RBE 550: Wildfire Assignment

Azzam Shaikh
April 1, 2024
GitHub repo: github.com/azzamshaikh/sampling_vs_combinatorial_planners

I. Introduction

The purpose of this assignment is to compare a two classes of motion planning algorithms: combinatorial planners and sampling based planners. To achieve this, a maze-like field with trees and bushes will be developed. Within this environment, an arsonist - the Wumpus - will navigate the field, setting fires while a firetruck - a Mercedes Unimog - will fight the fires. Each player will utilize a different planning approach to achieve their task. Since the Wumpus is confined to a grid, it will use a combinatorial planner, such as A*. For the Unimog, a sampling based method will be implemented, specifically a probabilistic road map. Since the Unimog utilizes standard Ackermann steering, a local planner will be implemented to move along the trajectory found by the probabilistic road map. The implementation of the local planner ensures the vehicle kinematic constraints are followed. For both players, collision detection will need to be considered.

Both players will start on opposing sides of the field and the simulation will run for a set time. Simulation metrics, such as the ratio of burned and extinguished vegetation and CPU time required to compute various planning tasks, will be obtained at the end of a run. A total of five runs will be executed and the data will be averaged. The report will conclude with a discussion regarding the results of the simulation.

The implementation of the simulation environment was developed using Python and was built from scratch. The `pygame` library was utilized to create and visualize the simulation environment and the resulting visuals and animations.

II. Methods

The methods used to develop the environment, the Wumpus and its combinatorial planner, and the Unimog and its probabilistic road map and local planners will be discussed.

A. Environment

The environment requirements consisted of developing a flat square field, 250 meters on a side, filled with obstacles. Each obstacle is a piece of vegetation in the land that is shaped like a tetromino where a single obstacle is a square unit of 5 meters.

To visualize this field, the environment requirements were scaled to fit within a 1000 pixel by 1000 pixel window. This resulted in a 4x scaling from meters to pixels. Thus, for each square obstacle, its 5 meter dimension is equivalent to 20 pixels. This scaling will be utilized for any conversion of metric values to pixels.

For the obstacle generation, the code developed for the Flatlands assignment will be expanded for this application. The `Tetromino` class from the `tetromino.py` file generates the obstacle field for a given vegetation density, in this case 10%. Every time the simulation is ran, a new arrangement of obstacles are set.

These objects are generated as `pygame.Surface` objects with an associated Rect parameter and mask. Each obstacle has a singular state at a given time - either intact, burning, or extinguished. During the simulation, the states may change depending on if the Wumpus has set it fire or if the Unimog has extinguished it.

All the obstacles are stored in a `pygame.sprite.Group()` object and is passed to the Scheduler to allow simultaneous rendering and updating of the obstacles states. This object is also passed to the Wumpus and Unimog for collision checking as well as the Scheduler for scheduling goals.

Figure 1 visualizes an example simulation layout. The green objects are the vegetation and the tan background is the ground.

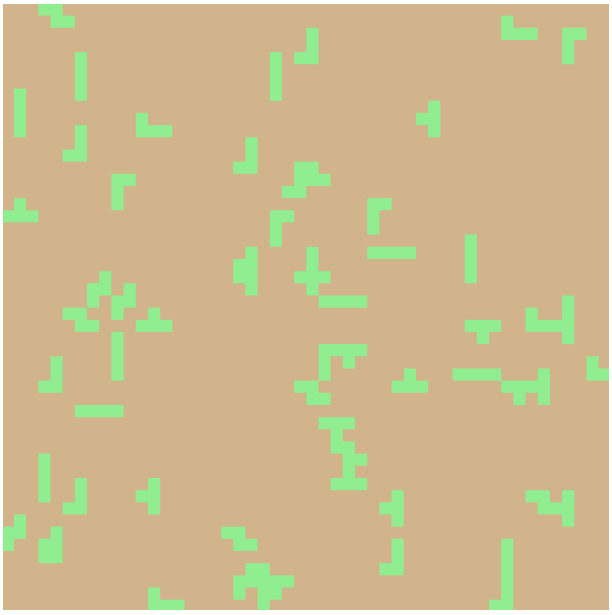

Fig. 1. Visual of the simulation environment.

Since each simulation will run for 3600 seconds, the simulation time needed to be defined. Since `pygame` is being used as the simulation renderer, the code for the program runs within the game loop. Thus, each loop of the game loop, or

each frame, is considered to be 1 second. The FPS of the simulation is set at 60; thus, for 3600 seconds, or frames, a single simulation will run for approximately 60 seconds.

B. Wumpus

The Wumpus is managed by the `Wumpus` class from the `wumpus.py` file.

For the Wumpus, no explicit dimensions were stated for its modeling. Thus, an aerial view of the Wumpus was as a 3 m by 3 m, lavender square.

