16-350, Spring 2019, Final Project Report.
Shashank Ojha (shashano)

1 The Problem

The objective of this project was to write a multi-goal planner for an autonomous vehicle in a parking lot. Given some starting state of the vehicle and some number of desired goal parking spots, the planner should return a plan to reach the goal. This plan might be incorrect in that the environment will be partially known and an obstacle might later be discovered impeding the original calculated plan. At this point, the vehicle will update its knowledge of the world and replan.

1.1 Formal Definition

The state of a vehicle will be defined as (x, y, θ). Note that velocity is omitted from this state since the vehicle will travel at relatively slow velocities in a parking lot. Now given a starting state $(x_{start}, y_{,start}\, \theta_{start})$ and a set of n goal states $\{(x_1, y_{,1}\, \theta_1), (x_2, y_{,2}\, \theta_2), \ldots, (x_n, y_{,n}\, \theta_n)\}$, the planner should return a sequence of steps for the vehicle to make. Steps are defines as tuples of $(dx, dy, d\theta)$ depending on the discretization of the map. The map consists of 2D points with obstacles.

1.2 Actions and Lattice Graph

In order to make motions of the vehicle realistic, not all steps are valid. For example, the vehicle cannot turn in place with the step $(0, 0, d\theta)$. To model this, we will define a action template for the vehicle. The vehicle will be able to make 8 total actions: **(1)** Forward, **(2)** Forward Sharp Left, **(3)** Forward Slight Left, **(4)** Forward Sharp Right **(5)** Forward Slight Right, **(6)** Backward, **(7)** Backward Slight Left, **(8)** Backward Slight Right. How these are exactly defined are left to implementation.

2 Planner Implementation

There were many design choices that were made with respect to the construction of the graph and action template that were made for this project. I will go through each one one by one.

2.1 Map

The map was defined to be a 2x2 grid. Different dimensions of the graph were possible. To define a map, I simply made a file consisting of N by N bits with 0's representing clear areas and 1 representing obstacles. The more bits there were, the more dense the graph was. The size of the graph had no impact on the size of the vehicle. All the maps I worked with, however, were 40 x 40 cells.

2.2 Action Template

The 8 actions above were defined as classes. Each class had two functions: a cost function and an effects function. The cost function simply returned an integer representing the cost of the action. Backwards actions were more expensive then forward actions. The effects function took in a state (x, y, θ) and output a sequence of steps $(dx, dy, d\theta)$. This was necessary because actions could mean different things depending on the original state. For example, forward while facing North only changes the y position, whereas forward while facing East only changes the x position. The number of grid cells moved was current set to 2.

In addition, the orientation of the vehicle must have been in a multiple of 22.5 degrees. However, no actions actually results in the orientation being a multiple of 22.5 degrees. The end effect of any action resulted in the vehicle at an orientation in the multiple of 45 degrees. Some actions such as Slight Left however outputted two steps. For example, if the vehicle was facing East, the Slight Left action would first move the vehicle East by 2 cells and also update the orientation to 22.5 degrees and then another step would move the vehicle North East and then update the orientation to 45 degrees. These two steps are just to make the visualization more natural.

A pictorial version of the action template can be found in Figure 1 below.

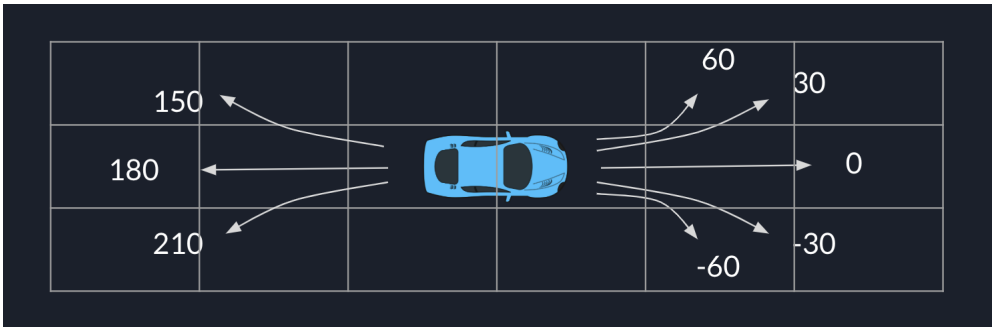

Figure 1: Vehicle Action Template

2.3 Obstacle Checking

In order to perform obstacle checking on the vehicle, I estimated the vehicle using a bunch of points and rotated each of those points around the center of mass by θ, where θ was the current orientation of the vehicle. Formally, for each point (x_i, y_i) that represented the vehicle, I performed the following matrix transformation:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cost(\theta) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} x_i' \\ y_i' \end{bmatrix}$$

This yielded the point (x_i', y_i'), which can then be checked against the map to see if it collided with an obstacle. The process is also described pictorially in Figure 2.

