# CS460 HW2 Report

Girish Ganesan - gg655

November 5, 2021

## Problem 1

### Collision

The collision detection algorithm I used is based on the separating axis theorem for convex polygons. Given the robot polygon and an obstacle, I first compute the orthonormal vectors for each of the edge vectors of both polygons, by first finding the normal and subsequently normalizing the length of each vector to 1. Then, I project the coordinates of the obstacle and robot vertices onto each of these normals in turn, looking for a case in which the projections (which are scalar values) of all the robot vertices are greater than those of all the obstacle vertices or vice versa. This would indicate separation along some axis, which is only possible if the convex polygons are non-overlapping.
If all the candidate normals fail this test, then we conclude that the robot and the obstacle are overlapping. The same test is done for each of the obstacle in turn, and if even one of them fails the non-collision test, the function returns because a collision has been detected.
After considering all obstacles, if no collision has been detected (there has always been a separating axis between each of the obstacles and the robot), then the function returns true (that the current state is collision-free).
I tested this function by using sample() to spawn a large number of copies of the same robot in different configurations and colored them according to their collision status (Fig 1).

### Tree

Many of the following methods rely on a BFS search through the tree for some node with a desired property. This operates by initializing a queue with the root node of the tree. While this queue is nonempty, the first element is popped off, examined for some criteria, and if we need to continue searching, the queue is extended with the list of children of the current node being examined. In this manner, we are able to paradigmatically look-up every node in the tree exhaustively.
Additionally, the Tree structure is implemented as a TreeNode saddled with additional metadata about the specific problem (including the start state, goal

state, and the obstacles in the environment). Each TreeNode contains a state variable (its value) and a list of TreeNodes that are its children. In RRT, a node's child is any node such that there exists an edge from the node to the target.

1. Add: Initiate a BFS search for the parent node (point1). Once the node is found, add the child (point2) to the parent node's list of children.

2. Exists: Initiate a BFS search for the desired point and return when found.

3. Parent: Initiate a BFS search. At each stage, examine the children list of the currently visited node and determine if the target node is in the list. If so, return the current node.

4. Nearest: Initiate a BFS search through the tree. Initialize variables representing the nearest node and its distance from the target to None and $\infty$, respectively. When each node is examined, check if its distance from the target as computed by the metric function is less than the current minimum. If so, replace the current minimum with this new node. At the end of the search, return the global minimum of all nodes currently in the tree.

5. Extend: I implemented extend using a discretized approach. A high-level parameter for this algorithm is the step size. First, the step towards the new node is broken up into horizontal and vertical components. The robot is checked for collisions at each point between the start and the new nodes with horizontal and vertical offsets from the initial state calculated as mentioned above from the step size. Once a collision is detected or the total sum of step distances adds up to more than 1, the last safe position is added to the tree.

## RRT

My implementation of RRT begins by initiating a Tree with the start node. Then, for the number of iteration specified by iter_n, random points are sampled from the workspace. The nearest point in the existing tree to this new random point is determined. Then, the extend function is called to connect the nearest point at most 1 unit in the direction of the random point, as long the random point represents a valid configuration for the robot.
After the tree is constructed, the path is recovered by backtracking along the branches of the tree using the parent function. At each iteration, starting with the goal node, the parent of the currently considered node is determined and the current node is added to the end of a growing list. Once the start node is reached, the path-growing process terminates and the algorithm outputs the list created in reverse order (to reflect the transition from start to goal node).
As mentioned before in the explanation of the extend function, one of the parameters in my implementation of RRT is the step size, which determines how often we check for collisions while extending out from the source node. I found

empirically that a step size of 0.1 worked well for the environment bounds we were given. The following results are measured with this setting.
The RRT implementation works very well when given an adequate amount of iterations. However, as seen in the graph below, increasing the number of iterations for an environment of given complexity reaches diminishing returns beyond a certain point (Fig 2 and 3).

## Visualize

1. Visualize Path: The obstacles are drawn. Then, the robot is drawn at each state in the path is drawn, and a residual connection is made from the previous state to the new state to show the path ordering.

2. Visualize Configuration: I chose to discretize the space into a grid of $D^2$ points, where $D$ is an adjustable parameter for resolution. At each grid point, the algorithm checks if the robot is in collision and colors that grid square accordingly (white if collision free, black if colliding). One of the byproducts of this technique is that the rough outline of each polygon's Minkowski sum with the robot is shown, irrespective of the convexity or non-convexity of the obstacle. Increasing the resolution would result in a quadratic-time speed delay, but it would make the configuration space visualization more fine-grained (Fig 4).

