Anna Matusewicz
CS 76 Prof. Balkcom
11/16/17

Design Notes

ArmRobot

Realies on shapely and annoy libraries. Takes as an input the arm legths of the robot, the initial joint angles, a list of the obstacles (shapely objects), a maximum number of random configurations, and a number of neighbors. The class contains a useful method to calculate the positions of all the joints of the arm from the list of the current angles using sine and cosine.

build_roadmap with Annoy

I build the roadmap by first creating a list of all the randomly generated arm positions and the starting and ending positions, kept in `self.random_points`. Then, I add them all to `AnnoyIndex` using the "euclidean" way of finding neighbors. I then use `t.build` to build 10 trees.

```
t = AnnoyIndex(len(self.random_points[0]), "euclidean")
for i in range(len(self.random_points)):
    v = self.random_points[i]
    t.add_item(i, v)

t.build(10)
```

Finally, I iterate through each point in `self.random_points`, getting the set amount of neighbors using Annoy each time, and add each edge to the `self.road_map` dictionary as an undirected edge. One interesting observation is that the higher k is for a k-PRM, the smoother and faster the solution is. I believe this is because more neighbor options means more educated decisions about what the next configuration should be.

collides

If only one list of angles is provided I create a `LineString` using shapely for the robot and utilize the `.intersects()` method of the library in order to tell if a certain configuration would collide with an

obstacle. If two list of angles are provided I create `LineString` 's between the corresponding joints to test whether or not the path between two configurations collides with an obstacle

```
for obs in self.obstacles:
if arm.intersects(obs):
    return True
```

A-star Search

This is the exact same class as used in `Mazeworld` the only notable difference being I use the angle distance between a node and the goal as a heuristic, similar to the Manhattan Distance from the previous assignment.

```
def angle_distance(self, angle):
    dist = 0
    for i in range(len(self.random_points[angle])):
        dist += abs(self.random_points[angle][i] - self.random_points[1][i]) % (2 * mat

    return dist
```

DrawModel.py

I have my draw function for ArmRobot split up into 3 methods: draw, draw_config, and draw_answer. `draw` depicts the starting state of the arm and the obstacles and is mainly used for testing. `draw_config` depicts all the random points in the space, connected according to which is neighbors with which. Additionally the start and end locations are marked. `draw_answer` animates the solution by utilizing a `step` method.

Mobile Robot

Realies on shapely and numpy libraries. Takes as an input the start_state, goal_state, and a list of obstacles (shapely objects). There are three small helper methods in this class: collides, get_euclidean, and backchain. `collides()` accepts two points, first it checks to see if they are in the test space, then it checks whether or not the points or the transitions between the two causes any collisions:

```
l = LineString([(point1[0], point1[1]), (point2[0], point2[1])])
for obs in self.obstacles:
    if l.intersects(obs):
        return True
```

If so for either, the method returns true.

`get_euclidean()` returns the distance between a point and the goal point by utilizing the `.distance()` method from shapely.

Finally, `backchain()` simply returns a path from the start_state to the goal_state by traversing through `self.bp` then reversing the list.
If there is no result it returns `None`

RRT

For the number of random points allowed, my RRT begins by creating a random point then picking the closest point on the tree that does not cause a collision between teh two (useful in cases of thin walls). If a chosen point is within a distance less than the set duration for branching, the branching duration is shrunk to increase chances of being near the goal point. Then, for each control, the chosen point is branched upon using:

```
single_action(t, controls_rs[j], md)
```

Each new location, if it has not already been expanded upon and is valid, is then added to the tree and given a backpointer to the previous point.

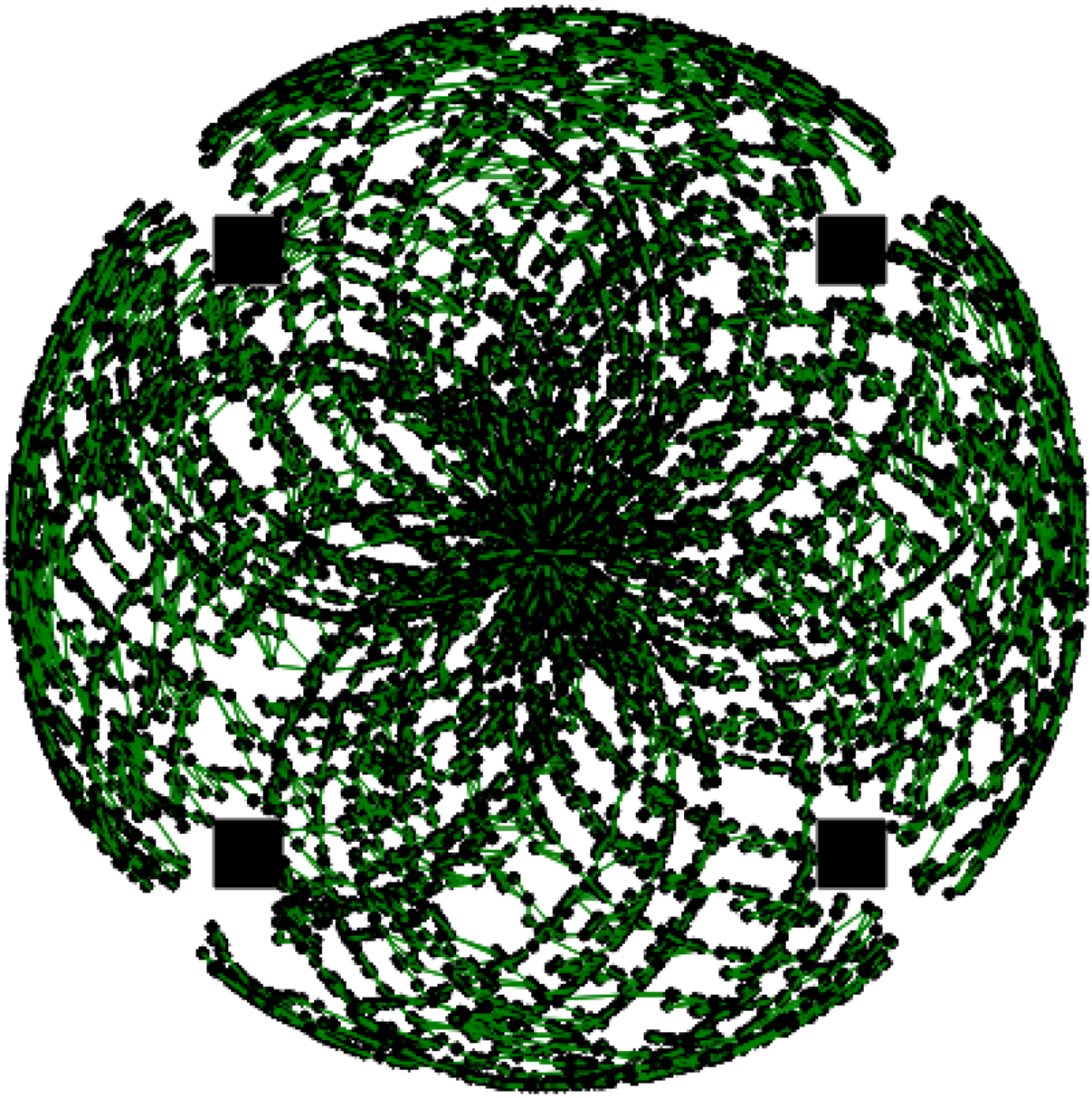
If a found point is within 2 units of the goal point, the RRT ends and the goal point is given a backpointer to this previous point. I decided to accept a radius from the goal point as it is nearly impossible for this search algorithm to find the goal point exactly.

