Computer Science 384
St. George Campus

February 4, 2019
University of Toronto

Homework Assignment #2: Multi-Agent Pacman
Due: February 26, 2019 by 10:00 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 3 grace days.

Overview: Assignment #2 asks you to implement four agents for the Pacman assignment as well as an improved evaluation function.

Total Marks: This assignment has a total of 50 marks and represents 11% of the course grade.

What to hand in on paper: Nothing.

What to submit electronically: You must submit your assignment electronically. Download the assignment files from the A2 web page. Modify `multiAgents.py` appropriately so that it solves the problems specified in this document. You will submit the following files:

- `multiAgents.py`
- `acknowledgment_form.pdf`

How to submit: If you submit before you have used all of your grace days, you will submit your assignment using MarkUs. Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time, though the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission. More detailed instructions for using Markus are available at: http://www.teach.cs.toronto.edu/~csc384h/winter/markus.html.

- *Make certain that your code runs on teach.cs using python3 (version 3.7) using only standard imports.* This version is installed as python3 on teach.cs. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 2 Clarification page:
http://www.teach.cs.toronto.edu/~csc384h/winter/Assignments/A2/a2_faq.html.
You are responsible for monitoring the Assignment 2 Clarification page.

Questions: Questions about the assignment should be posted to Piazza:
https://piazza.com/utoronto.ca/winter2019/csc384/home.

Introduction

Acknowledgements: This project is based on Berkeley's CS188 EdX course project. It is a modified and augmented version of "Project 2: Multi-Agent Pacman" available at http://ai.berkeley.edu/multiagent.html.

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

- Files you will edit or create:
  - **multiAgents.py** - Suitably augment this with your implementation of the search agents described in tasks 1 to 3 in this handout.
- Files that are useful to your implementations:
  - **pacman.py** - Runs Pacman games. This file also describes a Pacman GameState, which you will use extensively in your implementations.
  - **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.
- Files that are unlikely to be of help to your implementations, but are required to run the project.
  - **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.
  - **multiagentTestClasses.py** - Project specific autograding test classes.
  - **grading.py** - File that specifies how much each question in autograder is worth.
  - **projectParams.py** - Project parameters, facilitates nicer output during autograding.
  - **pacmanAgents.py** - Classes for specifying ghosts' behaviour.

The Pacman Game

Pacman is a video game first developed in the 1980s. A basic description of the game can be found at https://en.wikipedia.org/wiki/Pac-Man. In order to run your agents in a game of Pacman, and to evaluate your agents with the supplied test code, you will be using the command line. To familiarize yourself with running this game from the command line, try playing a game of Pacman yourself by typing the following command from within the `./multiagent` subfolder:

```
python pacman.py
```

To run Pacman with a game agent use the `-p` command. Run Pacman as a `GreedyAgent`:

```
python pacman.py -p GreedyAgent
```

You can run Pacman on different maps using the `-l` command:

```
python pacman.py -p GreedyAgent -l testClassic
```

Important: If you use the command in an environment with no graphical interface (e.g. when you `ssh` to teach.cs, formerly CDF), you must use flags `-t` or `-q`, which suppress graphical output.

The following commands are available to you when running Pacman. They are also accessible by running `python pacman.py --help`.

- `-p` Allows you to select a game agent for controlling pacman, e.g., `-p GreedyAgent`. By the end of this project, you will have implemented four more agents, listed.
  - `ReflexAgent`
  - `MinimaxAgent`
  - `AlphaBetaAgent`
  - `ExpectimaxAgent`
- `-l` Allows you to select a map for playing Pacman. There are 9 playable maps, listed.
  - `minimaxClassic`
  - `trappedClassic`
  - `testClassic`
  - `smallClassic`
  - `capsuleClassic`
  - `openClassic`
  - `contestClassic`
  - `mediumClassic`
  - `originalClassic`

- `-a` Allows you to specify agent specific arguments. For instance, for any agent that is a subclass of `MultiAgentSearchAgent`, you can specify the depth that you limit your search tree by typing `-a depth=3`
- `-n` Allows you to specify the amount of games that are played consecutively by Pacman, e.g., `-n 100` will cause Pacman to play 100 consecutive games.
- `-k` Allows you to specify how many ghosts will appear on the map. For instance, to have 3 ghosts chase Pacman on the `contestClassic map`, you can type `-l contestClassic -k 3`
  Note: There is a max number of ghosts that can be initialized on each map, if the number specified exceeds this number, you will only see the max amount of ghosts.
- `-g` Allows you to specify whether the ghost will be a `RandomGhost` (which is the default) or a `DirectionalGhost` that chases Pacman on the map. For instance, to have DirectionalGhost characters type `-g DirectionalGhost`
- `-t` Allows you to run Pacman with text graphics.
- `-q` Allows you to run Pacman with no graphics.
- `--frameTime` Specifies frame time for each frame in the Pacman visualizer (e.g., `--frameTime 0`).

