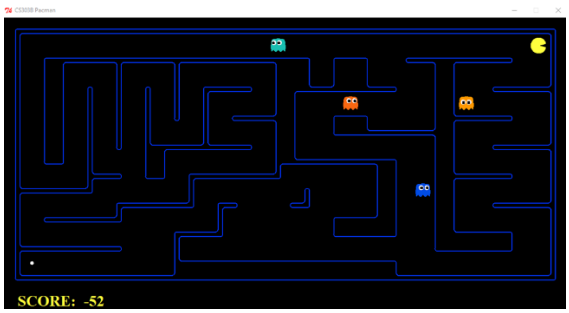
Assignment 2

Handling Ghosts, Minmax, Evaluation of States.



Introduction

This assignment will develop your knowledge of building a reflex agent that selects its next actions based on some heuristics, then we will implement minimax and expectimax algorithms. We will also develop a new evaluation function.

1. Reflex Agent (question 1)
2. Minimax (question 2)
3. Evaluation function (question 3)

Please install a new version of the pacman game to build your answers to this assignment (do not use the installation from assignment 1 because there are some new files).

Files you will edit in the assignment (you will edit and submit these files only):

- **`multiagents.py`** Where your answers will be

Files you should use and understand:

- **`pacman.py`** The main file that runs Pacman games.
- **game.py** The logic behind how the Pacman world works. Includes AgentState, Direction and Grid.
- **util.py** Useful data structures for implementing search algorithms

Files to support the program and game (you don't need to look at these files):

`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
`miltiagentTestClasses.py`	Project 2 specific autograding test classes

Important: the assignment will be graded partially using an automated script so please don't change any files **except the ones you submit.**

Autograder

You can use the autograder provided to test your work against some test cases that have already been implemented:

Type (at a command line):

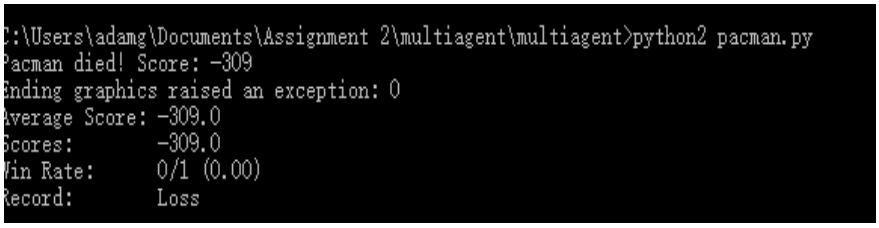
- `python2 autograder.py`

Please develop your programs in small parts and test it by the autograder or just running it (eg python2 pacman.py –layout tinyMaze –pacman YourAgent) often to avoid numerous errors at the same time which will be difficult to solve…

Getting Started

Download the code for the project in the zip archive multiagent.zip and extract it on your local computer.
Ensure you have installed and are running python 2 (not python 3).

1. Type > python2 pacman.py -h to get a list of commands that can be used in the game
2. **Play the game manually a few times to understand it (ghosts, power pills, moving etc):**
    a. Type > `python2 pacman.py`



3. Now test the example reflex agent
    a. Type > `python2 pacman.py -p ReflexAgent`
    b. Notice it doesn't play well even in simple layouts (observe -l command to change layouts):
    c. Type > `python2 pacman.py -p ReflexAgent -l testClassic`
    d. Read the code of this simple reflex agent (see multiagents.py) and ensure you understand it

The game works as follows:

- You can move the pacman agent North, East, South, West by the up, left, down, and right arrows.
- You receive 10 points for each pellet eaten,
- You loose 1 point for each second,
- You win if eat all pellets,
- If you eat a power pellet (large one) you make the ghosts scared for 10 seconds – can't eat you.

Question 1 Reflex Agent (4 marks)

This questions requires you to update the existing reflex agent we looked at previously to make it perform better. Identify the (possibly multiple) places you will update in the file multiAgents.py by comments provided.

- Improve the existing ReflexAgent in multiAgents.py to play better. The provided code contains examples of how to query information about the **GameState**.
- A good agent will consider food and ghost locations to perform well.
- Your new agent should be able to easily clear the **testClassic** layout (see getting started section earlier).
    - `Python2 pacman.py -p ReflexAgent -l testClassic`
- Try your new agent on the mediumClassic layout with 1 or 2 ghosts (you can turn animation off – see game options):
    - `python pacman.py --frameTime 0 -p ReflexAgent -k 1`
    - `python pacman.py --frameTime 0 -p ReflexAgent -k 2`
- How does your agent perform? Likely will die with 2 ghosts.

Hints:

Use the inverse of important values such as distance to food rather than raw values.

Command line Options: you can make the ghosts more smart using the option -g DirectionalGhost; you can use -f to run with a fixed random seed (same choices every game if the randomness prevents you evaluating your agents performance as you are developing); you can test multiple games in a row with -n; you can turn off graphics with -q to run many games quickly.

Grading: the autograder 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

- python2 autograder.py -q q1

To run it without graphics, use:

- python2 autograder.py -q q1 --no-graphics

Question 2 Minimax (5 marks)

This question requires you to write an adversarial search agent implementing minmax. Identify the places you will update in the file multiAgents.py by comments provided:

- It will be written in the class MinmaxAgent in the file multiagents.py
- Your algorithm will have to work with any number of ghosts so your minimax tree will have multiple min layers (one for each ghost) for every max layer
- Your code should expand the game tree to an arbitrary depth.
- Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction.
- MinimaxAgent extends MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction.
- Make sure your minimax code refers to these two variables where appropriate as these variables are populated in response to command line options. You can print out debugging statements to check which options you are using.
- **A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.**

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only way reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call GameState.generateSuccessor. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run:

- `python2 autograder.py -q q2`

This will show what your algorithm does on a number of small trees, as well as a pacman game.

To run it without graphics, use:

- `python2 autograder.py -q q2 --no-graphics`

Hints:

- 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.
- The evaluation function for the pacman test in this part is already written (self.evaluationFunction). You shouldn't change this function, 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.
- The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (> half of the time) despite the prediction of depth 4 minimax.

    - `python2 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4`

- 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 bad at winning. He'll often move around randomly without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior, the new evaluation function in the next question will clean up all of these issues.
- When Pacman believes that his death is unavoidable**, he will try to end the game as soon as possible because of the constant penalty for living**. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst. Make sure you understand why the agent rushes to the nearest ghost in this case:

    - `python2 pacman.py -p MinimaxAgent -l trappedClassic -a depth=3`

Question 3 Evaluation Function (6 marks)

This question requires you to write a new evaluation function in the provided place betterEvaluationFunction in multiAgents.py.

You may use any tools at your disposal for evaluation, including your search code from the last project. 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).

- `python2 autograder.py -q q3`

Grading: the autograder will run your agent on the smallClassic layout 10 times. It 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 EC2, so this machine will have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs).
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

Hints

- As for you did in the 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 and give different weights to them:
  - 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.