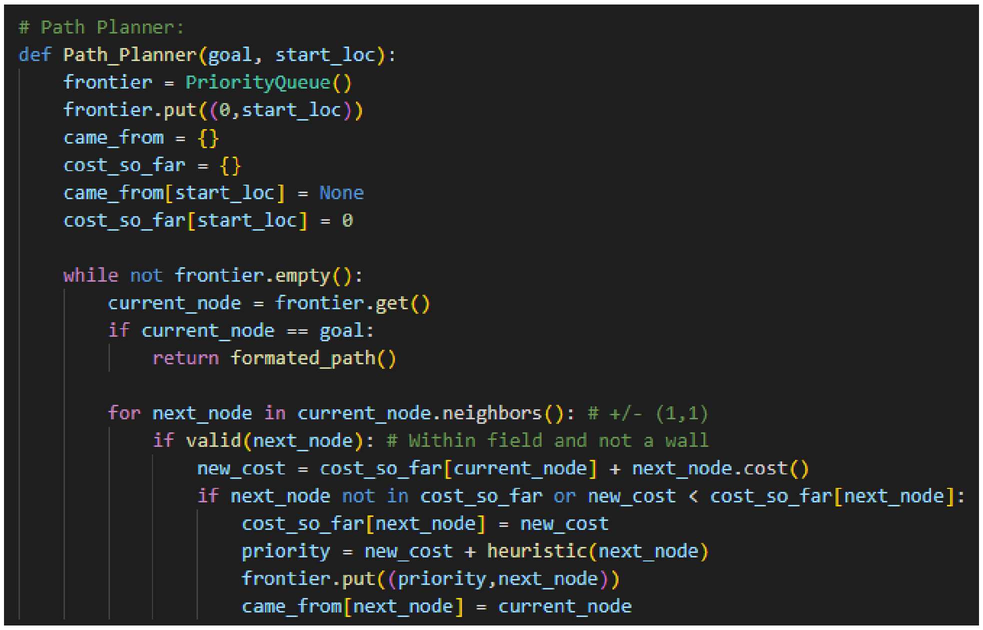
John Hall

RBE550 – HW4

Wildfire Report

Wumpus:

The Wumpus was the easiest component to implement in this assignment because it uses almost line for line the same path planner as the hero in the Flatland assignment. The only real difference was that the Wumpus had to plan to a node touching the goal node instead of the actual goal because that would be inside of an obstacle. I chose to have the Wumpus use the same grid discretization as the field, so in the animation you can see it move from grid location to another. This made the path planner super fast and able to plan paths across the map and around obstacles in milliseconds. Because there was no set speed for the Wumpus, I tuned its velocity to be 2.5 m/s. The Wumpus simply chooses the closest obstacle based on Euclidean distance as its target.



```
# Path Planner:
def Path_Planner(goal, start_loc):
    frontier = PriorityQueue()
    frontier.put((0,start_loc))
    came_from = {}
    cost_so_far = {}
    came_from[start_loc] = None
    cost_so_far[start_loc] = 0

    while not frontier.empty():
        current_node = frontier.get()
        if current_node == goal:
            return formated_path()

        for next_node in current_node.neighbors(): # +/- (1,1)
            if valid(next_node): # Within field and not a wall
                new_cost = cost_so_far[current_node] + next_node.cost()
                if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                    cost_so_far[next_node] = new_cost
                    priority = new_cost + heuristic(next_node)
                    frontier.put((priority,next_node))
                    came_from[next_node] = current_node
```

Figure 1. Pseudocode describing Wumpus A path planner*

Firetruck:

The firetruck, specifically the Probabilistic Roadmap it utilized for path planning, was more difficult to implement. To simulate the kinematics of the truck, I was able to use very similar code from the Valet assignment, the biggest difference is that I changed the collision checking from mask based to kd-tree based.

