Final Exam Project

**Introduction**

In this project, you will implement the agent programs for a simple reflex agent, an expectimax agent and a q-learning agent. You will test your agents on a grid-based simulation of an autonomous driving domain.

The code for this project contains the following files, available as a zip archive called final\_exam.zip on canvas.

|  |  |
| --- | --- |
| **Files you will edit:** |  |
| agent.py | A file that implements different agent classes. This is the file where you will implement your code for different agents. |
| features.txt | You will specify the number of features defined for your learning agent in this file |
|  |  |
| **Files you should read but NOT edit:** |  |
| driving.py | The main file to run in this project. |
| problem.py | This file implements the controllers for different agent types. |
| state.py | This is an important file for your reference. This file consists of all the helper functions you will need to implement your agent program. This is your interface to the environment. |
| environment.py | This is the file that defines the autonomous driving environment and implements the simulator functions. You may not access the attributes and methods of this class directly. |
| autograder.py | This is the file that grades your project. |
|  |  |
| **Files you can ignore:** |  |
| test.json | This file has the test cases encoded for your project. |

**Files to Edit and Submit:** You will fill in portions of agent.py during the assignment and features.txt where you will specify the number of features you have defined. Please do not change the other files in this distribution or submit any of our original files other than these files.

**Python**

This project is developed under 3.7 but should also work under 3.6. If you encounter any python issues, please let us know on Ed Discussion.

**Libraries**

This project uses the numpy library that may not be available by default in a python installation. You may have to install it prior to beginning your project. Please see instructions to install it below:

* <https://numpy.org/install/>

**Visualization**

The autonomous driving environment is a grid-based discretized simulation of driving on a highway as shown below.



The autonomous agent (A) is indicated by the character ‘A’.

The other cars on the road are indicated by the character ‘^’.

The direction of movement of all cars is from the bottom of the grid to the top of the grid. The goal is to have your agent drive successfully outside the top row of the road without crashing into other cars or moving out of the sides of the road (which is also considered as a crash).

**General Information on the Simulator**

At the start of the program, the environment is initialized with an autonomous driving agent (referred to as A throughout this project) and some other cars at random locations on the road.

The position of a car on the grid is represented by (r, c) where, r indicates the row (counting from 0, starting from the bottom to the top of the grid) and c indicates the column of the grid (counting from 0, starting from the left to the right of the grid).

At every step, your agent A and other cars take an action from the list of available actions: move forward (F), move left (L), move right (R), and wait (W). These available actions are all considered legal for your agent at any given step. However, the legal actions for any other car depend on its surroundings at a given step. The movement of the other cars (i.e., all cars except for your agent) are random but they will not crash into each other. However, they may crash into your agent A.

When a car crosses/moves out of the top of the road, its position updates to None, where None is a keyword in python indicating no value. Essentially, the car does not enter the road again. Also, no new cars are deployed in the environment once the environment is initialized.

At any step, if there is a crash or if your agent has successfully reached the top end of the road, the simulator terminates.

**Getting Started**

To get started, run driving.py in manual control mode which uses keyboard inputs.

*python driving.py --agent manual*

The commands available for manual control are:

F: to move forward

L: to move left

R: to move right

W: to wait in the same location

S: to stop the simulation (this is a command available only in manual control and is not a legal action)

Note: The action commands are case sensitive i.e., only upper-case characters will be recognized. For example, to move forward press shift+f.

Another option is to observe a random agent drive. Since the default agent type is random, you can simply run the following.

*python driving.py*

Once the simulator has started you may press enter to step through the environment.

**Input arguments**

|  |  |  |
| --- | --- | --- |
| --agent | String | Available agent types are manual, random, reflex, expectimax, and learning. The autonomous agent (A) is initialized as the specified agent type. |
| --height | Integer | Specify the length of the road (discretized distance). Default is 10. |
| --width | Integer | Specify the width of the road (number of lanes). Default is 4. |
| --occupancy | Float | This argument takes a value between 0 and 0.3. Specify the percentage w.r.t. the number of grid cells that must populated with other cars. For example, when it is 0, there are no other cars on the road and when it is 0.2, 20% of the cells are occupied with other cars. Default is 0.1. |
| --init | Float | This argument takes a value between 0 and 0.9. Specify the starting location of A w.r.t. the length of the road. For example, when it is 0.1, your agent A is initialized somewhere near the bottom of the grid and when it is 0.9, it is initialized somewhere near the top of the grid. Default is 0.2. |
| --seed | Integer | This seed is used to populate other cars on the road randomly. A given seed will result in the same initialization of other cars on the road. Default is 3. |
| --episodes | Integer | Specify the number of episodes you wish to train the agent on. This argument is used by the learning agent only. Default is 50000. |
| --features | Integer | Specify the number of features you wish to define for your learning agent. If this argument is not passed and there is a mismatch in the number of features defined, the code will throw an error. Default is 4. |

You may customize the road by using the height, width and seed arguments described above.