Since the Wumpus is confined to a grid, it requires a combinatorial planner for movement. Thus, the Wumpus utilizes an A* planner for its motion. The Wumpus motion is set as a uniform square (8 motions) and it can move 2 meters per second, or 8 pixels per second. This value was selected since the Wumpus has legs and needs to run to the location. The length of his legs prevents him from moving any faster.

The visualization for the Wumpus is generated as a `pygame.Surface` object with an associated Rect parameter and mask. When searching for a path, the `pygame.sprite.spritecollideany()` function will be used to detect if a collision is occurring.

The Wumpus will receive its goal from the Scheduler. While the Wumpus knows that a firetruck is present, he is not afraid of getting caught by them. While they can try and apprehend him, the Wumpus knows the main priority for the firefighters is to put the fires out. In addition, he is aware that his fire will also spread, helping out his objective. Thus, the goal selection for the Wumpus is whatever he feels like doing at that time, which is completely at random. From the list of obstacles, the Wumpus will select any intact obstacle at random and go set it on fire.

When the Wumpus receives a goal, its will search the environment via an A* approach. The same search model used by the differential drive robot for the Valet assignment was applied here. When searching, each of the motions will be checked to see if it is a valid node (in bounds and not colliding with an obstacle) and then compute the heuristic cost. The heuristic cost is calculated by summing the step cost for the motion, the distance between the current node to the goal node, and cost of the previous node. Each node is then added to a PriorityQueue that is ordered via the heuristic cost. The node with the lowest heuristic cost is then selected for the next iteration of the search sequence.

Once the Wumpus reaches its goal, the Scheduler updates the state and color of the obstacle to show its on fire.

Figure 2 visualizes an example search for the Wumpus. The green obstacles represent intact bushes while the orange obstacles represent the bushes the Wumpus has set on fire. The purple dots represent the opened nodes and the final path is connected via lines.

A test simulation can be ran for the Wumpus via the `wumpus\_simulation.py` file.

C. Unimog

The Mercedes Unimog is managed by the `Unimog` class from the `unimog.py` file.

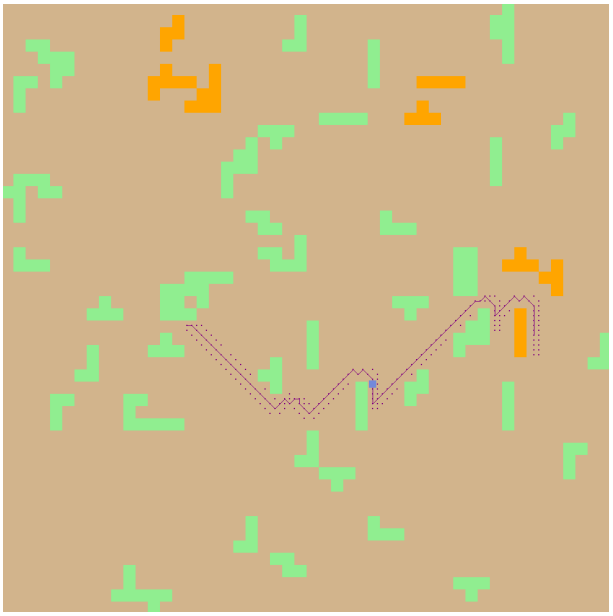

Fig. 2. Wumpus search path to a goal.

For the Unimog, since it is a car like robot with Ackermann steering, specific model parameters had to be defined. The width, length, and wheelbase were all scaled to their pixel dimension equivalents. Thus, an aerial view of the Unimog was a 4.9 m by 2.2 m, red rectangle.

The kinematic model of the Unimog can be seen in Figure 3.

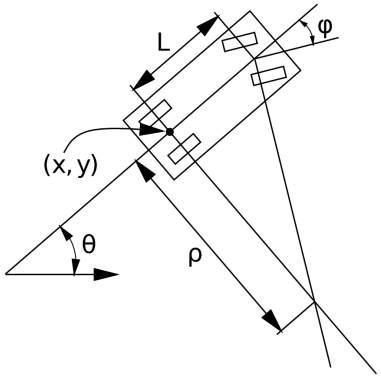

Fig. 3. Kinematic model of the Unimog [1].

In the global coordinate frame, the Unimog can be represented by $q = (x, y, \theta)$. However, because of the presence of the steering angle and velocity, the motion of the Unimog can be represented by the Equation 1. These equations as described by Equation 13.11 from the reference textbook, Planning Algorithms [2].

$$\begin{aligned} \dot{x} &= f_1(x, y, \theta, v, \phi) \\ \dot{y} &= f_2(x, y, \theta, v, \phi) \\ \dot{\theta} &= f_3(x, y, \theta, v, \phi) \end{aligned} \tag{1}$$

Since the control inputs to the Unimog is the steering angle ϕ and the velocity v, if $\dot{\theta}$ is set equal to the steering angle input, the motion constraint equations can be simplified as shown in Equation 2, where L represents the wheebase of the Unimog, which is 3.0 meters.