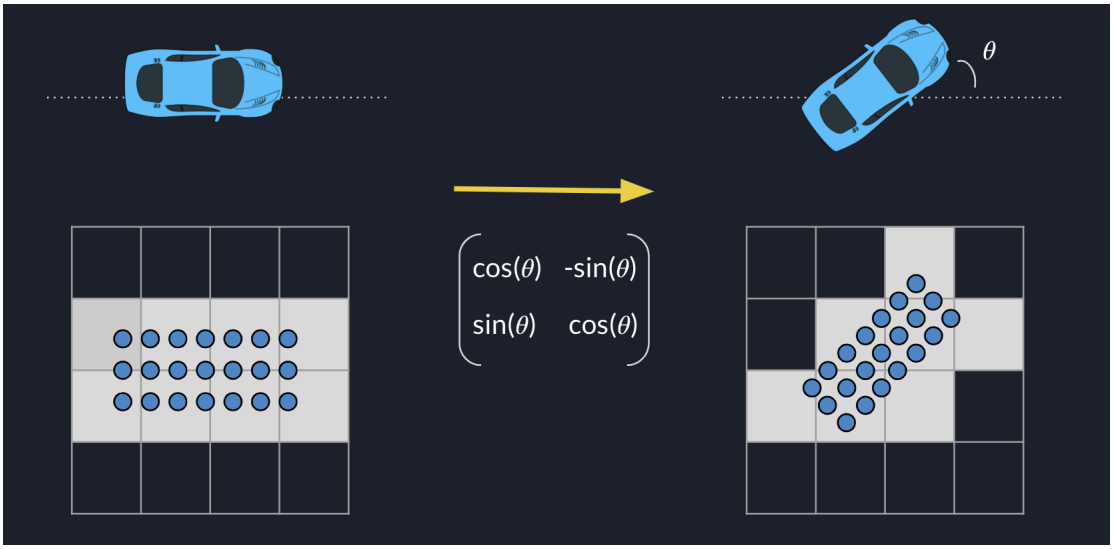

Figure 2: Obstacle Checking Process

2.4 Algorithm

The algorithm I use is simply a series of Forward Weighted A* searches. The vehicle starts out under the assumption that there are no obstacles in the graph and generates a plan using an Forward Weighted A* search on the lattice graph and action template described above. The vehicle then executes the plan updating it's model of the world as it goes. It can see up to a certain radius of grid cells around its center of mass. When obstacles are discovered, it checks whether the current plan is no longer feasible due to collision with those obstacles. If there are no issues, then the planner continues to the next step. If there is a collision, then the planner replans using its updated model of the world with a fresh Forward Weighted A* search. This process is repeated until the vehicle either reaches it's goal or the search explores its entire search space and no path is found.

In order to support multi-goals, Weighted A* search was simply modified to check if the expanded state is any one of the goal states.

The heuristic used was euclidean distance to the goal. When multiple goals were possible, the heuristic was simply the max of euclidean distances toward each goal. Note that this heuristic is still consistent and admissible.

3 Results

First of all it important to note that Weighted A* was necessary for this planner. A* by itself took 30 seconds to execute with just a single goal, while Weighted A* with $\alpha = 2$ took only 0.8 seconds. The number of steps executed by the vehicle was the same despite the inconsistent heuristic introduced by weighted A*. Note that this was on just a 40x40 grid. With larger maps, regular A* would be infeasible. Dijiktra's did not complete.

A summary of the results are in the table below.

Algorithm Performance				
Algorithm	Expanded States	Number of Replans	Steps Executed	Total Planning Time (sec)
Dijkstra's	N/A	NA	NA	NA
A*	26,273	15	42	34.84
Weighted A* ($\alpha = 2$)	508	17	42	0.79

4 Demo

Running the code on your own machine:

1. **Required Software**: You need to download Python3.6 and install numpy, PIL, and tkinter using pip.
2. **Running the code**: The format is

```
python3 visualizer.py [alpha] [map filename] --full

OR

python3 visualizer.py [alpha] [map filename] --partial
```

The first version is for when the vehicle has full knowledge of the environment. The second version is if the vehicle only has partial knowledge and builds it's model of the environment as it goes. The α parameter is change the approximation factor of the weighted A*. Note $\alpha = 1$ is just normal A* and $\alpha = 0$ is just Dijkstra's.

Below are some examples:

Video Link: `https://youtu.be/-7SltOEq6jw`

```
python3 visualizer.py 10 40x40map2.txt --full
```

Video Link: `https://youtu.be/QajIyY5khMo`

```
python3 visualizer.py 1.5 40x40map3.txt --partial
```

Video Link: `https://youtu.be/mQGr0lOuu8s`

```
python3 visualizer.py 1.7 40x40map4.txt --partial
```

Note that the format of the start and end goals is a little more involved. For the purposes of this project, I just have another goal config variable commented out in case you want to run the multigoal version of the program. It can be found at the top of the visualizer.py program. It is much slower. I've attached a video of multigoal version below as well.

Video Link: `https://youtu.be/ESH9Xvt-dHI`