Installing and Running the Simulation:

Running the simulation requires Python3.x/Python 2.x with pygame installed.

Installation:

For Windows and Linux System : Follow the anaconda python installation guide provided before to install anaconda python. After that install pygame by typing

pip install pygame.

For MacOS:

Type: pip install pygame

If the above does not work try following steps in the document "pygame_mac_Installation.pdf"

Running the simulation:

To run the program run python SearchAgent.py <config_file_name>

For example python SearchAgent.py dynamic_config1.txt

On the gui your car is the white color car.

There are 5 example config files provided for your testing. dynamic_config<number>.txt is the config file which contains the starting location of your car and the number and the starting location of other cars on the road. Your assignment will be evaluated against a different set of config files.

==========

We describe the simulator in which you have to write the code. There are 5 files provided.

1. Simulator.py -contains the code for the gui. **Do not modify this file.**
2. Environment.py – contains the code for environment and movement of other cars. **Do not modify this file.**
3. searchUtils.py – contains utility functions for search algorithms. You may use these utility functions. You can add/modify these utility functions as per your requirement.
4. randomagent.py – contains the example code for a car which takes a random action at each timestep.
5. SearchAgent.py – contains the basic code for your car. ***This is the file you need to modify***.

You need to **write the drive function in the SearchAgent.py** which returns the sequence of actions to be taken by the car. This function is called at each step.

The drive function takes as input the python list of a goal states (with only one element) and the inputs sensed from the environment. On calling env.sense it returns the status of the grid as seen by the car. 0 if the cell is clear, 1 if there is any other car present and -1 if the information is not known to your self driving car (i.e. the cell is not in the visibility range of the car).

As an example, the drive function can implement A search from the car's current location to the goal state and stores the action sequence.*

The update function in SearchAgent class is called at each timestep. Your algorithm should choose the action in the update function based on the action sequence generated using the drive function.

The intended action is conveyed to the environment using act function which returns the updated state of the car and also updates the state in the environment.

Environment first updates the position of your car before updating the position of any other car.

Following are the class variables for the agent class:

valid_actions – list of valid actions available for your car.

Env – Instance of the environment class.

Searchutil – Instance of the searchUtils class

state – dictionary representing state of your car. State["location"] – provides the location of your car in grid cell. For eg. If state["location"] = (2,3) it means that car is located in the cell 2,3.

Following are the functions which you can use to write the algorithm for self driving car to choose an action at each timestep.

Environment class functions – can be accessed by calling self.env.<functionname>

1. getGoalStates() – returns the list of goal states.
2. act(car,a) – takes action a for the car and returns the new state. Updates the position of the car in the environment.
3. sense(car) – returns the grid cell status as seen by the car. 1 if another car is present, 0 if the cell is free and -1 if the cell is not visible to your car.
4. applyAction(car,s,a) – returns the location where the car will move on applying action a in state s based on the inputs sensed by your car. Does not execute the action and does not update the state of the car in the environment.
5. getAction(s1,s2) – returns the action which takes a car from state s1 to state s2. The purpose of this function is to help you in retrieving the action sequence if you have a sequence of states visited after calling applyaction/act function. Please note that this function will not check for presence of other cars so a getAction will return forward if you pass it states (3,0) and (3,1) even if there is a car in (3,1). You need to use applyAction function to get the location where car will move on applying given action. You can also use the information about the presence/absence of other cars from the sense method.

Note: whenever you need to pass car as argument, such as in applyAction(car,s,a) you can pass self for car i.e, calling applyAction(car,s,a) as applyAction(self, s,a). You can see these behavior in provided dummy code "SerchAgent.py->update function" for act(car,a) and sense(car).

Searchutils class functions – Searchutils class contains utility functions for the search algorithm can be accessed by calling self.searchutil.<functionname>

1. retrieveActionSequenceFromState(s) – retrieve the sequence of actions taken to reach the give state s. The action sequence is updated each time applyAction is called on env.

2. isPresentStateInList(state,searchlist) – returns 1 if the state is present in searchlist
3. isPresentStateInPriorityList(state,searchlist) – returns 1 if the state is present in priority list searchlist.
4. insertStateInPriorityQueue(searchList,state,distanceToGoal) – insert state with cost distancetogoal in the searchList.
5. checkAndUpdateStateInPriorityQueue(searchList,state,distanceToGoal): checks if the state is present in the searchList with a cost higher than distanceToGoal. If the existing cost is higher then state is reinserted with the cost distancetoGoal in the searchList.