$$\begin{aligned} \dot{x} &= v \cdot \cos(\theta) \\ \dot{y} &= v \cdot sin(\theta) \\ \dot{\theta} &= \frac{v}{L} \tan(\phi) \end{aligned} \tag{2}$$

Based on this equation, the global position and heading values can be obtained via Equation 3.

$$\begin{aligned} x_{new} &= x_{previous} + \dot{x} \cdot dt \\ y_{new} &= y_{previous} + \dot{y} \cdot dt \\ \theta_{new} &= \theta_{previous} + \dot{\theta} \cdot dt \end{aligned} \tag{3}$$

For the path planning of the Unimog, a sampling based approach was required. Specifically, a probabilistic road map (PRM) planner had to be used. For the implementation of the PRM planner, since I didn't have a lot of experience implementing such a planner, a third party resource was utilized as a reference for the PRM implementation. An implementation developed by Atshushi Sakai for PythonRobotics was used as reference [3]. This implementation contains functions for creating obstacle KD trees, how to sample points, how to generate a road map, and using a Dijkstra planner to search the generated road map. The approach described was expanded for the this use case with `pygame` and is stored in the `PRM` class in the `prm.py` file.

The PRM implementation works as follows. In the simulation initialization, the obstacles sprite group object and start position of the Unimog get sent to the PRM object. The PRM object will create a KD tree storing the position of the objects. For a preset number of samples and a robot radius, the `sample_points()` function will generate a list of vertices in the free space. Each vertex is checked to ensure its distance from an object is greater than the robot radius. This will ensure the vertex will not cause a collision for the robot. After collecting the preset number of samples, the road map can be generated via the `generate_road_map()` function. Out of the samples generated, a KD tree of the sampled points will be made. From this KD tree, the function will go through all combinations of samples and attempt to create edges/connections between the vertex/samples. For each combination of vertex and edge, two things will be checked. First, if the distance between the vertex and edge is greater than the max edge length, the combination will be rejected. Second, the path generated will be checked for collisions. It will step through the path at a step length equal to the robot radius. For each step, it will check the obstacle KD tree for the closest obstacle. If the obstacle distance is smaller than the robot radius, the combination will be rejected. If there is no collision and the path length is shorter than the max length, the combination will get stored. After evaluating all combinations, the road map can be generated. A visual of a road map can be seen in Figure 4. The light blue lines show a visual of the connections between the vertices and edges.

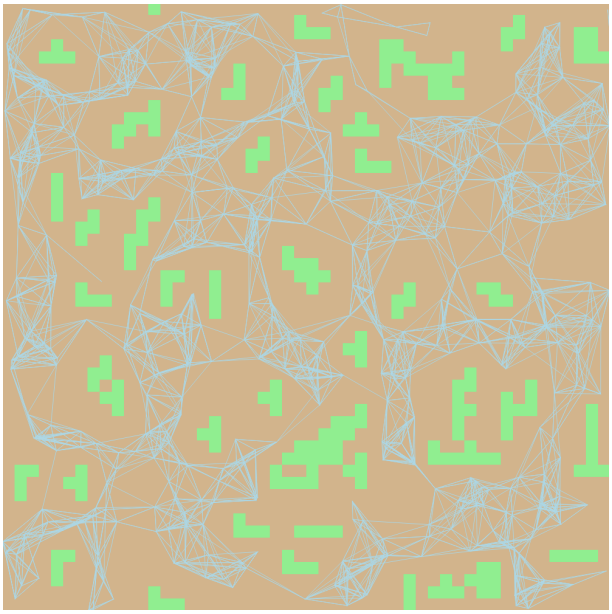

Fig. 4. Visual of the roadmap.

The density of the map can be modified by adjusting the number of samples and the number of connects for each vertex. For example, if the samples are set to 1000 and the maximum number of nodes is 50, the road map shown in Figure 5 gets generated. It can be seen that the connects are extremely dense as compared to Figure 4. There is an increased computation time to generate the higher density map however.

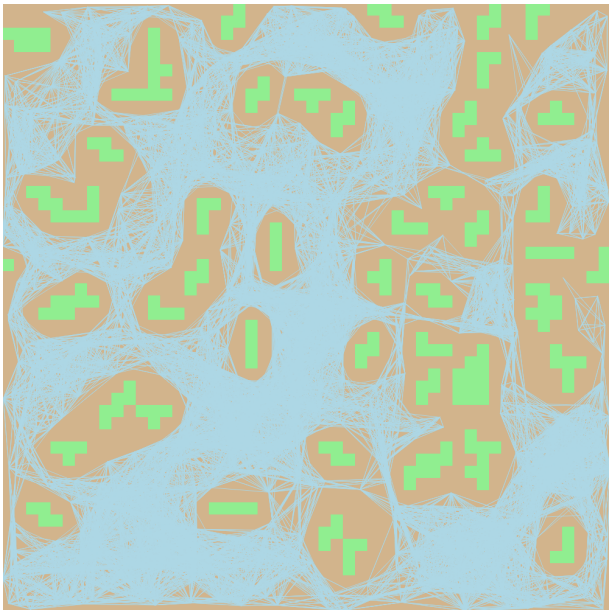

Fig. 5. Visual of a roadmap with higher samples and max edges.