display_planar.py

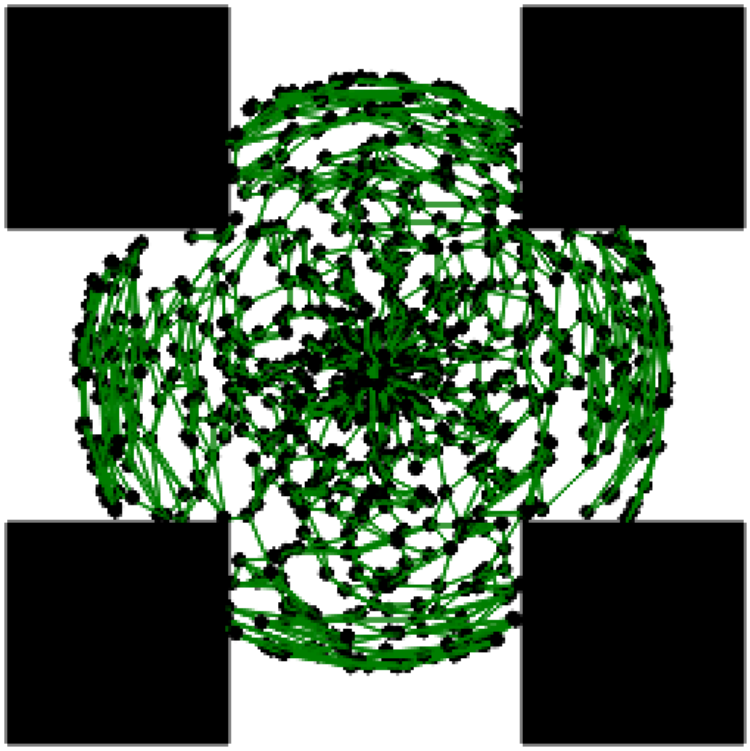
The two main parts to my draw function for MobileRobot are draw and step. `draw` first draws the RRT by iterating through the tree and drawing every edge. Second, it draws the obstacles in the robot's world. Thirdly, by using the step function, it is able to animate the path of the robot, following the path created in the `backchain()` method. Finally, `draw` draws the start and end points to make them obvious.

