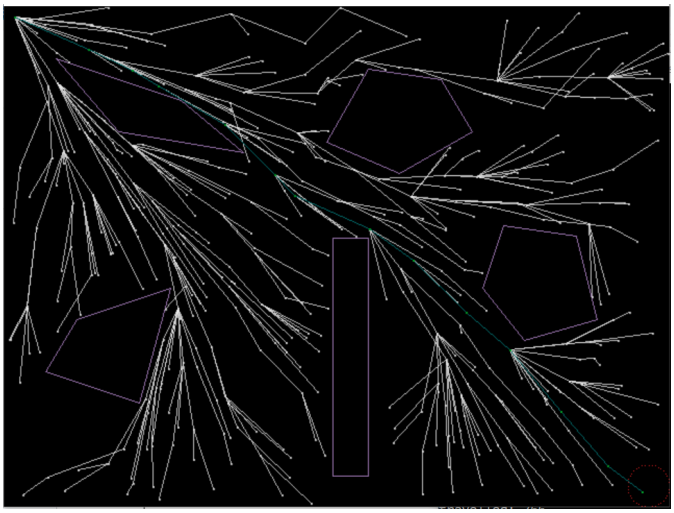

Figure 2: Tree generated with neighborhood radius of 80 (and step size of 30)

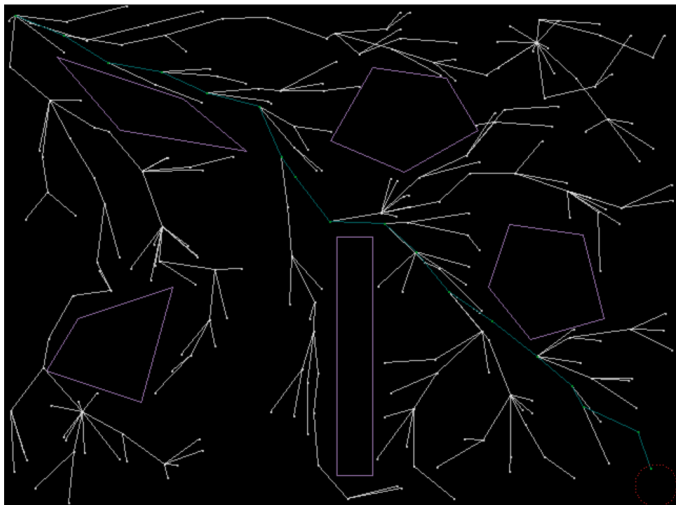

Figure 3: Tree generated with step size of 10. (and neighborhood radius of 60)

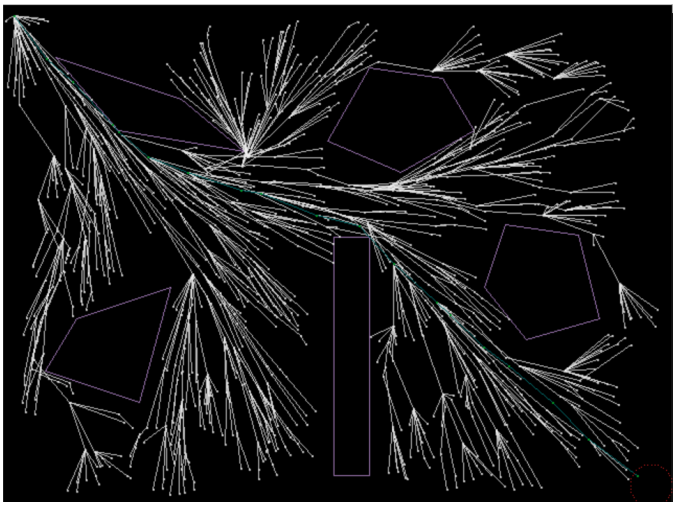

Figure 4: Tree generated with step size of 50 (and neighborhood radius of 60)

CONCLUSION AND FUTURE WORK

Overall, the RRT* algorithm was a straightforward to implement option for motion planning shortest paths in a continuous environment. There are several directions from my implementation where optimizations can be added to improve performance, especially if the desired use for the planner is in more complex and larger environments, or faster real-time applications.

The two most time-consuming procedures in the project are the object collision checking and the nearest neighbor search. Both can be improved by choosing more optimized data structures for storage, either through custom implementation or provided libraries.

One option to speed up collision detection would be to store triangles in an R-tree, which is a data structure designed for efficient spatial access. Coordinates can be stored in a

balanced manner and in hierarchical means based on minimum bounding boxes. Using this data structure would allow the algorithm to reduce the number of polygons that need to be checked for collision by only selecting those within a significant region close to the new point being checked.

To speed up the nearest neighbor search, the RRT could be stored in a KD-tree. This is a structure that generalizes a binary search tree into multiple dimensions. This would benefit the run time of both the search for the nearest node in the tree, as well as finding the nearest neighborhood of nodes.

Additional capabilities of the RRT* algorithm could also be explored by constraining the possible movements of the agent along the path. Rather than allowing for complete freedom of movement in 2D space, the agent could be constrained by a turning radius (similar to a car) or given other limitations on the types of movements it could make.

A final aspect that I would like to implement as part of the project would be smoothing out the determined paths, for example using splines formed between the waypoints, to create a smooth path between start and goal as opposed to the jagged turns made.

REFERENCES

[1] LaValle, Steven M. *Planning Algorithms*. Cambridge University Press, 2006.

[2] Karaman, Sertac, et al. “Anytime Motion Planning Using the RRT*.” *2011 IEEE International Conference on Robotics and Automation*, 2011, https://doi.org/10.1109/icra.2011.5980479.

[3] Text of : “Generate an obstacle course in 2D space by creating a list of simple polygons that represent obstacles. The minimum x and y coordinates are both 0, the maximum x coordinate is 640, the maximum y coordinate is 480. The obstacles should not be overlapping and can be a mix of many sizes and many degrees of complexity. To report the obstacles, list each point as a space separated 'x y' coordinate on each line, then use an empty line to denote the end of a polygon before the next.” prompt. *ChatGPT*, Default (GPT-3.5) version, OpenAI, 10 August 2023, chat.openai.com.