Once the road map is complete, it can be stored in memory and queried as needed for various goals. In order to find a

goal, given a start position and goal position, a graph search algorithm can be used to find a path. The start position is selected to be the current position of the Unimog and the goal position is the location of the fire to extinguish. The goal selection criteria for the Unimog will be discussed in the next section.

For the start or goal position, the sampled points KD tree will be queried to find the nearest node/vertex to that position. That will then correspond to the start or goal position in the road map. In the code provided by Atshushi Sakai [3], a Dijkstra planner is utilized to traverse the path. This planner will expand by searching all nodes nearby until it reaches the final goal. Figure 6 visualizes the implementation of this planner. The blue nodes represent the open node during the search and the magneta lines shows the final path.

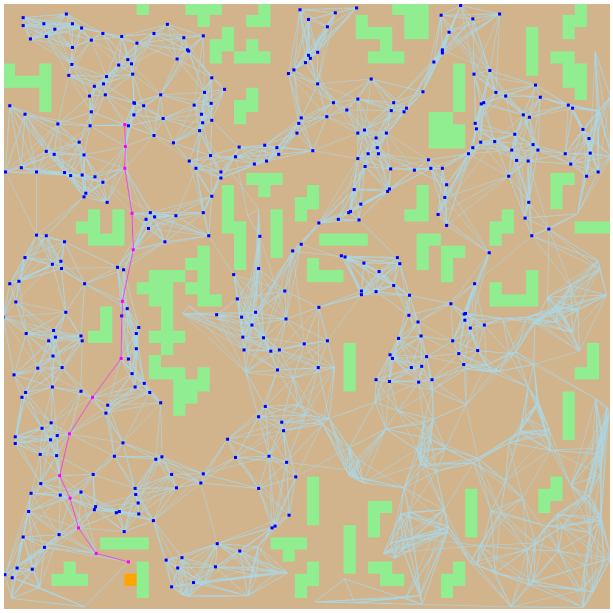

Fig. 6. Visual of the roadmap being traversed with a Dijkstra planner.

While this solution may work, for multiple search goals, this is not very efficient as it searches a large portion of the field before finding a solution. Since the Unimog has air support available, it knows the location of the goal. Thus, the search can be modified by traversing the nodes in the direction going to the goal, effectively creating an A* search for the PRM planner. Figure 7 visualizes the modified implementation of this planner. It can be seen that the shortest route gets found without exploring the entire field. This results in more efficient searches of the road map.

Once the PRM has found a path, the path is sent to the Unimog as a series of waypoints and the Unimog will then begin its own search for kinematically correct paths to the fire via its own planner. The same search model used by the car for the Valet assignment was applied here. Effectively, the search is a A* styled search where each possible new state is based on a motion primitive. Different combinations of velocity and steering angle will be simulated to predict the new position of

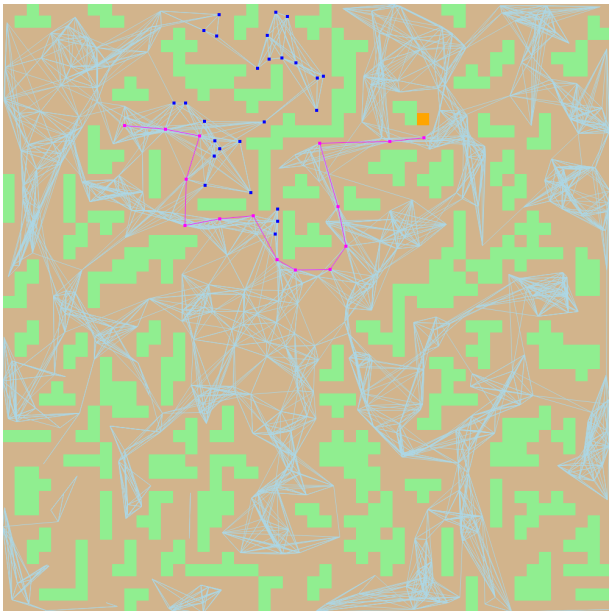

Fig. 7. Visual of the roadmap being traversed with a A* planner.

the car for some delta time. Each new state will be checked to see if it is a valid node (in bounds and not colliding with an obstacle) and then compute the heuristic cost. The heuristic cost is calculated by summing the step cost for the motion, the distance between the current node to the goal node, and cost of the previous node. Additionally, to expand the functionality from the Valet assignment, costs for turning and reversing were also added to have the planner prefer forwards motions. Each node is then added to a PriorityQueue that is ordered via the heuristic cost. The node with the lowest heuristic cost is then selected for the next iteration of the search sequence. Once the final goal is reached, the search will have generated a kinematically correct path for the Unimog to follow.