Testing

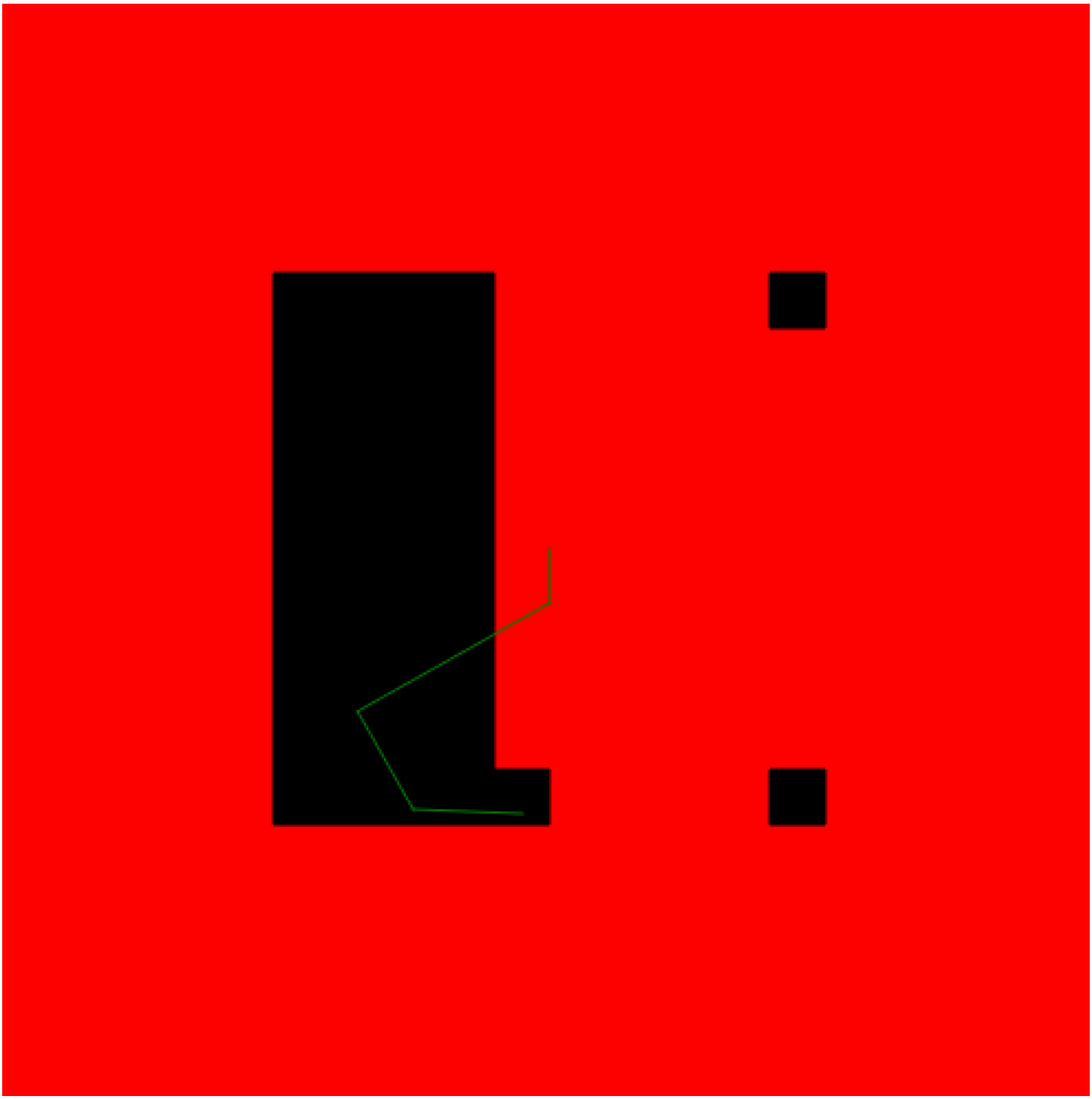
ArmRobot



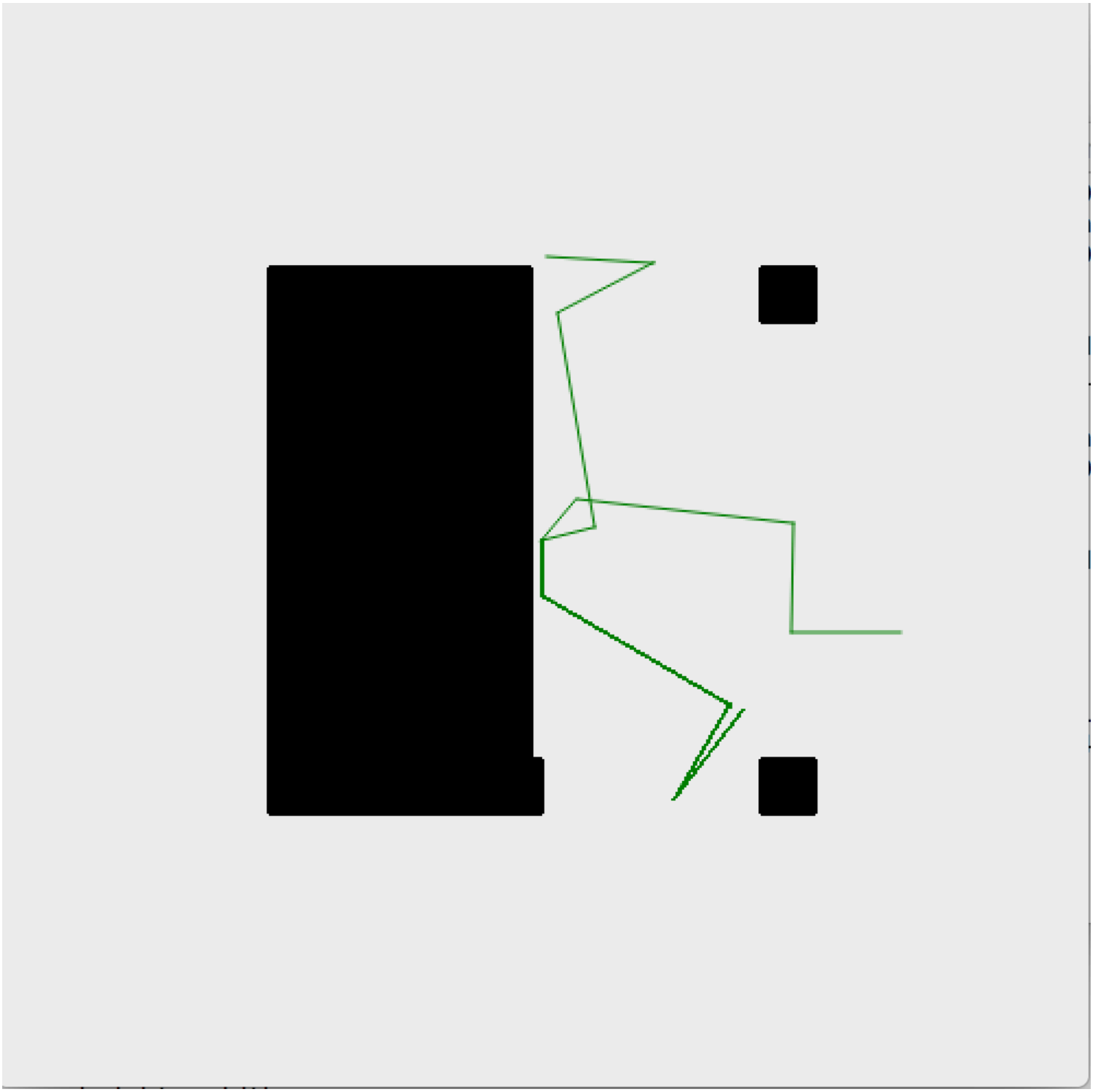
My first attempt at creating random configurations failed as it ended up creating a lot of unconnected subgraphs but was very pretty.