In the image below, the firetruck has a circle that contains itself. This radius is called the collision radius and at each sampled point or point along a path, the firetruck searches the obstacle kd-tree for any points withing the collision radius. These points are then checked if they lie inside the firetruck for collision. This method was far faster, and for the number of points sampled it was crucial.

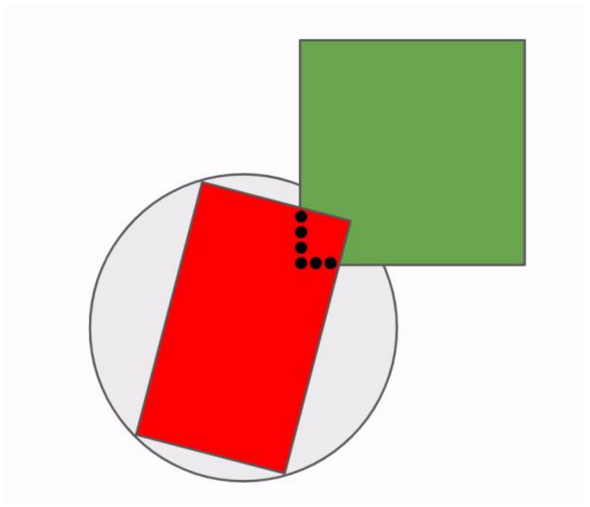
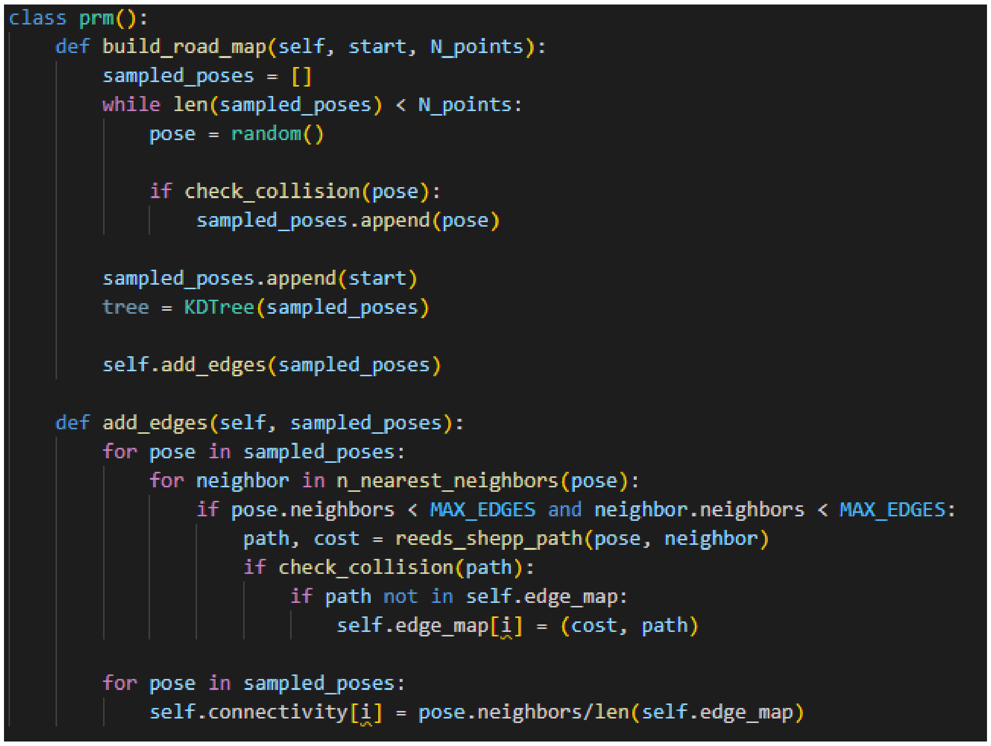


