Computer Science 384
St. George Campus

Friday, March 2, 2018
University of Toronto

Homework Assignment #3: Multi-Agent Pacman
— Part I —
Due: March 14, 2018 by 11:59 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 #3 is comprised of two parts. Part I asks you to implement three agents for the Pacman assignment. There will also be a set of short answer questions that accompany Part I, to be submitted in `writtenAnswers_p1.pdf`. Part II asks you to implement another game agent as well as an improved evaluation function. This document details the requirements for Part I.

Total Marks: Part I has a total of 40 marks. Part II – not specified here – has a total of 30 marks. This assignment, (Part I and Part II), has a total of 70 marks and represents 15% 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 A3 web page. Modify `multiAgents.py` appropriately so that it solves the problems specified in this document. Create a file `writtenAnswers_p1.pdf` containing your responses to the short answer questions. You will submit the following files:

- `multiAgents.py`
- `writtenAnswers_p1.pdf`
- `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 python2 (version 2.7) using only standard imports.* Your code will be tested using this version and you will receive zero marks if it does not run using this version. **Note that this version is different from the previous assignment in that it does not use python3.**
- *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 3 Clarification page:
http://www.teach.cs.toronto.edu/~csc384h/winter/Assignments/A3/a3_faq.html.
You are responsible for monitoring the Assignment 3 Clarification page.

Questions: Questions about the assignment should be posted to Piazza:
https://piazza.com/utoronto.ca/winter2018/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.

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

- 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.

    - **writtenAnswers_p1.pdf** - Where the requested written responses should be placed

- 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:

```
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 the flag `-q`, which suppresses 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`

- `-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 contains testcases for both Part I and Part II of this assignment. The autograder will run all tests. However, we will only be marking questions 1 to 3 for the tasks related to this document. (Questions 4 and 5 will follow in Part II.)

Question 1 : Reflex Agent (8 points + 3 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
```

Explorative Questions – to submit (3 points):

1. What is the difference in the definition of a heuristic value for a game state, and for a state in A* search? What properties make a heuristic in either situation 'good'? (3 points total)

Question 2: Minimax (10 points + 5 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, part II will clean up these issues.

Explorative questions – to submit (5 points):

1. Note that Pacman will have suicidal tendencies when playing in situations where death is imminent. Why do you think this is the case? Briefly explain in **one or two sentences**. (2 points total)

2. Consider an evaluation function that returns the square of a state's true minimax value. Consider three cases:

   (i) In planning a move (or strategy) for a certain state *S*, the agent does not hit any terminal state.

   (ii) In planning a move (or strategy) for a certain state *S*, the agent only hits a terminal state.

   (iii) In planning a move (or strategy) for a certain state *S*, the agent sometimes hits the terminal state.

   Answer the following questions:

   (a) Will the search result in the same strategy in cases (i) and (ii)? [Same/Not Same] (1 point)
   (b) Will the search result in the same strategy in cases (i) and (iii)? [Same/Not Same] (1 point)
   (c) Will the search result in the same strategy in cases (ii) and (iii)? [Same/Not Same] (1 point)

Question 3: Alpha-Beta Pruning (10 points + 4 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 some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

Explorative Questions – to submit (4 points total):

1. You should notice a speed-up compared to your `MinimaxAgent`. Consider a game tree constructed for our Pacman game, where *b* is the branching factor and where depth is greater than *d*. Say a minimax agent (without alpha-beta pruning) has time to explore all game states up to and including those at level *d*. At level *d*, this agent will return estimated minimax values from the evaluation function.

   (a) In the best case scenario, to what depth would alpha-beta be able to search in the same amount of time? (1 point total)

   (b) In the worst case scenario, to what depth would alpha-beta be able to search in the same amount of time? How might this compare with the minimax agent without alpha-beta pruning? (2 points total)

2. **True or False:** Consider a game tree where the root node is a max agent, and we perform a minimax search to terminals. Applying alpha-beta pruning to the same game tree may alter the minimax value of the root node. (1 point total)

HAVE FUN and GOOD LUCK!