In order to define the motion primitives, the velocity and steering angle inputs and simulation time will need to be considered based on the Unimog's specific model parameters.

For the steering, a minimum turning radius of 13 meters was defined. Based on this turning radius, the max steering angle can be calculated. Using the following formula $\psi = atan(wheel\_base/turning\_radius)$ [4], the maximum steering input is 22 degrees.

For the model simulation time, since the `pygame` simulation time is per frame or per second, the desired simulated delta time was set equal to this value. Thus, the simulation will predict the position of the Unimog after 1 second.

For the speed of the Unimog, a max velocity of 10 meters per second was available. This results is a speed of 40 pixels per second. At this value, the model simulation creates large steps that were not suitable for reaching the waypoints effectively. After some tuning, a speed of 8 meters per second, or 32 pixels per second, was selected as this provided the optimal search step size for the environment.

Using these motion primitives for the planner, once the

Unimog reaches its goal, the Unimog will sit for 5 seconds, or frames, and the Scheduler will update the state and color of the obstacle to show its extinguished.

Figure 8 visualizes an example search for the Unimog. The grey obstacles represent the extinguished vegetation. The blue dots and lines represent the opened nodes and the final path, respectively, found by the PRM. The red dots and lines represent the nodes searched and final path, respectively, found by the local planner.

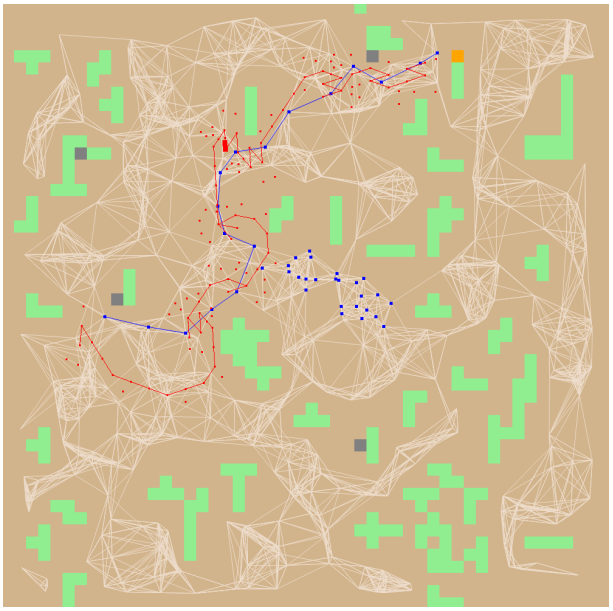

Fig. 8. Unimog search path to a goal.

A test simulation can be ran for the Unimog via the `unimog_simulation.py` file.

D. Scheduler

To tie the Wumpus simulation and Unimog simulation together, a Scheduler was developed. The Scheduler is managed by the `Scheduler` class from the `scheduler.py` file.

The purpose of the Scheduler is to (1) manage the states and rendering of the vegetation (as described in Section II-A), (2) provide goals for the Wumpus to search for and set fire to (as described in Section II-B), (3) expand any set fires, and (4) provide goals for the Unimog to search for and extinguish.

For any fires set by the Wumpus, if the obstacle has been burning for 10 seconds, or frames, the obstacle will set all obstacles within a 30 meter radius to fire as well. The Scheduler manages this aspect during rendering by checking obstacles on fire and finding the obstacles within 30 meters to also set on fire.

For providing the Unimog with goals, the Scheduler sends the location of an obstacle on fire that is closest to its location. This is accomplished by queuing a list of obstacles on fire, calculating their distance from the current position of the Unimog, and sending the closest one. This approach works effectively but generally depends on the quality of the road map. This will expanded on in Section IV.

With the implementation of the Scheduler, the Wumpus and Unimog can maneuver in the environment to accomplish their own objectives. The main simulation can be ran for both players via the `simulation.py` file.

III. Results

Once all supporting code was developed, the simulation can be ran via the `simulation.py` file. Figure 9, 10, and 11 shows sequences of a live simulation run. It can be seen that both players are working to accomplish their objectives.

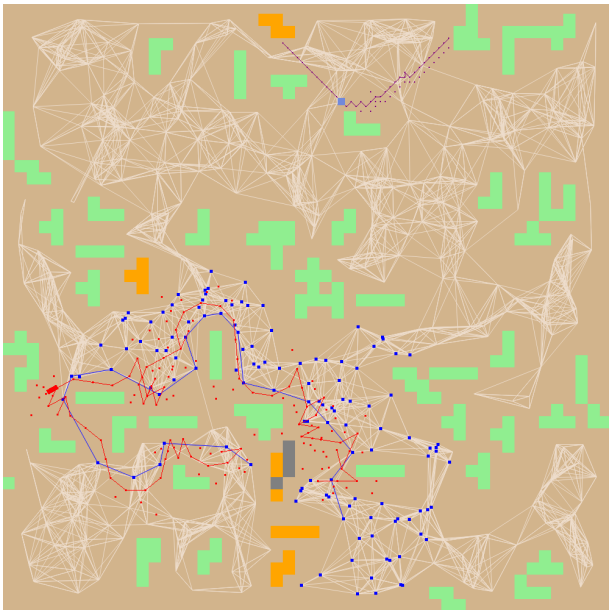

Fig. 9. Sequence 1 of a live simulation.