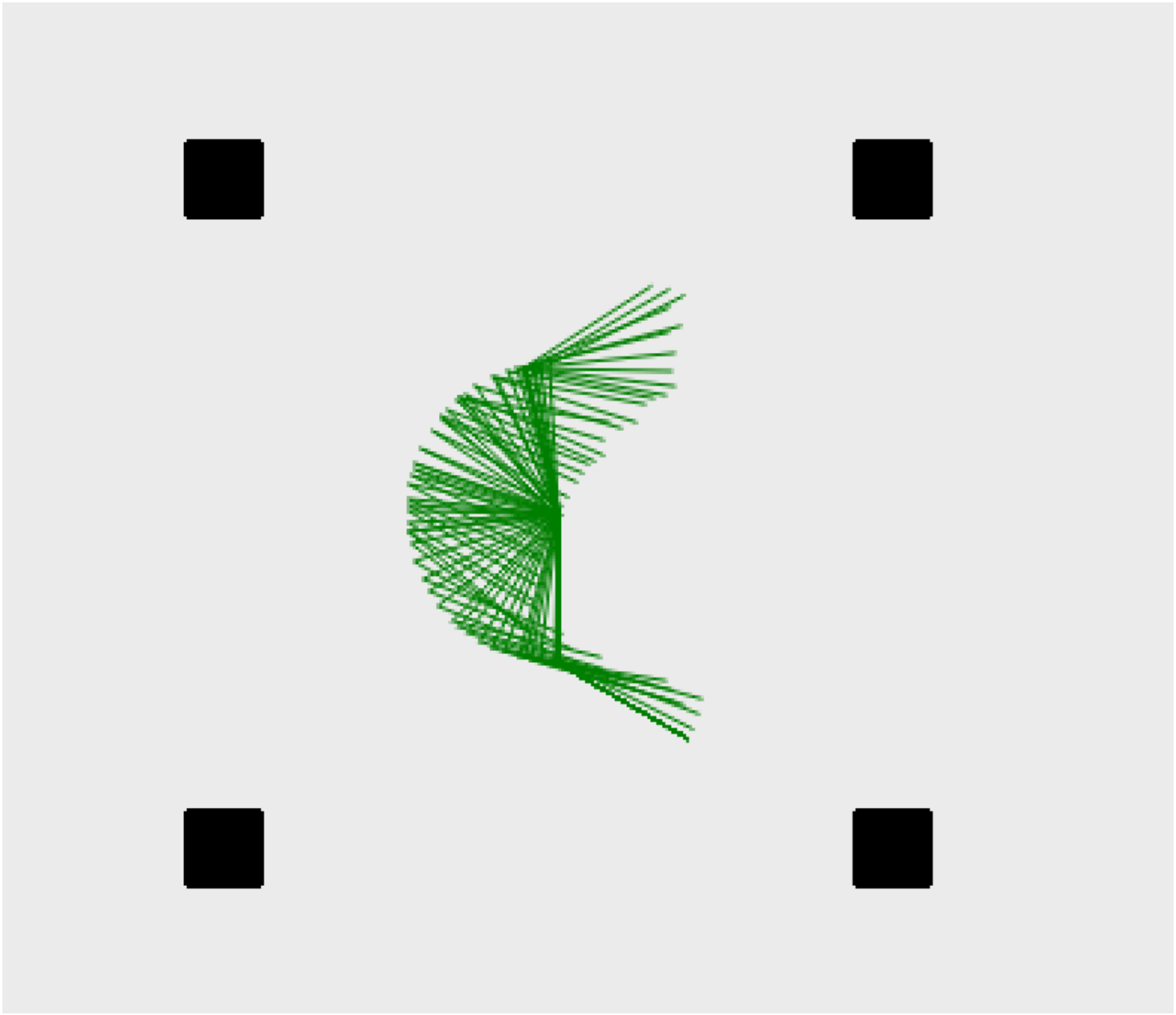
This is a successful example of my configuration space



