RBE 550: Valet Assignment

Azzam Shaikh
March 18, 2024
GitHub repo: github.com/azzamshaikh/mobile_robot_autonomous_parking

I. Introduction

The purpose of this assignment is to implement path planning algorithms to park various vehicles in a compact space while taking vehicle kinematics and collisions into account. The objective is to gain experience with kinematic planning under nonholonomic constraints.

Three vehicles - a diwheeled robot, a standard car, and a truck with a trailer - will be simulated to parallel park in the simulation. The vehicles will begin in the Northwest corner of the 2D environment and park near the Southern border. The robot and car will have vehicles in front and behind the target parking spot while the truck will have only one vehicle in front of it. In the center of the environment, an object is present that the vehicles must avoid. For this simulation, skidding is not allowed for the vehicles.

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

II. Delivery Robot

For the delivery robot, the robot has diwheeled kinematics with skid steering. The source code for the delivery robot can be found in the `robot` folder in the `src` directory.

A. Methods

The methods for implementing the robots kinematics, collision detection, and the planning algorithm will be described.

1) Kinematics: The kinematic model of the robot can be seen in Figure 1.

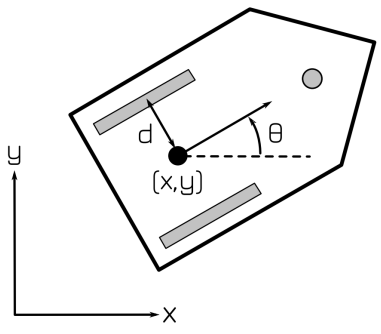

Fig. 1. Kinematic model of differential drive robot [1].

Based on this model, the kinematic equations can be described by Equation 1. These equations are as described by Equation 13.16 from the reference textbook, Planning Algorithms [2].

$$\begin{aligned}\dot{x} &= \frac{r}{2}(u_L + u_R)\cos(\theta) \\ \dot{y} &= \frac{r}{2}(u_L + u_R)\sin(\theta) \\ \dot{\theta} &= \frac{r}{L}(u_R - u_L)\end{aligned} \tag{1}$$

The u_L and u_R variables refer to the angular velocity of the left and right wheel, respectively. These variables can be modified to be in the reference frame of robots overall velocity as opposed to angular velocity of the wheels. In this case, $v = \omega \cdot r = \frac{v_L + v_R}{2}$ where $v_L = r \cdot u_L$ and $v_R = r \cdot u_R$. With these equations, the robot kinematics can be defined in the global coordinate frame with respect to the action variables of velocity, v, and steering angle, ω. The final representation of these equations can be seen in Equation 2.

$$\begin{bmatrix}\dot{x} \\ \dot{y} \\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1\end{bmatrix} \begin{bmatrix}v \\ \omega\end{bmatrix} \tag{2}$$

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

$$\begin{aligned}x_{new} &= x_{previous} + \dot{x} \cdot dt \\ y_{new} &= y_{previous} + \dot{y} \cdot dt \\ \theta_{new} &= \theta_{previous} + \tan\frac{\sin(\dot{\theta} \cdot dt)}{\cos(\dot{\theta} \cdot dt)}\end{aligned} \tag{3}$$

2) Collision Detection: After defining the robot kinematics, a collision detection system could be implemented. Since the `pygame` library was used for implementing the simulation, all objects placed in the display window are defined using Rect objects. These Rect objects define the boundary of a surface or image that is loaded onto the screen. With all objects being defined as Rect objects, one of the many Rect collide methods can be used to detect if there is any collision between two objects. Thus, when the planner is searching for a path, a Rect object is placed at each point in the search path and tested for any collision. This detection is achieved using the `collidelistall()` function.

3) Planning Algorithm: For the planning algorithm, an A* style search algorithm was implemented for finding the path from the start to the goal position. This planner searches the environment on a pixel to pixel basis. The pseudocode for the algorithm can be seen in Algorithm 1. Since `pygame` is being used, the traditional algorithm implementation had to be modified to run within the game loop of the program. Effectively, the while loop is ran by the game loop itself. Thus,

within the game loop, there is an if clause to determine if the goal has been found or not. If the goal has not been found, the planner object gets called.