An example of a command you might want to run is:

```
python pacman.py -p GreedyAgent -l contestClassic -n 100 -k 2 -g DirectionalGhost -q
```

This will run a GreedyAgent over 100 cases on the `contestClassic` level with 2 `DirectionalGhost` characters, while supressing the visual output.

Important Note: To run the autograder for all questions, run the following command:

```
python autograder.py
```

Note that the provided starter code also contains testcases. The autograder will run all tests.
In order to prevent `autograder.py` from displaying graphics, you can pass the `--no-graphics` argument.

Question 1 : Reflex Agent (8 points)

Don't spend too much time on this question, as the meat of the project lies ahead.

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: 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 (i.e., how good is it to perform this action in this state); in later parts of the project, you'll be evaluating states (i.e., how good is it to be in this state).

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.

To test with the autograder, you may run:

```
python autograder.py -q q1
```

Question 2: Minimax (10 points)

Here you will implement a minimax search agent for the game of Pacman. The function `getAction` in the `MinimaxAgent` class can be found in `multiAgents.py`. Your minimax search must work with any number of ghosts. Your search tree will consist of multiple layers - a max layer for Pacman, and a min layer for each ghost. For instance, for a game with 3 ghosts, a search of depth 1 will consists of 4 levels in the search tree - one for Pacman, and then one for each of the ghosts.

Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. You shouldn't change this function now, but recognize that now we're evaluating *states* rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

You will have to implement a depth-bound specified in `self.depth`, so the leaves of your minimax tree could be either terminal or non-terminal nodes. Terminal nodes are nodes where either `gameState.isWin()` or `gameState.isLose()` is true. However, the leaves of your tree search might also be non-terminal nodes. You can run your `MinimaxAgent` on a game of Pacman by running the following command:

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=3
```

To test with the autograder, you may run:

```
python autograder.py -q q2
```

Hints and Observations

- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behaviour, it will pass the tests.
- Pacman is always agent 0, and the agents move in order of increasing agent index.
- All states in minimax should be GameStates, either passed in to getAction or generated via GameState.generateSuccessor. In this project, you will not be abstracting to simplified states.
- On larger boards such as openClassic and mediumClassic (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. Don't worry if you see this behavior, Q4 and Q5 will clean up these issues.

Question 3: Alpha-Beta Pruning (10 points)

You will now implement a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree. The function `getAction` in `AlphaBetaAgent` is found in `multiAgents.py`. Again, your algorithm must use the depth-bound specified in `self.depth` and evaluate its leaf nodes with `self.evaluationFunction`. Watch your agent play by running the following command:

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

To test and debug your code, run

```
python autograder.py -q q3
```

The correct implementation of alpha-beta pruning will lead to Pacman losing the game in some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

Question 4: Expectimax (10 points)

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question you will implement the `ExpectimaxAgent`, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices. The only difference in implementation of Expectimax search and Minimax search is that, at a min node, Expectimax search will return the average value over its children as opposed to the minimum value.

As with the search and constraint satisfaction problems covered in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees. You can debug your implementation on small the game trees using the command:

```
python autograder.py -q q4
```

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly.

To see how the ExpectimaxAgent behaves in Pacman, run:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

The correct implementation of Expectimax will lead to Pacman losing some games in some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

Question 5: Evaluation Function (12 points)

Write a better evaluation function for Pacman in the provided function, `betterEvaluationFunction`. Your evaluation function should evaluate states (in contrast to the function employed by your reflex agent, which evaluated actions). You may use any tools at your disposal for evaluation. With depth 2 search, your evaluation function should clear the `smallClassic` layout with one random ghost more than half the time and still run at a reasonable rate. (To get full credit, Pacman should be averaging around 1000 points when he's winning.)
`python autograder.py -q q5`
Grading: the autograder will run your agent on the `smallClassic` layout 10 times. We will assign points to your evaluation function in the following way:

- If you win at least once without timing out the autograder, you receive 1 points. Any agent not satisfying these criteria will receive 0 points.
- +1 for winning at least 5 times, +2 for winning all 10 times
- +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)

- +1 if your games take on average less than 30 seconds on the autograder machine. The autograder is run on the teach.cs machines which have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs). You can use your teach.cs login to run your program on the teach.cs machines.
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

Hints and Observations

- As for your reflex agent evaluation function, you may want to use the reciprocal of important values (such as distance to food) rather than the values themselves.
- One way you might want to write your evaluation function is to use a linear combination of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values and adding the results together. You might decide what to multiply each feature by based on how important you think it is.

HAVE FUN and GOOD LUCK!