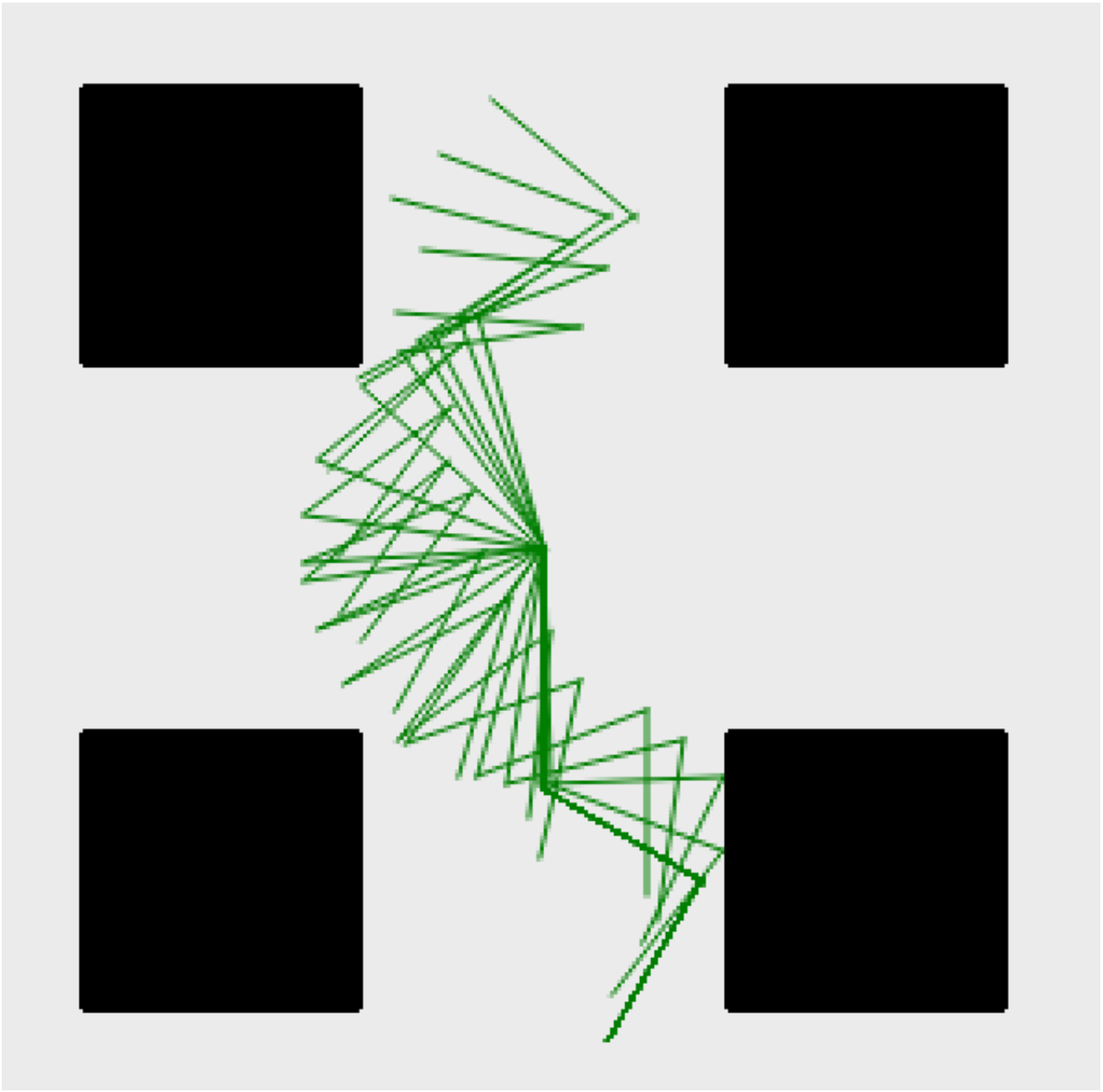
Above is an example of the screen turning red in the case of a collision, showing the effectiveness of my collision detection.



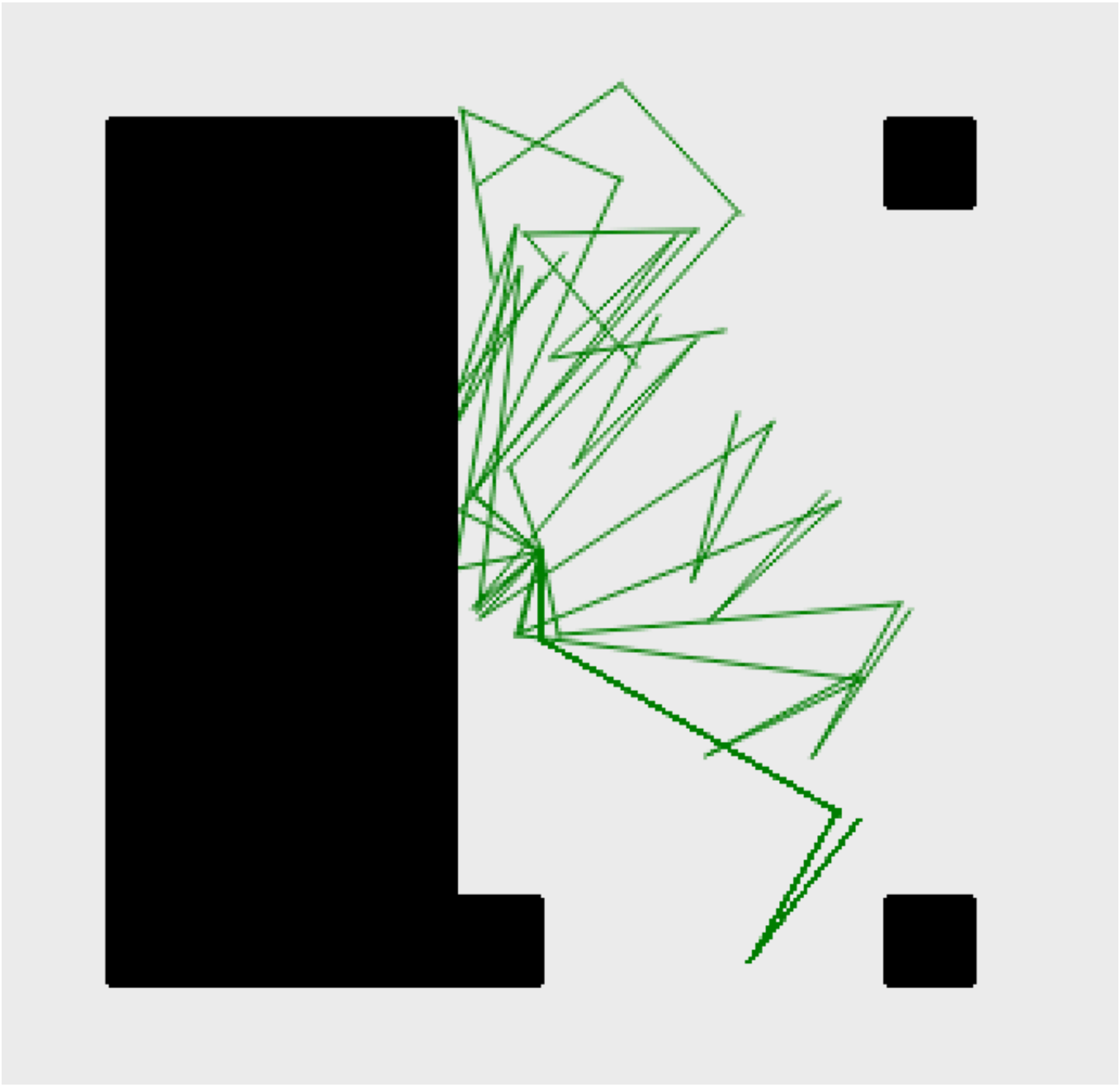
Above is an example of the result when there are not a lot of random configurations generated: the solution jumps around without much realism.