Algorithm 1 Planning algorithm pseudocode for the differential drive robot

```
while goal not found do
    get node from frontier
    if node is goal then
        goal found is True return node
    end if
    add node to explored
    for child in expand do
        if child not in explored and not in frontier then
            add child to frontier
        else if child in frontier then
            if child cost less than child cost in frontier then
                delete child from frontier
                add child to frontier
            end if
        end if
    end for
end while
```

4) Controller: Once the path is found, the path is downsampled to create waypoints and are then passed to the Controller to move the robot to each waypoint. Due to the skid steering capability of the robot, two controllers were created. One controller, the position controller, controls the overall movement of the robot to the different waypoints. Once all waypoints have been reached, an orientation controller gets called to correct the robot to the appropriate heading. Both controllers use a simple PID control to reduce the error between the robots current pose and the desired pose.

B. Results

Once all aspects of the simulation were implemented, the simulation could be ran. The overall simulation environment contains two parked cars and two potholes for the robot to avoid.

Figure 2 visualizes the simulation environment and the path generated by the planner in white.

Figure 3 visualizes the simulation environment and the path taken by the robot in blue.

C. Discussion

Overall, the differential drive robot was able to successfully reach the goal pose without collisions and following its kinematic constraints. A video of the motion is available for viewing.

III. Car

For the car, standard Ackermann steering will be implemented with a wheelbase of 2.8 m. The source code for the car can be found in the `car` folder in the `src` directory.

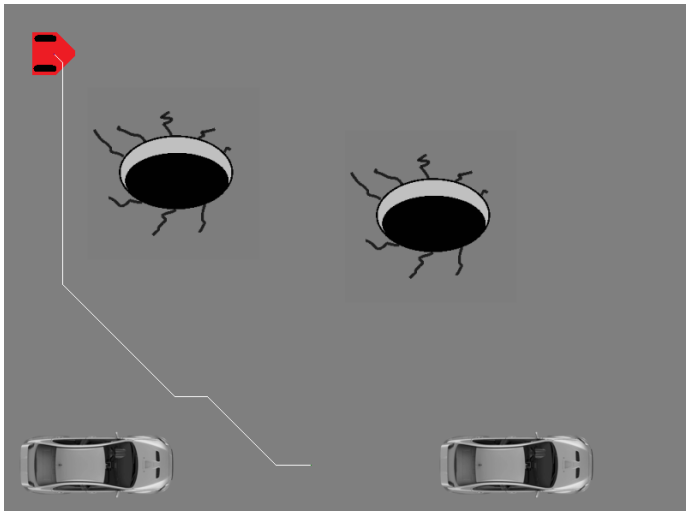

Fig. 2. Path generated by search algorithm.

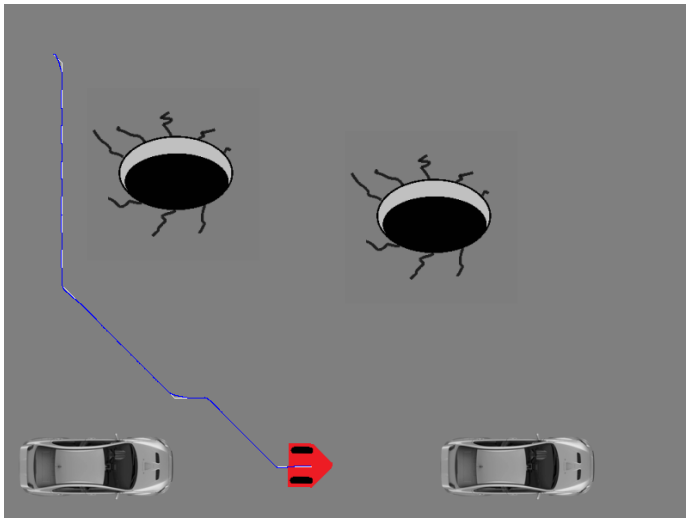

Fig. 3. Path followed by the robot.