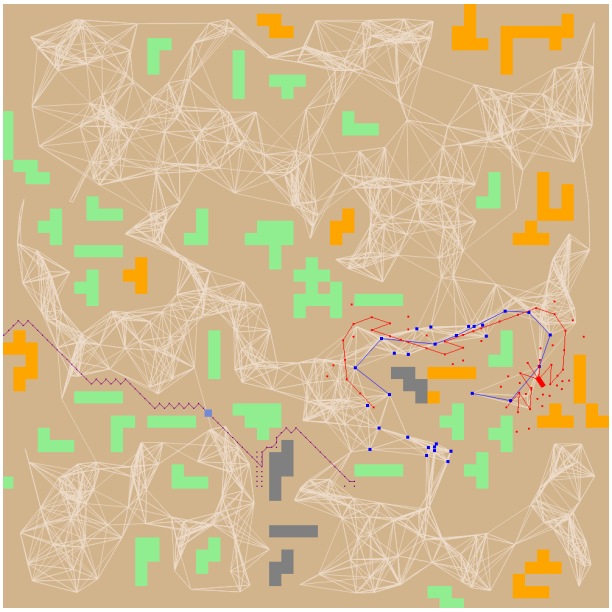

Fig. 10. Sequence 2 of a live simulation.

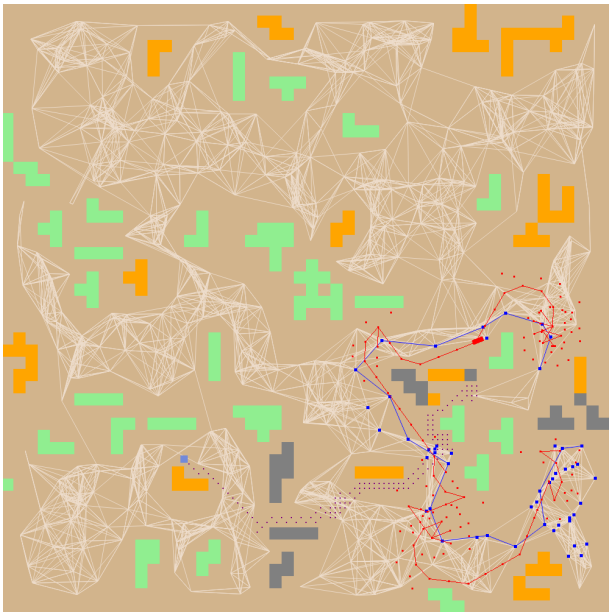

Fig. 11. Sequence 3 of a live simulation.

In order to evaluate the effectiveness of the different planners, 5 simulation runs were ran and various data was collected for each run. The data for each run can be seen in the `runs` folder. For each run, the following was collection:

- Total number of obstacles
- Total number of intact obstacles and ratio
- Total number of burned obstacles and ratio
- Total number of extinguished obstacles and ratio
- Total time Wumpus was searching
- Total time Unimog was searching
  - Time required for sampling
  - Time required for map generation
  - Time PRM was searching
  - Time Unimog local planner was searching

Table I summarizes the data on all obstacle states over the 5 runs along with the ratio averages and Figure 12 visualizes the averaged ratios in a bar chart. Based on the data, it can be seen that the overall ratios between the Unimog and the Wumpus are relatively similar. With an initial glance, this may suggest that the Wumpus and its combinatorial planner had found solutions at a similar pace as the Unimog and its sampling planner. However, the total burned obstacles includes obstacles that were burned as the fire spread while the total extinguished obstacles are all achieved per obstacle. Thus, this would indicate that the Unimog, per obstacle, extinguished more obstacles than the Wumpus set fire to. This highlights the capability and efficiency of the sampling based approach used by the Unimog.

Table II summarizes the data on the measured CPU execution time over the 5 runs along with the averages and Figure 13 visualizes the summed time over all 5 runs in a bar chart. The values collected describe the time required to complete that specific function. The sampling phase and map

TABLE I
OBSTACLE STATISTICS OVER ALL RUNS.