3. Visualize RRT: This algorithm first visualizes the configuration space as presented in Visualize Configuration. Then, the full RRT roadmap is displayed on screen, after the entire tree has been generated. A BFS search is used to accomplish this, which children being connected by an outgoing edge from their parent (Fig 5). If a path was found from start to goal, it is outlined in green.

## RRT Star

Two functions are implemented in the Tree class for $RRT^*$ to work.

1. Get Cost: The get_cost method works by climbing up the tree from a leaf node all the way to the root. At each step, a running counter of the cumulative cost up to that point is updated with the cost of the path from the current node to its parent (i.e. the metric distance). The process is then repeated for the parent node, and so on.

2. Rewire: The rewire function works by first locating all the nodes in the tree within a r-neighborhood of the target node by BFS search. Then, the nodes are sorted by the cost of a path from the start node to the given node to the new node. The lowest-cost node is connected to the new node in the main $RRT^*$ algorithm, so it is ignored in the rewire function. For all other nodes, if the cost of a path through the target point to the current node is less than the cost of the existing path to the current node,

the current node is removed from its old path and added as a child of the newly added node.

A high-level parameter of this algorithm is the value of $r$, which should ideally be chosen to isolate a $\log n$-neighborhood of the new node (where $n$ is the number of nodes current in the tree). For testing purposes, I set $r = 2$ which performed well qualitatively for the environments and problems I generated.

Visualize RRT-Star is a similar function to the one used for the regular RRT visualization. The only change is that now the tree is grown by the $RRT^*$ algorithm instead of the RRT algorithm.

The RRT-Star success rate for the same conditions and environment as the RRT algorithm is shown below. The success rates are quite similar across the iteration counts on average (Fig 6). The RRT-Star path length, on the other hand, falls somewhat steadily unlike regular RRT. A typical example is shown below. After the path is initially found, RRT-Star's completeness demonstrates that it will find the optimal path given enough time, so this downwards trend in path length is an expected result (Fig 7).

## Problem 2

### Robot

The transform function is calculated by using the transformation matrix for a rotation followed by a translation. This is executed in numpy, before the relevant physical parameters (x and y coordinates in real space) are extracted from the resulting vector for each of the four vertices of the robot.

### Sample

The sampling method I used was uniform sampling for both the spatial coordinates and the angular coordinate. For the latter, I rescaled a random number between 0 and $2\pi$ to be between $-\pi$ and $\pi$ by subtracting $\pi$ after the initial multiplication by $2\pi$.

### Tree

Most functions in the Tree class remained the same with minor modifications. Nearest and Extend were the only two that certainly changed perceptibly. I introduced an angular distance metric that calculates an offset between $-\pi$ and $\pi$ between two angles. This was used in the updated metric function for nearest. Furthermore, I had to introduce a stepping variable for the angle $\Theta$ based on the step size for the spatial coordinates $x, y$ such that once the extend function reaches the spatial coordinates of point2, the value of $\Theta$ would have also reached the angular coordinate of point2. In other words, the challenge was to determine a smooth sloop by which the angular parameter could evolve linearly with the x and y parameters.

### RRT

All the changes with the RRT algorithm occur within the Tree class. Success rate against iterations are plotted again, with no major surprises (Fig 8).

### RRT-Star

Results are shown in the following figures. Success rate tops out at a lower percentage than in the previous problem, likely due to the added complexity of the environment I created to test the more powerful translation-rotation abilities of the car as opposed to the robot from the first problem (Fig 9 and 10).

### Visualizer

Visualize Path was implemented by drawing each state of the robot along the path and connecting consecutive states with lines (Fig 11).

## Problem 3

### Robot

1. Kinematics: Given a state and a control vector, the components of each are broken down and isolated. The expression returned precisely follows equation (1) in the assignment description.

2. Propagate: The list of states that comprise the trajectory is initialized to the singleton list of just the start state. Then, for every control, for every timestep in the duration of that control, we consider the previous state (the last one in the trajectory list) and determine the next state by applying Euler integration on all the state components using the time derivatives supplied by the kinematics function. At the end, for a sequence of durations $[n_0, n_1 ... n_m]$, we will have $1 + \sum_{i=0}^{m} n_i$ states in the trajectory, including the start state.