A. Methods

The methods for implementing the car kinematics, collision detection, and the planning algorithm will be described.

1) Simulation Scaling: For this vehicle, since a length was specified for the wheelbase, the simulation environment had to be scaled. For the differential drive robot simulation, a standard conversion from meter to pixel of 3779 was used. This resulted in using small values to reflect speed and position. For the car, it was elected to set 2.8 meters to 100 pixels. This resulted in a scaling of 35 pixels per meter ($100pixels/2.8meters = 35\frac{pixels}{meter}$).

For the visual of the car, the axles are located directly at the front and end of the rectangle.

2) Kinematics: The kinematic model of the car can be seen in Figure 4.

In the global coordinate frame, the car can be represented by $q = (x, y, \theta)$. However, because of the presence of the steering

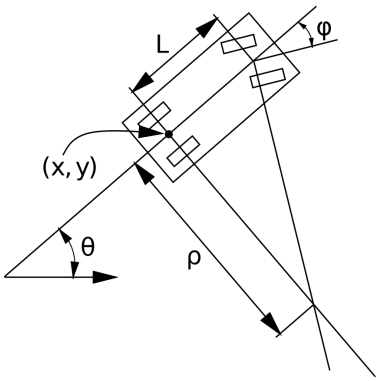

Fig. 4. Kinematic model of a car [1].

angle and velocity, the motion of the car can be represented by the Equation 4. These equations as described by Equation 13.11 from the reference textbook, Planning Algorithms [2].

$$\begin{aligned} \dot{x} &= f_1(x, y, \theta, v, \phi) \\ \dot{y} &= f_2(x, y, \theta, v, \phi) \\ \dot{\theta} &= f_3(x, y, \theta, v, \phi) \end{aligned} \tag{4}$$

Since the control inputs to a car 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 5.

$$\begin{aligned} \dot{x} &= v \cdot \cos(\theta) \\ \dot{y} &= v \cdot sin(\theta) \\ \dot{\theta} &= \frac{v}{L} \tan(\phi) \end{aligned} \tag{5}$$

Based on this equation, the global position and heading values can be obtained via Equation 6.

$$\begin{aligned} x_{new} &= x_{previous} + \dot{x} \cdot dt \\ y_{new} &= y_{previous} + \dot{y} \cdot dt \\ \theta_{new} &= \theta_{previous} + \dot{\theta} \cdot dt \end{aligned} \tag{6}$$

3) Collision Detection: Due to the nonholonomic constraints presented by the car and the rectangular shape of the vehicle, the collision detection used for the differential drive robot couldn't be used for this simulation. When rotating the Rect object for the car, a new Rect object is created that is large enough to cover all edges of the rotated image. Thus, this Rect object ends up being larger than the car model itself and any collision checking would find phantom collisions between the obstacles and the larger Rect object.

Therefore, a new collision detection approach had to be implemented for more accurate collisions. This led to creating an entirely new simulation environment. To accomplish accurate collision detection, `pygame`'s mask functionality was used. A mask object stores a 2D bitmask where each bit in the mask represents a pixel [3]. This allows accurate collision detection between objects. Thus, for each obstacle and for the car itself, a Surface gets created and a mask is created out of that surface. To detect collision, the `collide_mask(mask1,mask2)` functionality is used to detect whether any objects collide. During the search process, the algorithm simulates the position and orientation of the vehicle and updates the position of the mask in the screen. At each iteration, a collision check will occur to see if the node will cause any collision between the vehicle and any obstacles.

4) Planning Algorithm: Initially, the planner used for the robot was attempted to be used here. While a path was found, the controller failed to properly reach the various waypoints along the path. This led to significant troubleshooting with the controller but no viable solution was found. This led to the creation of an entirely new planner. The new approach utilized focused on using the car kinematics to find the best path. During each search expansion, select motion combinations of velocity and steering angle were simulated to predict the position of the car at some time in the future. These motion primitives would then be able to generate a path that followed the kinematic constraints of the vehicle.