Above is a simple example of the path a 2R ArmRobot may take to reach its goal.

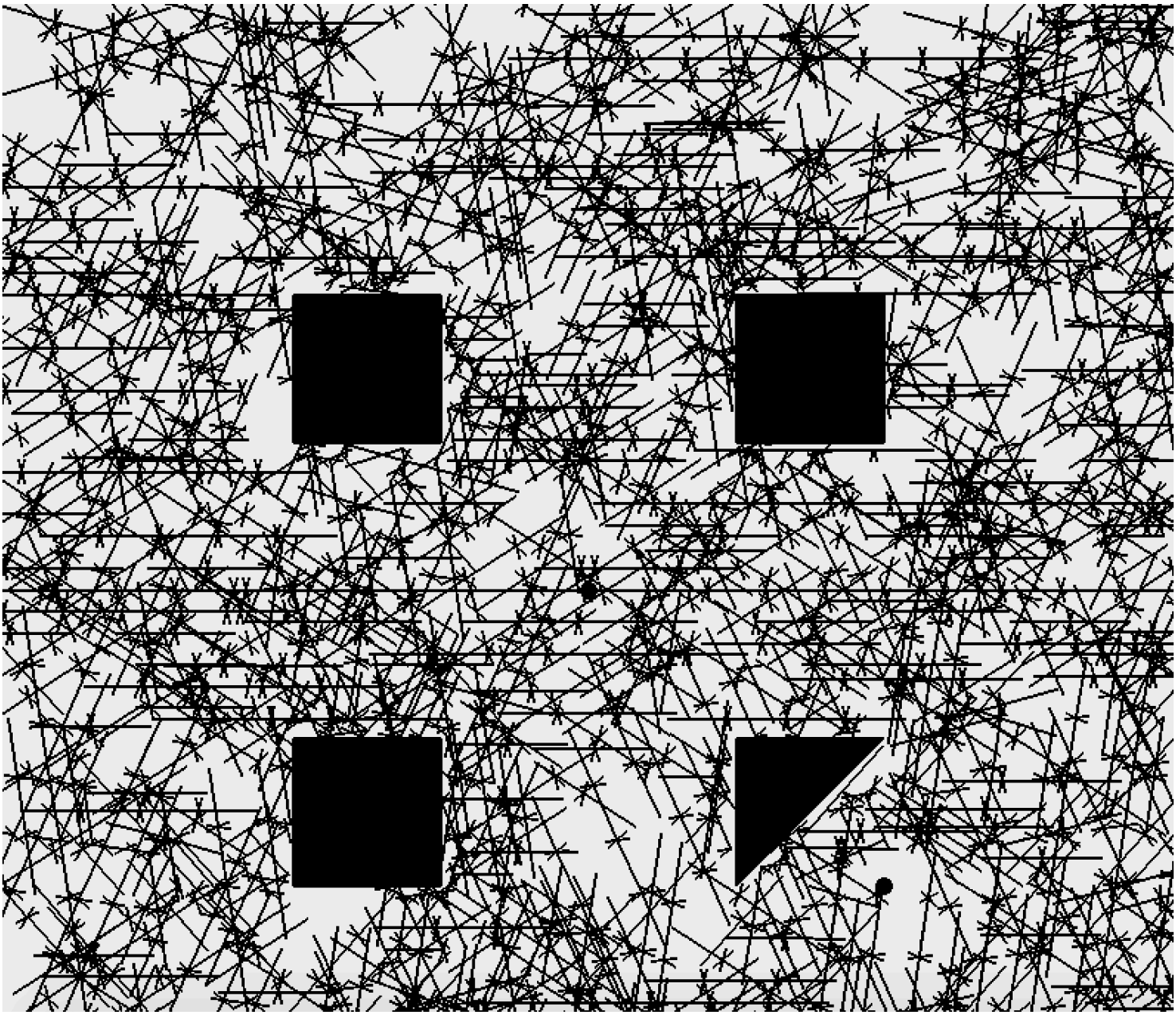


Above is an example of a 3R robot in a slightly harder maze.