**Running the autograder**

The autograder script takes the following arguments:

|  |  |  |
| --- | --- | --- |
| --q | String | The question you wish to be grades. Enter q1, q2, or q3. Default is None. |
| --verbose | Bool | Specify True if you wish to print the console output for each test case. Default is False. |
| --features | Integer | Specify the number of features you wish to define for your learning agent. If this argument is not passed and there is a mismatch in the number of features defined, the code will throw an error. Default is 4. |

Some examples of running the autograder:

*python autograder.py --q q1*

*python autograder.py --q q2 –verbose True*

*python autograder.py --features 5*

**Question 1 (5 points): Simple Reflex Agent**

﻿A reflex agent chooses an action at each choice point by following reactive rules given the current percept.

A percept is a (3 x 3) sized grid which is a localized view of the road with the agent A as the center of the grid with the following markers.

0: empty cell

1: other cars

2: autonomous agent (A)

-1: edge of the road (left, right and bottom edge of the road)

10: end of the road (top edge of the road)

Following are some of the examples of the percept.

|  |  |
| --- | --- |
| How the percept is displayed | Corresponding markers |
|  | -1 0 0  -1 2 0  -1 0 0 |
| A grid with white lines  Description automatically generated | 0 0 0  0 2 0  0 0 0 |
|  | 1 1 -1  0 2 -1  0 0 -1 |
|  | 10 10 10  0 2 -1  0 0 -1 |

Like the road grid, the percept grid is also indexed as (r, c).

For example,



In the above percept, your agent A is at (1, 1), the other cars are present at (0, 0) and (0, 1). The cells (0, 2), (1, 2) and (2, 2) have the value -1 indicating that A is at the edge of the road. In such a situation, moving right would lead your agent A to a crash and out of the road.

Note that the variable self.road in environment.py which is the whole 2D representation of the road (shown above) does not have markers (‘x’ or ‘G’) to show boundaries of the road. This is exclusively a representation used for the percept to maintain a 3 x 3 input given any location of ‘A’. To clarify, we have shown the boundaries of the road in bold white lines.

You are required to write your code in the get\_action function of the ﻿ReflexAgent class given the current percept.

The get\_action function is required to return an action among the legal actions (‘F’, ‘L’, ‘R’, ‘W’).

Hint: Implement a bunch of if-elif rules to govern the behavior of your reflex agent.

Note that the reflex agent cannot see beyond the sensing range nor has access to the percept history.

Test your code by running:

*python driving.py --agent reflex*

**Question 2 (10 points): Expectimax Agent**

﻿An expectimax agent chooses an action at each choice point based on the expectimax algorithm. Essentially, an expectimax agent, will take the expectation according to your agent A’s model of how the other cars move. To simplify your code, assume you will only be running against other cars which choose amongst their getLegalActions uniformly at random.

For this agent, you are required to write the get\_action function and the evaluation function of the ExpectimaxAgent class given the current road state.

The get\_action function is required to return an action among the legal actions (‘F’, ‘L’, ‘R’, ‘W’) and the evaluation\_function is required to return a score (float).

To help build the code, a bunch of helper functions are provided in the state.py file, specifically, in the ExpectimaxState class.

The default depth for the expectimax tree is 2.

Test your code by running:

*python driving.py --agent expectimax*

**Question 3 (10 points): Learning Agent**

A learning agent in this project is an approximate Q learning agent. An approximate Q learning agent chooses an action at each choice point based on the weights it has learned over the features extracted.

Unlike the pacman project, where the features were extracted for you, in this project you will implement the get\_features function of the Learning Agent class. The get\_features function is passed a state object which is an instance of the LearningState class. The LearningState class defines a set of helper functions for you to access the attributes of the environment such as the self.road, self.cars, self.height and self.width. These attributes are your friends when writing the get\_features function. For example, one of the features that you may find useful would be distance to the closest car on the road. Another feature that might be helpful is whether the agent chose an action that leads to a collision. Note that these attributes should be accessed only via the helper functions defined in the LearningState class.

In addition to writing the get\_features function you will also implement the ﻿get\_Q\_value, ﻿compute\_max\_Q\_value, and the ﻿update functions of LearningAgent class in agent.py.

Note: when writing your ﻿compute\_max\_Q\_value function, be aware to check for terminal states (crash or successfully crossing the top row of the road). You needn’t worry about checking for terminal states in your get\_features function or ﻿get\_Q\_value functions.

Test your code by running:

*python driving.py --agent learning --features <enter-#-features>*

**Testing**

Remember to test your agent for edge cases such as ‘A’ surrounded by cars in all directions, agent starting in the first, middle, or last row of the road, high occupancy of cars on the road, etc.