The implemented planner algorithm pseudocode can be seen in Algorithm 2. The algorithm is an A* style search algorithm, similar to the robot. As mentioned previously, the while loop is ran by the game loop itself. Thus, within the game loop, there is an if clause to determine if the goal has been found or not. If the goal has not been found, the planner object gets called.

For this planner, the motion primitives used are as follows: moving forward at 1 m/s (times the meters2pixel scaling) at -35 degrees, 0 degrees, and 35 degrees and moving backward at 1 m/s (times the meters2pixel scaling) at -35 degrees, 0 degrees, and 35 degrees, each at some specified delta time. This results in a six possible nodes in each search step. Figure 5 shows a visual of these motion primitives from some given start position with a dt of 1 second.

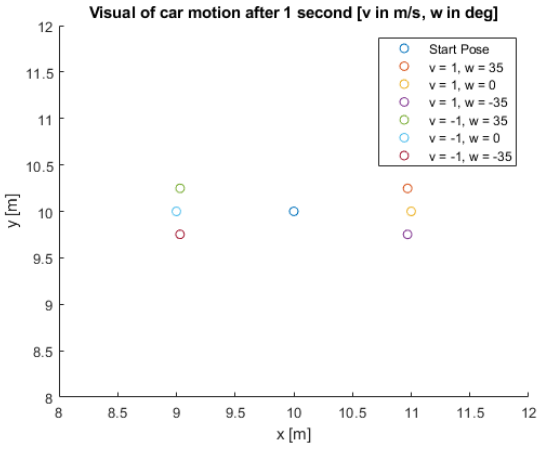

Fig. 5. Visual of motion primitives.

These path sequence found via the expansion of motion primitives from the start to the goal node will be one that follows vehicles kinematic constraints.

Algorithm 2 Planning algorithm pseudocode for the car

```
while goal not found do
    get node from frontier
    if angular diff and distance to goal < x,y then
        goal found is True return sequence
    end if
    add node to explored
    for motions in motion primitives do
        calculate new state based on motions and dt
        if new state is not valid then
            continue
        else
            create a node with the new state
            compute h value of the node as the distance from
                the node to the goal
            compute g value of the node as the distance
                from the node to the previous node and the
                angular difference between the node and the
                goal pose
            compute f value of the node as the sum of h and
                g
            add node to children
        end if
    end for
    for child in children do
        if child not in explored and not in frontier then
            add child to frontier
        end if
    end for
end while
```

B. Results

Since the planner was simulating the position of the car over time, the path from the start to the goal was captured and the resulting simulation shows a periodic snapshot of the cars motion over time.

Figure 6 visualizes the simulation environment and the various nodes searched by the planner in yellow.

Figure 7 visualizes the resulting path found by the planner in magenta and the position of the car at the end goal.

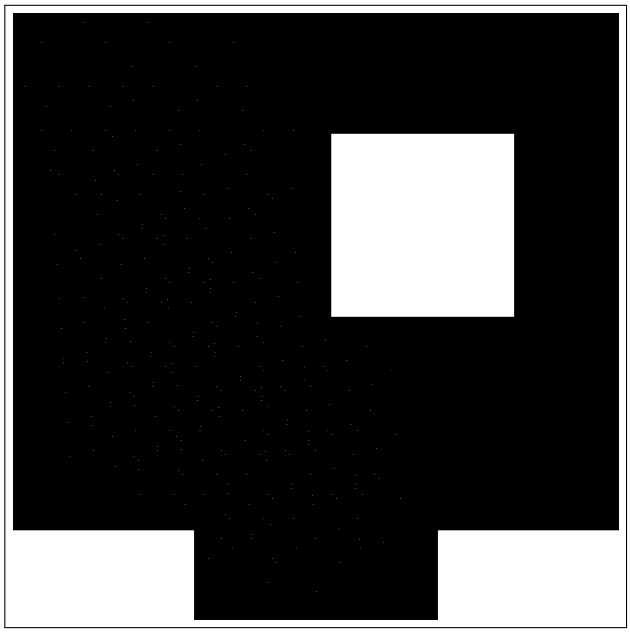

Fig. 6. Path generated by search algorithm.