Figure 2. Collision checking overhaul example

At the beginning of the simulation, the firetruck creates a roadmap by randomly sampling the environment for a set of initial poses at different positions and orientations. These poses are then connected together with edges to form a map. The edges are generated by collision-free Reeds-Shepp paths for kinematically feasible connections. Each node in the roadmap has a maximum number of connections it can form and there is a maximum edge length.



```
class prm():
    def build_road_map(self, start, N_points):
        sampled_poses = []
        while len(sampled_poses) < N_points:
            pose = random()

            if check_collision(pose):
                sampled_poses.append(pose)

        sampled_poses.append(start)
        tree = KDTree(sampled_poses)

        self.add_edges(sampled_poses)

    def add_edges(self, sampled_poses):
        for pose in sampled_poses:
            for neighbor in n_nearest_neighbors(pose):
                if pose.neighbors < MAX_EDGES and neighbor.neighbors < MAX_EDGES:
                    path, cost = reeds_shepp_path(pose, neighbor)
                    if check_collision(path):
                        if path not in self.edge_map:
                            self.edge_map[i] = (cost, path)

        for pose in sampled_poses:
            self.connectivity[i] = pose.neighbors/len(self.edge_map)
```

Figure 3. Pseudocode describing roadmap and edge creation.

Once the edges have been added to the map, another crucial piece of information is generated: the connectivity heuristic. I run through and find the nodes that don't have the max number of neighbors and assign a probability based on the number of connections it has divided by the total number of edges. This heuristic is used when updating the roadmap to sample near points where there is less connectivity to hopefully allow for the local planner to reach its goal.

There are two ways the roadmap is updated with different implementations:

1) The goal obstacle does not have a sampled pose within extinguishing distance.

2) The local planner is unable to find a path to the desired node

For the first case, the road map will sample using a normal distribution around the desired obstacle until there is a pose generated within the radius of extinguishing. This is normally fairly quick, but occasionally the pose generated will be in a hard-to-reach location.

The second case of updating the roadmap uses the connectivity heuristic to choose a select number of desirable points to sample around. The roadmap picks a random direction and creates a new point a certain distance away and will sample using a normal distribution at this new point. This logic can be visualized in figure 4.

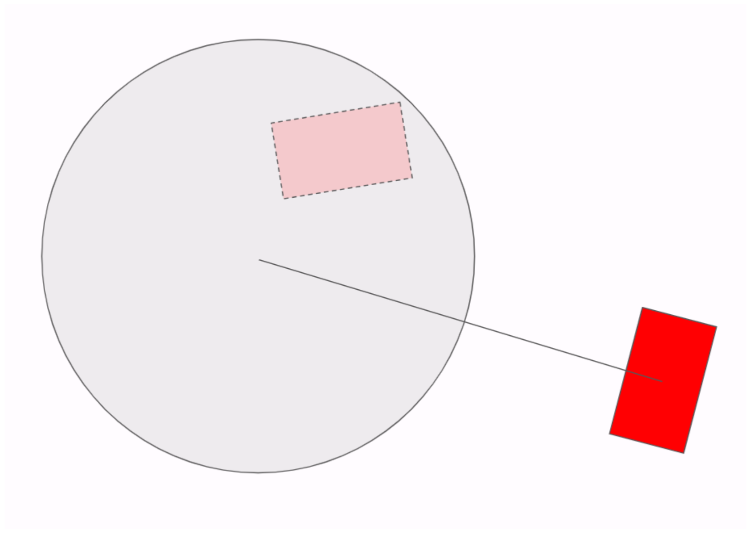


Figure 4. Updating roadmap to improve connectivity.

Each time the roadmap is updated, new edges are calculated, and the connectivity of the map is updated.

To actually plan a path using the roadmap, I chose Dijkstra's to be my local planner. I had attempted to use A*, but Euclidean distance was a pretty lousy heuristic when the paths all went in circles. The local planner was created as a class and would store the exploration of the roadmap from whatever location the firetruck was currently at. This meant that local planner could accept multiple goal locations and very quickly find the shortest path because of Dijkstra's node expansion.