Obstacle State Ratios							
Run	Total Obstacles	Intact		Burned		Extinguished	
		Total	Ratio	Total	Ratio	Total	Ratio
1	237	67	0.28	84	0.35	86	0.36
2	235	44	0.19	113	0.48	78	0.33
3	238	79	0.33	70	0.29	89	0.37
4	234	85	0.36	54	0.23	95	0.41
5	228	79	0.35	78	0.34	71	0.31
Average Ratio		0.30		0.34		0.36	

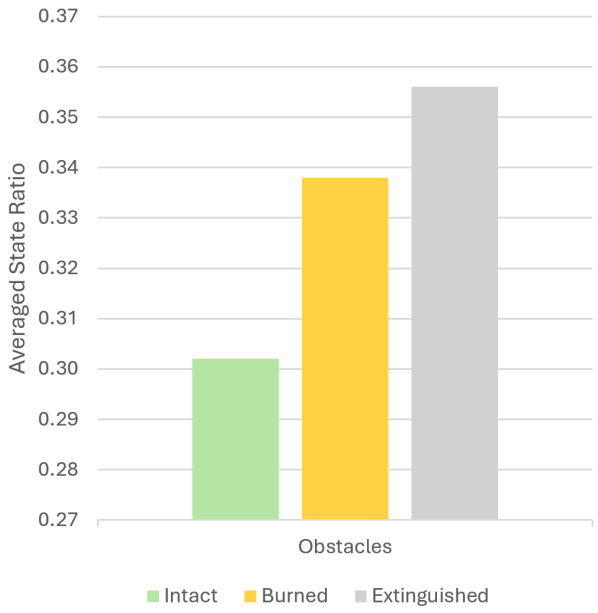

Fig. 12. Bar chart of the averaged state ratios for the three obstacle states.

generation phases were ran once during initialization and their values were collected before the simulation began. Since the Unimog's roadmap search and local search and the Wumpus's search were ran multiple times, their values represent the summation of time of each search call over the simulation duration. Time spent moving was not collected as the focus is to compare the effectiveness of the planners, not the speeds at which both players can move. All times were calculated using the `process_time()` function from the `time` module.

When comparing the total search times by vehicle, it can be seen that the Wumpus, with its combinatorial planner, always required more time for searching. This is expected since the grid based search that is utilized expands through the entire field at a fixed step size. For the Unimog, the combination of the map generation phase, the query phase, and the local search phase was always faster than the Wumpus. The sampling and map generation phase by itself accounted for roughly half the execution time. While the PRM may not generate a path for all possible states in the workspace, in most cases, there are enough points present to generate appropriate paths to a goal destination.

TABLE II
CPU EXECUTION TIME OVER ALL RUNS.

Run	CPU Execution Time [s]						
	Unimog Search Time						Wumpus Search Time
	Sampling Time	Map Generation	Roadmap Search	Local Search	Total Time	PRM Only Time	
1	0.031	0.594	0.000	0.906	1.531	0.625	1.828
2	0.031	0.578	0.094	1.078	1.781	0.703	1.813
3	0.016	0.594	0.031	0.828	1.469	0.641	2.906
4	0.016	0.609	0.016	0.672	1.313	0.641	2.125
5	0.016	0.563	0.047	1.000	1.625	0.625	1.688
Total Search Time [s]					7.719	3.234	10.359

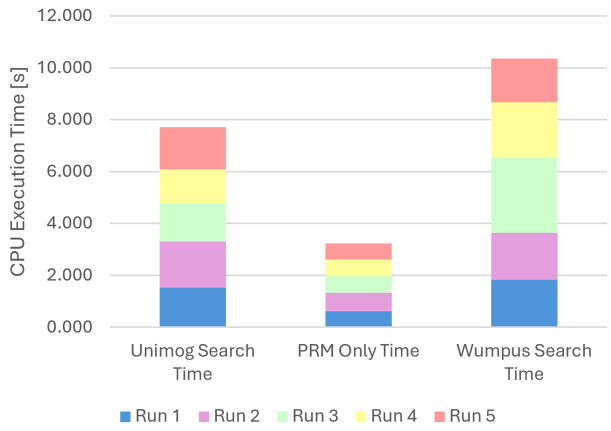

Fig. 13. Bar chart of the summed CPU execution time for the Unimog and the Wumpus.

IV. Discussion

Overall, based on the results acquired, the sampling based Probabilistic RoadMap used by the Unimog was better suited for finding quick and efficient paths to a goal destination in the environment. Compared to the combinatorial planner, the sampling based approach (PRM only) found paths in less than half the time. The generated path could then be used as waypoints for a local planner to follow to reach the goal efficiently, as opposed to having the local planner search the whole field. By generating a roadmap a priori, the Unimog could search the field for a suitable path without needing to traverse the entire field step by step, as required by the combinatorial planner. This reduced the computation during runtime by storing the map in memory and querying the map as required. The execution time required to query the roadmap itself was virtually negligible with all searches being less than a tenth of a second. Based on the data from Table II, the combinatorial planner required the most computational resources in order to reach its goal. This is expected as it is searching the entire field during runtime as opposed to looking up a path.