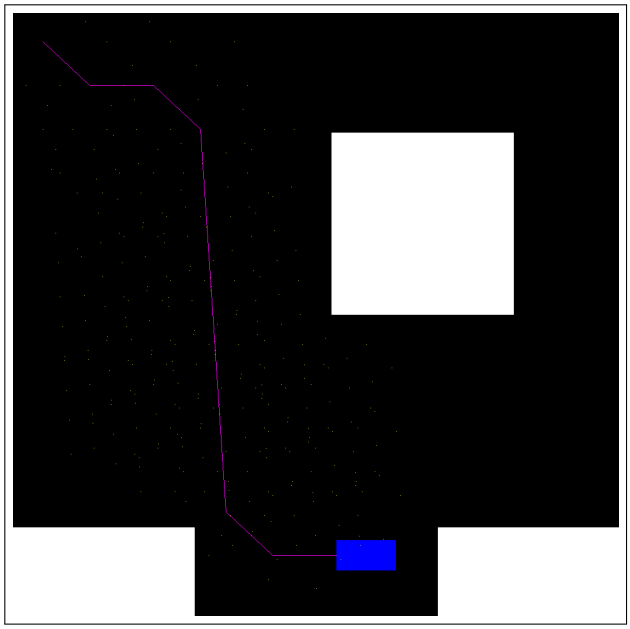

Fig. 7. Path followed by the car.

C. Discussion

During the development of the planner, it was noted that the largest impact to the path selected by the planner was the order of the PriorityQueue. By changing the order of the priority queue, different paths ended up getting selected. For example, when changing the sorting of the priority queue to be based on the distance from the goal (h value) as opposed to the total heuristic function (f value), a different path gets found. Figure 8 visualizes the various nodes searched by the planner with the changed sorting approach. It can be seen that much fewer nodes are searched with this approach as it selects the node that is closest to the goal first. Figure 9 visualizes the final path found by this version of the planner. Compared to Figure 7, it is clear that this is not the shortest path overall, but still a valid solution.

Overall, the car was able to successfully reach the goal pose without collisions and following its kinematic constraints. A video of the motion is also available for viewing.

Future work for this aspect of the project would include developing a controller to move the car to the various waypoints found by the planner, similar to the robot. Additionally, exploring how varying heuristic costs for forward versus reverse motions and penalties for being close to obstacles would provide valuable insight on developing an optimal planner.

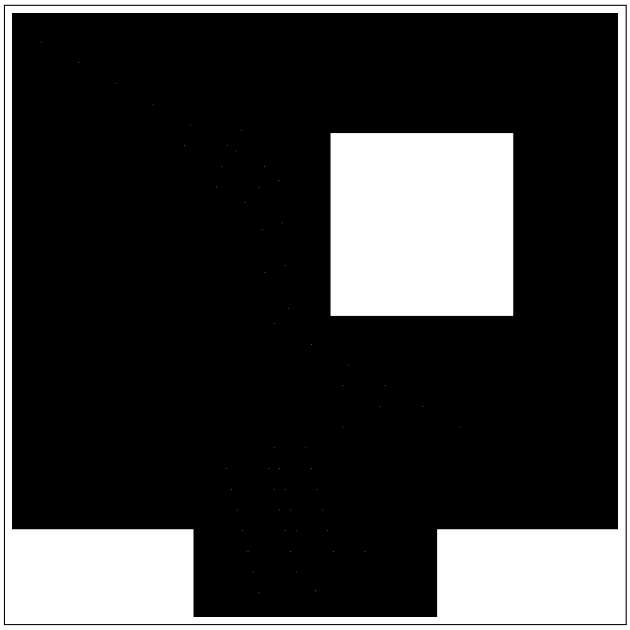

Fig. 8. Path generated by search algorithm.