In my opinion, the most difficult part, and the part that I wished I had more time to work on was how the firetruck prioritized planning to burning obstacles. In my final implementation, the firetruck will find the three closest burning obstacles based on Euclidean distance. These obstacles are then fed to the local planner where the shortest path is chosen. However, there were many instances I observed where the firetruck would choose a further obstacle because a closer obstacle would have required updated the roadmap.

Results:

Running the simulation was a little bit humiliating. The Wumpus dominated all runs without a doubt, both in terms of score and computation time.

	Firetruck		Wumpus	
Run	Score	Time	Score	Time
1	8	8.24	133	0.04
2	6	9.11	134	0.04
3	8	7.09	134	0.03
4	12	10.12	133	0.03
5	6	9.73	131	0.06

Table 1. Comparison of score and computation time for each player for all runs

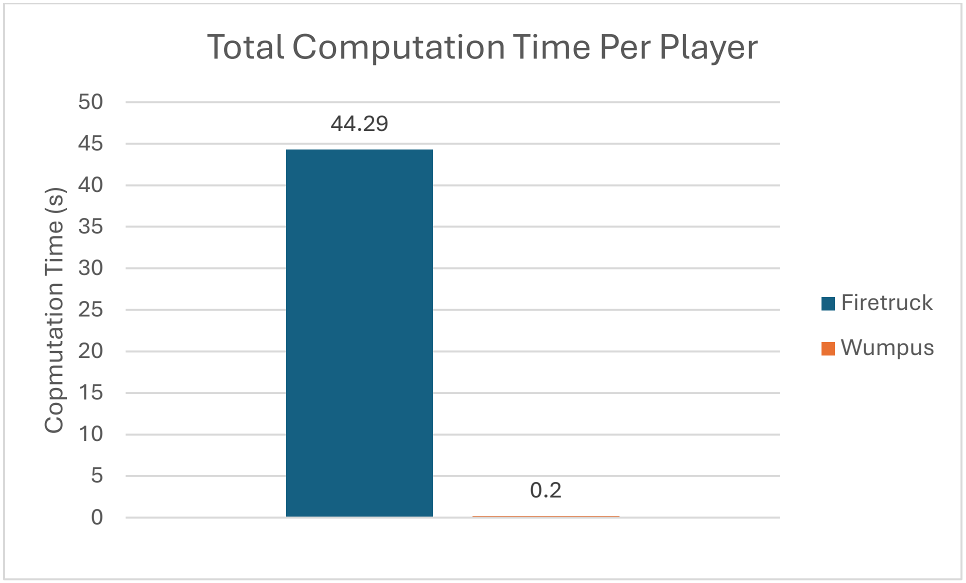


Figure 5. Graph comparing the total computation time between the firetruck and the Wumpus

The firetruck struggled to get the obstacles with enough time to extinguish them. There were plenty of times the firetruck would reach the obstacle successfully, but just not fast enough before it was fully burned. When I changed the time for an obstacle to burn from 10 to 15 seconds, it resulted in a high score of 32 for the firetruck.

Overall, I think the probabilistic road map was an interesting path planner, but I don't believe it is the most efficient. Especially for this problem, the firetruck would be doing 15-point turns because of the Reeds-Shepp paths and never make it to the obstacle in time. That being said, I think this method was far faster than hybrid-A* in terms of path generation. An interesting experiment would be to use hybrid-A* generated paths so nodes could be connected around obstacles.

Resources:
I took inspiration from PythonRobotics probabilistic roadmap implementation when designing my own.

I consulted ChatGPT for help with how to check if an obstacle point is withing the firetruck which pointed me to the OpenCV function I ended up using.

I used Scipy's kd-tree implementation to store my obstacle and sampled poses.

I once again used rsplan for Reeds-Shepp paths.