### Tree

The major update to the Tree class is the modification of the extend algorithm. Now, extend takes into two parameters n1 and n2 that determine the range of possible duration for a given control. Empirically I found that values between 10 and 30 worked best for my application, so I used these for my n1 and n2 values. Additionally, the control itself is randomly generated. I chose to bound the velocity between 0 and 1 and the angular velocity between $-\pi$ and $\pi$.
Once these parameters are set, the random shooting algorithm attempts to solve

the boundary value problem implicitly described by the motion planning setting. Specifically, it first chooses a random collision-free sample in the space and identifies the closest node in the existing tree to the sample. Then, the algorithm chooses a random action (the control) to perform for a random duration from that nearest point. The last collision-free state in the resulting trajectory is added to the list.

## RRT

The RRT algorithm again suffers a decrease in performance in the early stage, but recovers well once given an adequate number of iterations to accomplish this more complicated task (Fig 12 and 13).

## Visualize Trajectory

This function is similar to the Visualize Path function in the previous problem. The only difference is that the collision-free states at every timestep are calculated and plotted, not just the last collision-free state like in a conventional extend method.

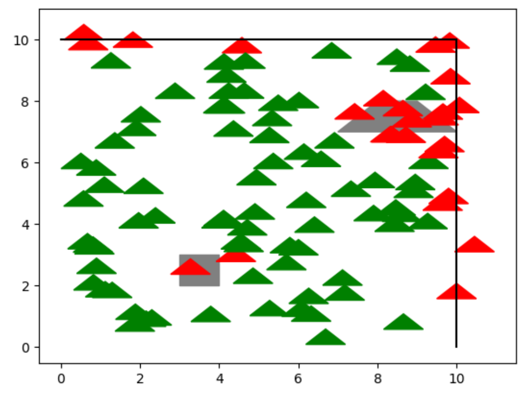

Figure 1: Collision checking with obstacles and boundaries

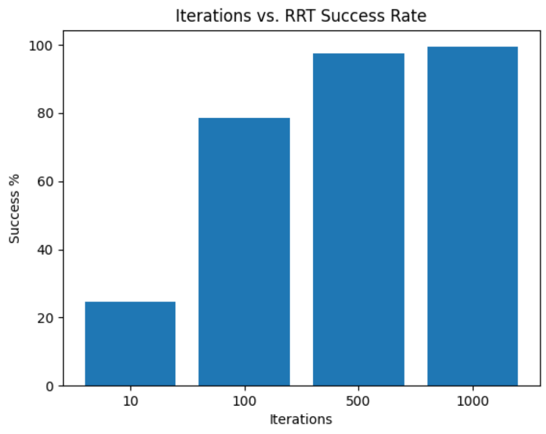

Figure 2: Success rate of RRT plotted against number of iterations allowed

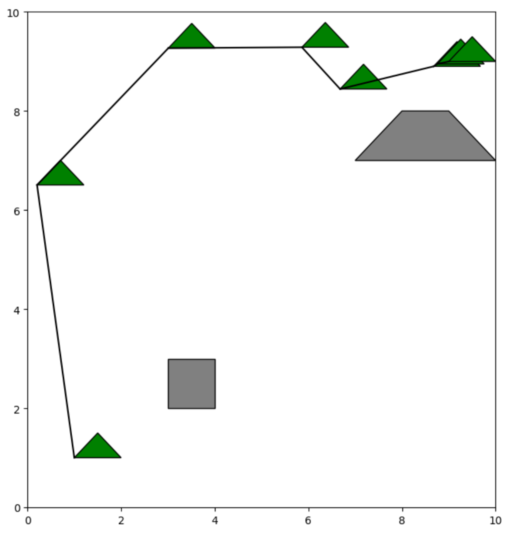