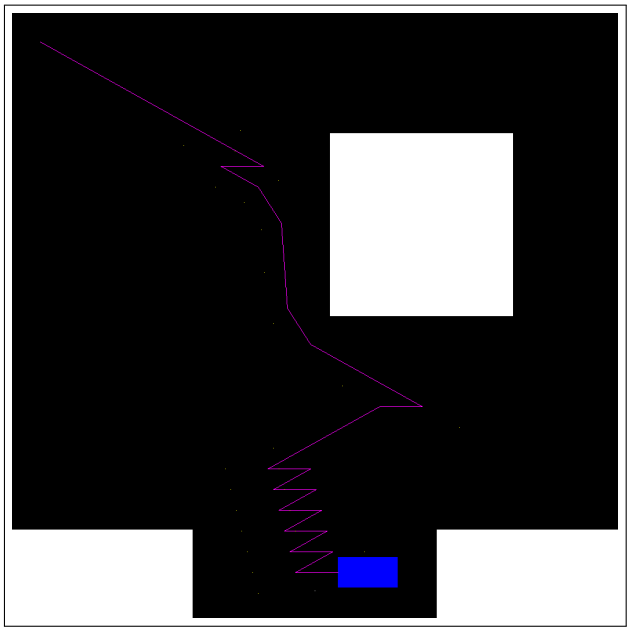

Fig. 9. Path followed by the car.

IV. Truck with Trailer

For the truck with a trailer, the truck also has standard Ackermann steering with a wheelbase of 3.0 m and width of 1.75 m. The distance between the truck and the axle center of the trailer is 5.0 m.

A. Methods

The methods for implementing the truck with trailer kinematics, collision detection, and the planning algorithm will be described.

1) Simulation Scaling: For this vehicle, the same scaling used for the car will be used here. In this case, the axle for the truck are located directly at the end of the rectangle surface. The trailer hitch is connected from the rear axle of the truck - in this case the end of the rectangle - to the center of the trailer. The trailer axle is assumed to be at the center of the trailer length.

For the truck simulation, while the assignment allowed the removal of one of the cars/obstacles, due to the scale of the truck in the simulation, there is enough room for the truck to fit between the two cars.

2) Kinematics: The kinematic model of the truck with a trailer can be seen in Figure 10.

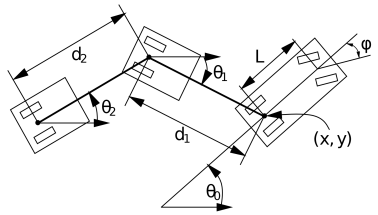

Fig. 10. Kinematic model of a truck with a trailer [1].

In the global coordinate frame, the trucks motion can be represented by the same set of equations for the car, as shown by Equation 5. However, due to the presence of the trailer, the motion of trailer angle needs to be accounted for. Thus, the new set of equations for the truck with the trailer can be seen in Equation 7. These equations as described by Equation 13.19 from the reference textbook, Planning Algorithms [2].

$$\begin{aligned} \dot{x} &= v \cdot \cos(\theta_0) \\ \dot{y} &= v \cdot \sin(\theta_0) \\ \dot{\theta_0} &= \frac{v}{L} \tan(\phi) \\ \dot{\theta_1} &= \frac{v}{d1} \sin(\theta_0 - \theta_1) \end{aligned} \tag{7}$$

Based on this equation, the global position and heading values can be obtained via Equation 8.

$$\begin{aligned} x_{car,new} &= x_{car,previous} + \dot{x} \cdot dt \\ y_{car,new} &= y_{car,previous} + \dot{y} \cdot dt \\ \theta_{0,new} &= \theta_{0previous} + \dot{\theta_0} \cdot dt \\ \theta_{1,new} &= \theta_{1,previous} + \dot{\theta_1} \cdot dt \\ x_{trailer,new} &= x_{car,new} - d_1 \cdot \cos(\theta_{1,new}) \\ y_{trailer,new} &= y_{car,new} - d_1 \cdot \sin(\theta_{1,new}) \end{aligned} \tag{8}$$

