**CPSC 481 Artificial Intelligence**

**Project 3 – Game**

**Mode**: team of up to four

**Due Date**: 12/5/2021

**Pacman with ghosts**

In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement minimax search and try your hand at evaluation function design.

The code base has not changed much from the previous project, but please start with a fresh installation, rather than intermingling files from project 1.

As in project 1, this project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

*python autograder.py*

Note: If your python refers to Python 2.7, you may need to invoke ***python3 autograder.py*** (and similarly for all subsequent Python invocations) or create a conda environment as described in Project 0.

It can be run for one particular question, such as q2, by:

*python autograder.py -q q2*

It can be run for one particular test by commands of the form:

*python autograder.py -t test\_cases/q2/0-small-tree*

By default, the autograder displays graphics with the -t option, but doesn’t with the -q option. You can force graphics by using the --graphics flag, or force no graphics by using the --no-graphics flag.

See the autograder tutorial in Project 0 for more information about using the autograder.

The code for this project contains the following files, available as a zip archive, **multiagent.zip**

**Files you'll edit:**

multiAgents.py Where all of your multi-agent search agents will reside.

**Files you might want to look at:**

pacman.py The main file that runs Pacman games. This file also describes a Pacman GameState type, which you will use extensively in this project.

game.py The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

util.py Useful data structures for implementing search algorithms. You don't need to use these for this project, but may find other functions defined here to be useful.

Supporting files you can ignore:

graphicsDisplay.py Graphics for Pacman

graphicsUtils.py Support for Pacman graphics

textDisplay.py ASCII graphics for Pacman

ghostAgents.py Agents to control ghosts

keyboardAgents.py Keyboard interfaces to control Pacman

layout.py Code for reading layout files and storing their contents

autograder.py Project autograder

testParser.py Parses autograder test and solution files

testClasses.py General autograding test classes

test\_cases/ Directory containing the test cases for each question

multiagentTestClasses.py specific autograding test classes

**Files to Edit and Submit**: You will fill in portions of **multiAgents.py** during the assignment.

**Evaluation**: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder’s judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

**Academic Dishonesty**: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else’s code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don’t try. We trust you all to submit your own work only; please don’t let us down. If you do, we will pursue the strongest consequences available to us.

**Proper Dataset Use**: Part of your score for this project will depend on how well the models you train perform on the test set included with the autograder. We do not provide any APIs for you to access the test set directly. Any attempts to bypass this separation or to use the testing data during training will be considered cheating.

**Getting Help**: You are not alone! If you find yourself stuck on something, office hours, section, and the discussion forum are there for your support; please use them. If you can’t make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don’t know when or how to help unless you ask.

**Discussion**: Please be careful not to post spoilers.

**Welcome to Multi-Agent Pacman**

First, play a game of classic Pacman by running the following command:

*python pacman.py*

and using the arrow keys to move. Now, run the provided ReflexAgent in multiAgents.py

*python pacman.py -p ReflexAgent*

Note that it plays quite poorly even on simple layouts:

*python pacman.py -p ReflexAgent -l testClassic*

Inspect its code (in multiAgents.py) and make sure you understand what it’s doing.

**Question 1: Reflex Agent**

Improve the **ReflexAgent** in multiAgents.py to play respectably. The provided reflex agent code provides some helpful examples of methods that query the GameState for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the testClassic layout:

*python pacman.py -p ReflexAgent -l testClassic*

Try out your reflex agent on the default mediumClassic layout with one ghost or two (and animation off to speed up the display):

*python pacman.py --frameTime 0 -p ReflexAgent -k 1*

*python pacman.py --frameTime 0 -p ReflexAgent -k 2*

How does your agent fare? It will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good.

Note: Remember that newFood has the function **asList()**

Note: As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.

Note: The evaluation function you’re writing is evaluating state-action pairs; in later parts of the project, you’ll be evaluating states.

Note: You may find it useful to view the internal contents of various objects for debugging. You can do this by printing the objects’ string representations. For example, you can print newGhostStates with **print(newGhostStates**).

Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using **-g DirectionalGhost**. If the randomness is preventing you from telling whether your agent is improving, you can use -f to run with a fixed random seed (same random choices every game). You can also play multiple games in a row with -n. Turn off graphics with -q to run lots of games quickly.

Grading: We will run your agent on the openClassic layout 10 times. You will receive 0 points if your agent times out, or never wins. You will receive 1 point if your agent wins at least 5 times, or 2 points if your agent wins all 10 games. You will receive an addition 1 point if your agent’s average score is greater than 500, or 2 points if it is greater than 1000. You can try your agent out under these conditions with

*python autograder.py -q q1*

To run it without graphics, use:

*python autograder.py -q q1 --no-graphics*

**Plz. Ignore questions 2-5 for this project.**

**Grading for this project:**

There is a project 3 rubric attached below project 3 description on Canvas.

Your program needs to pass autograder tests.

You can find the test cases located in "test\_cases" folder.

**Deliverables:**

**Include team members in a readme file**.

Submit multiAgents.py and readme to the submission link.

One team only submit ONE copy. Pick a team member who will be responsible for submitting the project.

Other members do not need to submit. Canvas may mark your project as "Missing", but you do not need to worry about it.

I'll contact you if I cannot find your name in any readme files.