Figure 3: Visualize Path

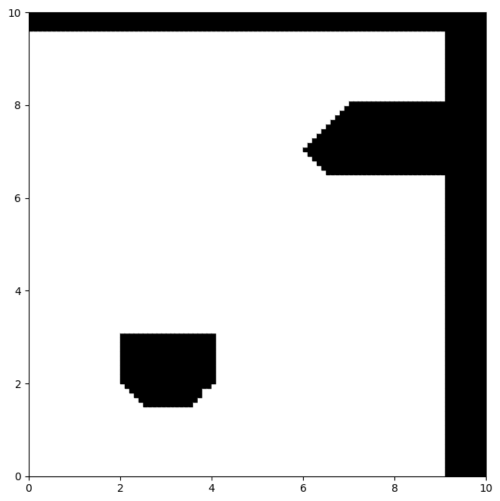

Figure 4: Visualize Configuration

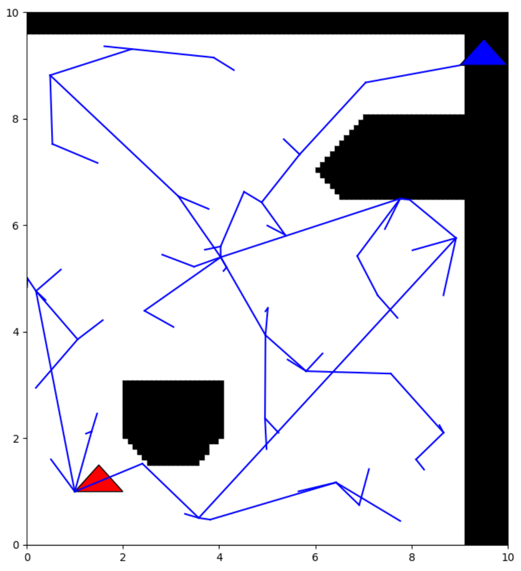

Figure 5: Visualize RRT

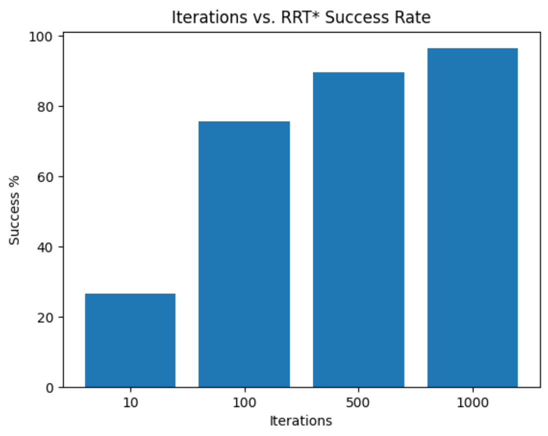

Figure 6: RRT-Star Iterations vs. Success Rate

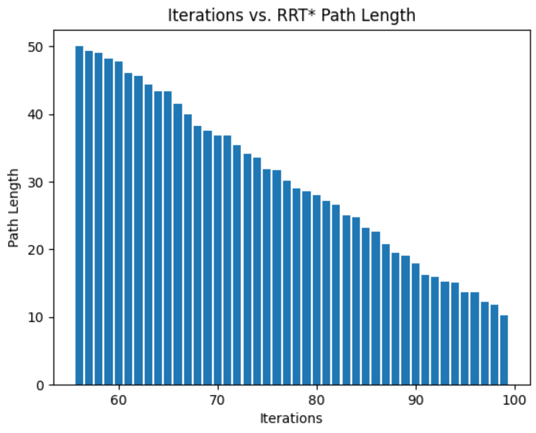

Figure 7: RRT-Star Iterations vs. Path Length

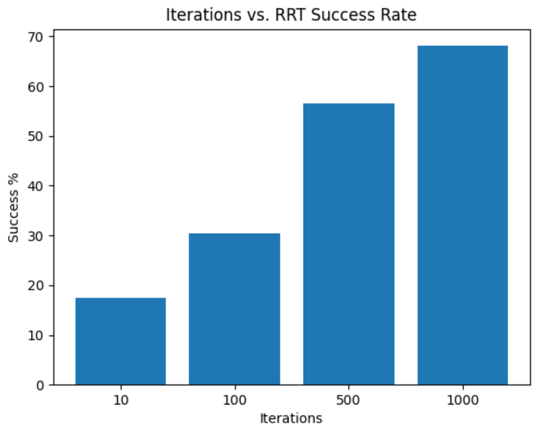

Figure 8: RRT Iterations vs. Success Rate

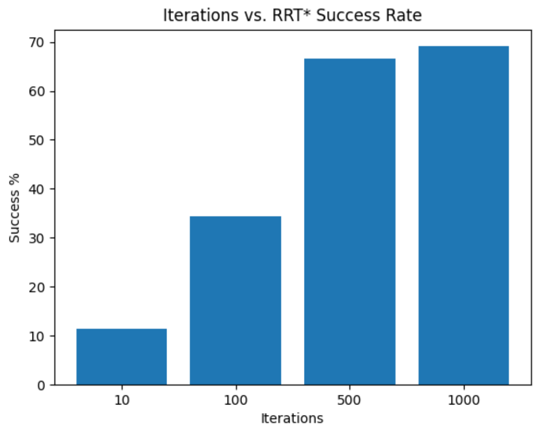

Figure 9: RRT-Star Iterations vs. Success Rate

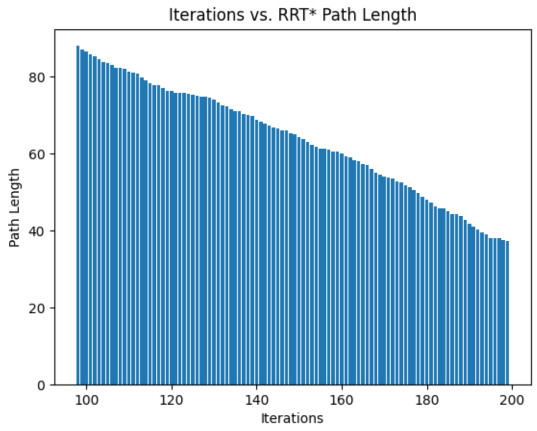

Figure 10: RRT-Star Iterations vs. Path Length

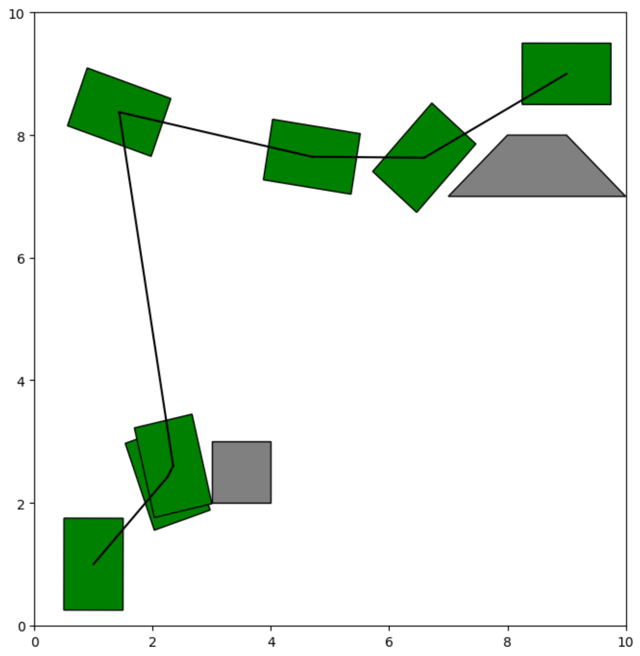

Figure 11: Visualize Path

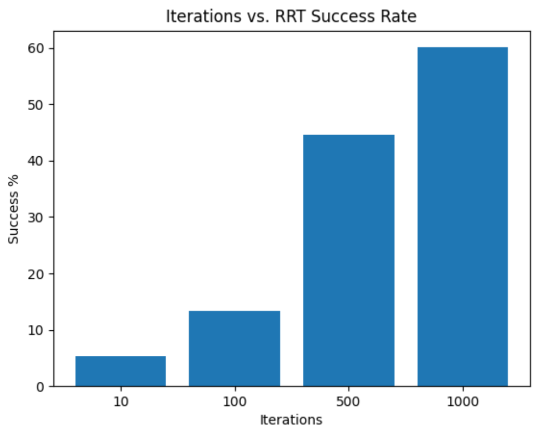

Figure 12: Iterations vs. RRT Success Rate

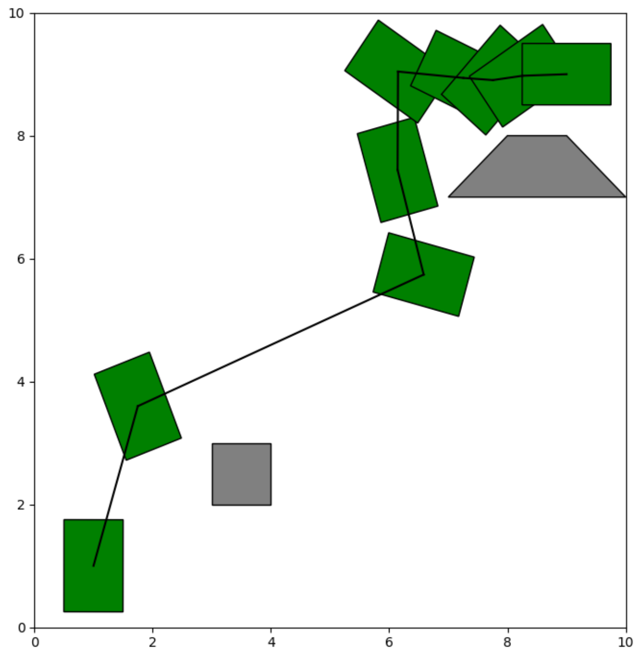

Figure 13: Visualize Trajectory