3) Collision Detection: The collision detection system used for the car was extended for this application. An additional Surface and mask was developed for the trailer and added to the simulation. During the search process, the algorithm simulates the position and orientation of the vehicle and updates the position of the mask in the screen. At each iteration, a collision check will occur to see if the node will cause any collision between the vehicle and any obstacles.

4) Planning Algorithm: The planning algorithm is effectively the same as described for the car in Algorithm 2. The main difference was adding the calculations for the trailer kinematics from the motion primitives to ensure a node is valid before adding it as a valid child of the expansion.

B. Results

Since the planner was simulating the position of the truck and trailer over time, the path from the start to the goal was captured and the resulting simulation shows a periodic snapshot of the cars motion over time.

Figure 11 visualizes the simulation environment and the various nodes searched by the planner in yellow.

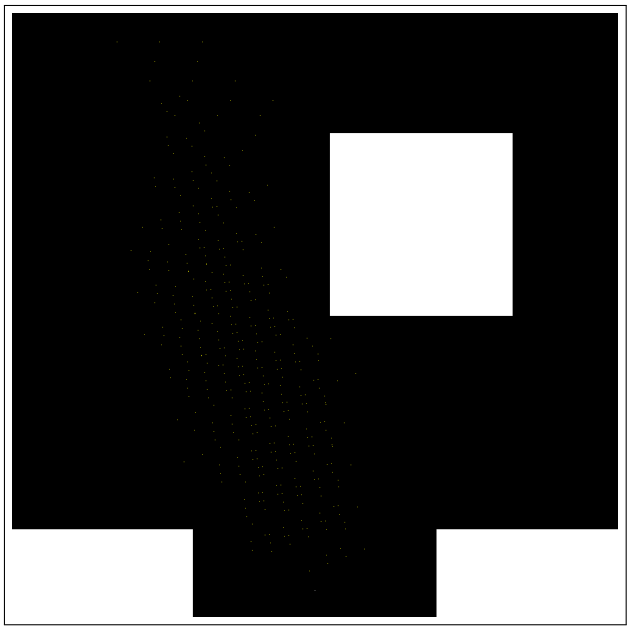

Fig. 11. Path generated by search algorithm.

Figure 12 visualizes the resulting path found by the planner in magenta and the position of the truck and trailer at the end goal.

C. Discussion

As described in the Discussion section for the car, a similar, alternate planner was implemented for the truck and trailer that sorted the PriorityQueue of nodes based on the distance to the goal. In this case, a different path was also found. Figure 13 visualizes the various nodes searched by the planner with the changed sorting approach. It can be seen that much fewer nodes are searched with this approach as it selects the node that is closest to the goal first. Figure 14 visualizes the final path found by this version of the planner. Compared to Figure 12, it is clear that this is not the shortest path overall, but still a valid solution.

Overall, the truck with a trailer was able to successfully reach the goal pose without collisions and following its kinematic constraints. A video of the motion is also available for viewing.