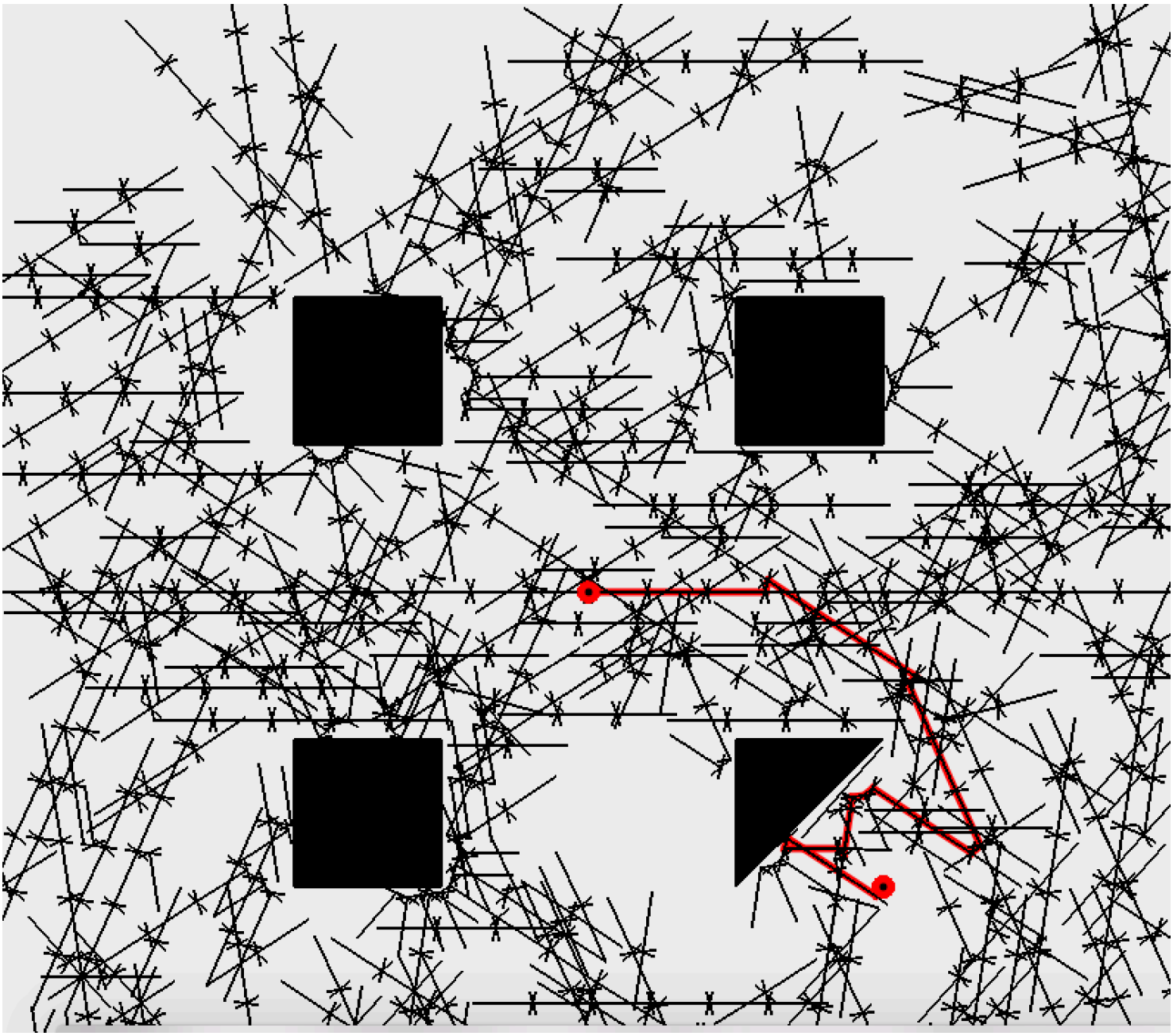


Above is an example of a 4R arm that is forced to travel clockwise in a difficult maze. This is important as the robot arm tends to want to travel counter-clockwise due to the way Annoy finds neighbors.