While the results describe the success of the sampling based approach, there were a few limitations noted when running various simulations. First, the success of the Unimog was entirely dependent on the quality of the roadmap generated. During the map generation process, when cycling through all possible combinations of vertices and edges, if the number of edges for a node is greater than a certain number, in this case 10 edges, the loop will cap the connections for that edge. Thus, there may still be other valid edges available for vertex. Additionally, when checking for collisions, if a collision is predicted, that edge would not be added to the vertex. Combining these two aspects during the generation process, in certain maps generated, there were instances of large spaces between obstacles for the Unimog to pass through, however, no node connections were made. Thus, this left a large gap in the map where the road map couldn't find a path through. This resulted in finding long paths around the obstacles to get to that specific destination, or even the creation of road sections that are not connected. Figure 14 visualizes this scenario. In the bottom right corner of the field, there exists a connection of roads that are not connected to the rest of the graph. In this case, based on the goal selection process for the Unimog, the nearest obstacle on fire and its associated nearest node on the roadmap lie on the disconnected series of roads. Since it can't be reached, the graph search will traverse through the whole roadmap in an attempt to reach that goal. Thus, resulting in the Unimog getting stuck at this point for the remainder of the simulation, unless a closer obstacle gets set on fire. To remedy this issue, the roadmap could be modified to have greater samples or greater number of edges per node. This would however increase computation time; thus, the tradeoff between computation resources and maximum field searching capabilities would need to be analyzed.

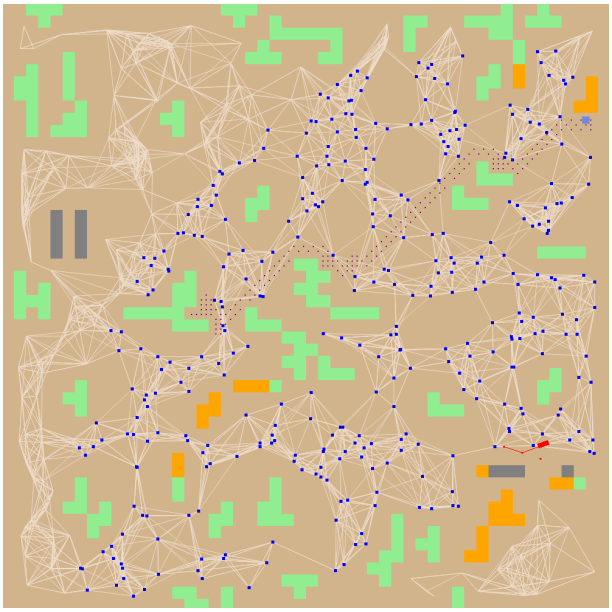

Fig. 14. Visual of the simulation environment.

An additional edge case exists where the Unimog obtains a new goal, and the PRM visual shows a valid connection is present, but the PRM fails to find a path. Figure 15 visualizes this issue. In the top left of the field, the closest roadmap node to the closest fire is supposedly on the piece of road that is extending to the left. However, the node fails to search that specific node when attempting to find a path. This issue has occurred several times where a path exists, at least visually, but

the planner decides to not move to that specific node. There are other cases where certain paths are found going one way and not the other. It is unclear what is causing this issue at this moment.

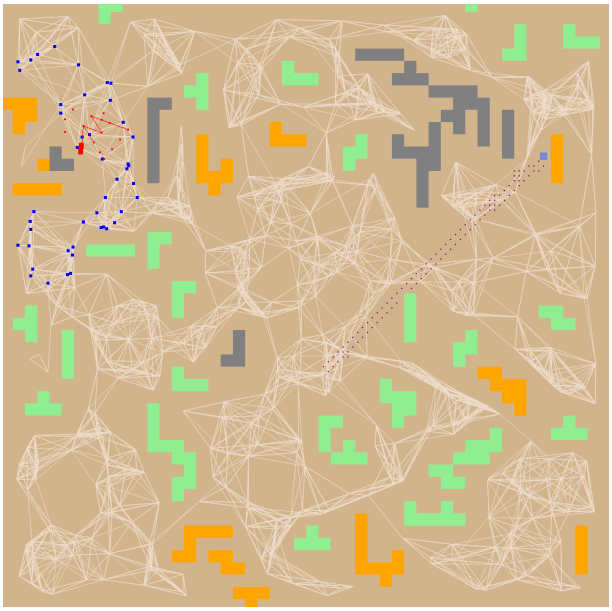

Fig. 15. Visual of the simulation environment.

Minus these specific edge cases, the simulation achieves the required objectives for this assignment. Future work would consider modifying the local planning of the Unimog to move in smoother segments, potentially via Reeds-Shepp curves.

REFERENCES

[1] D. M. Flickinger. Valet Assignment. (2024, Spring). RBE 550. Worcester, MA, USA: Worcester Polytechnic Institute

[2] S. M. LaValle, Planning Algorithms. Cambridge: Cambridge University Press, 2006.

[3] A. Sakai, "probabilistic_road_map.py," AtsushiSakai / PythonRobotics, https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/ProbabilisticRoadMap/probabilistic_road_map.py (accessed Mar. 18, 2024).

[4] Calculator Academy Team, "Turning radius calculator," Calculator Academy, https://calculator.academy/turning-radius-calculator/ (accessed Mar. 18, 2024).