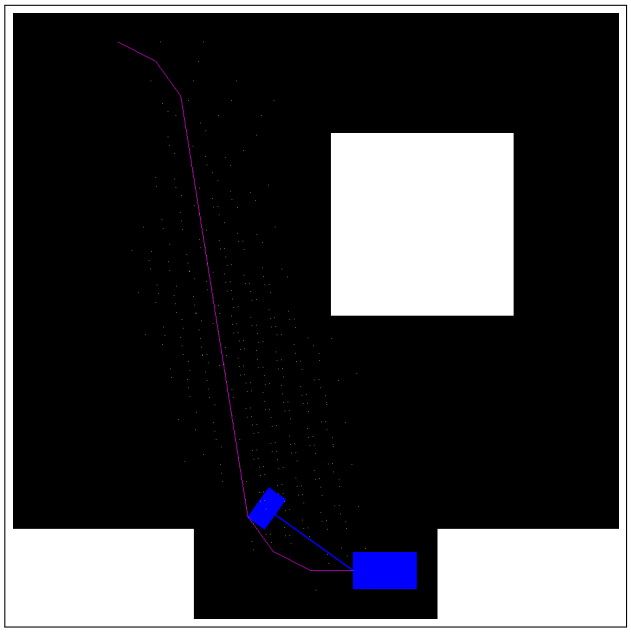

Fig. 12. Path followed by the truck and trailer.

Future work for this aspect of the project would include developing a controller to move the truck with the trailer to the various waypoints found by the planner, similar to the robot. Additionally, exploring how varying heuristic costs for forward versus reverse motions and penalties for being close to obstacles would provide valuable insight on developing an optimal planner.

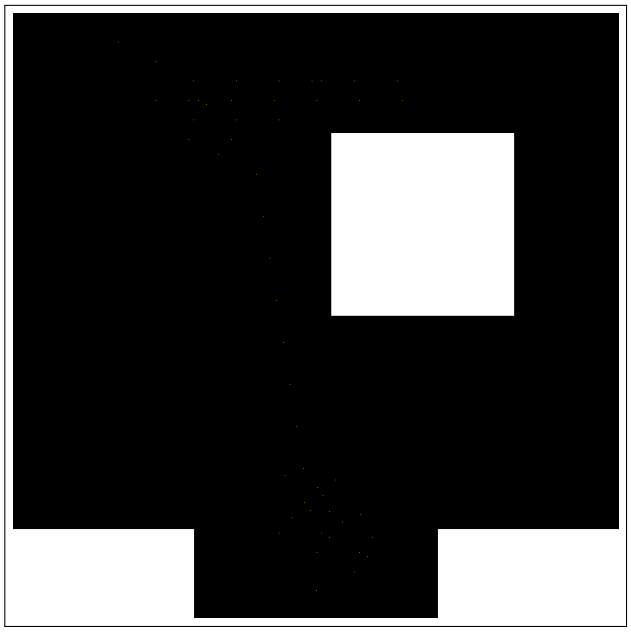

Fig. 13. Path generated by search algorithm.

REFERENCES

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

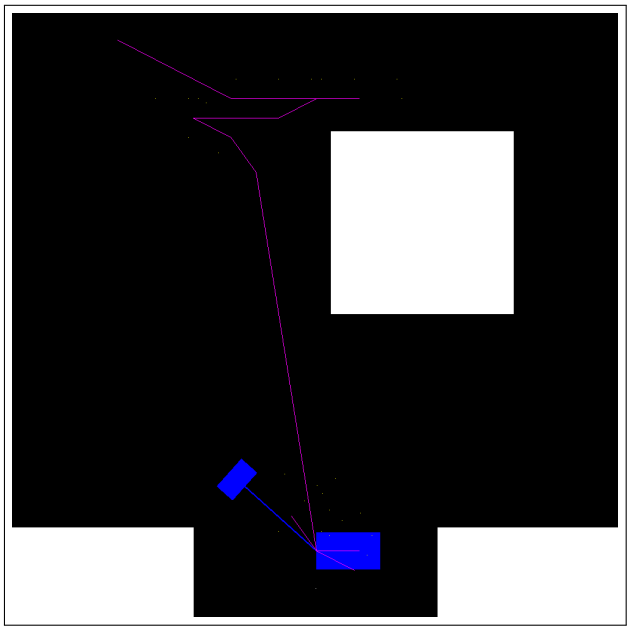

Fig. 14. Path followed by the truck and trailer.

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

[3] "pygame.mask," Pygame.mask - pygame v2.6.0 documentation, https://www.pygame.org/docs/ref/mask.html?highlight=bitmask (accessed Mar. 9, 2024).