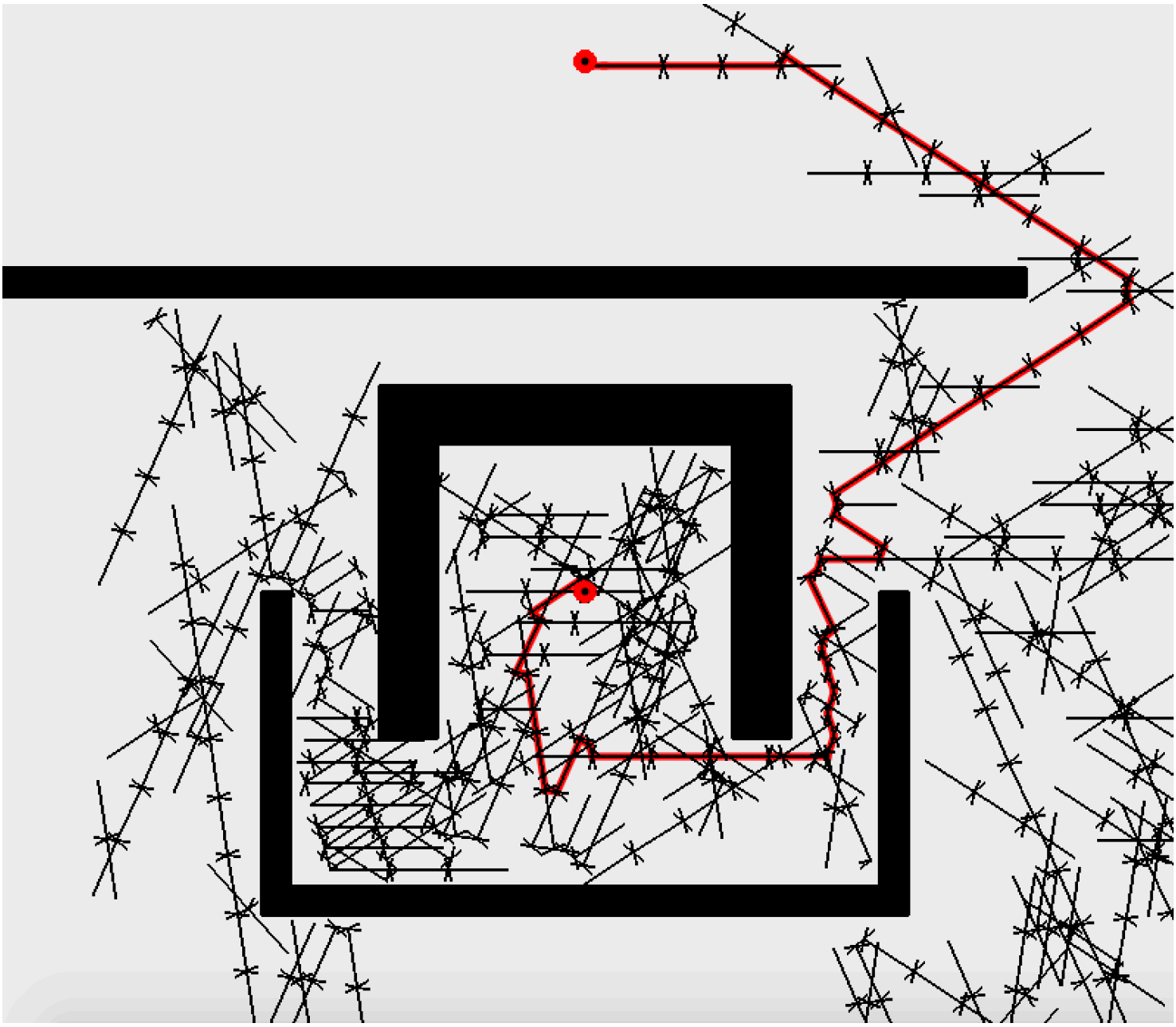
MobileRobot



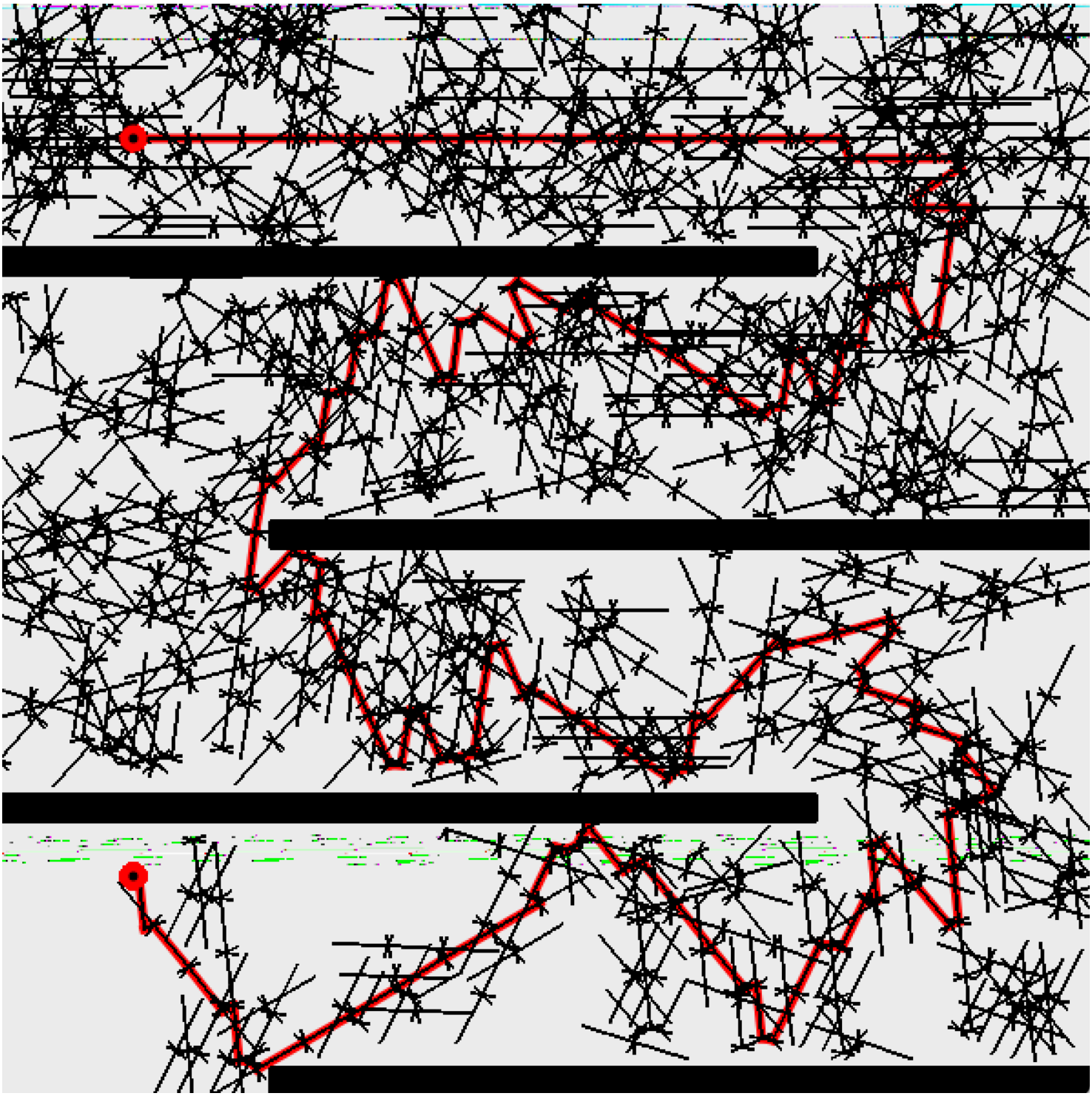
Above is an example of the RRT when no solution is found and it is allowed to expand to the full amount of random points.



Above is an example of an easy maze, which required less than 1000 random points to solve.



Above is an example of a medium maze in which the robot needs to navigate some turns to find its goal. This required less than 1500 random points.



Above is an example of a difficult maze that needed 2000 random points to solve. Here the robot must navigate multiple corners to find its goal.