American University of Sharjah

School of Engineering
Department of Computer Engineering
P. O. Box 26666
Sharjah, UAE

Instructor: Omar Arif
Office: ESB-2178
Phone: 971-6-515 4821
e-mail: oarif@aus.edu
Semester: Fall 2022

CMP 333 Artificial Intelligence
Course Project 2
Fall 2022
Due Date: 27th November 2022

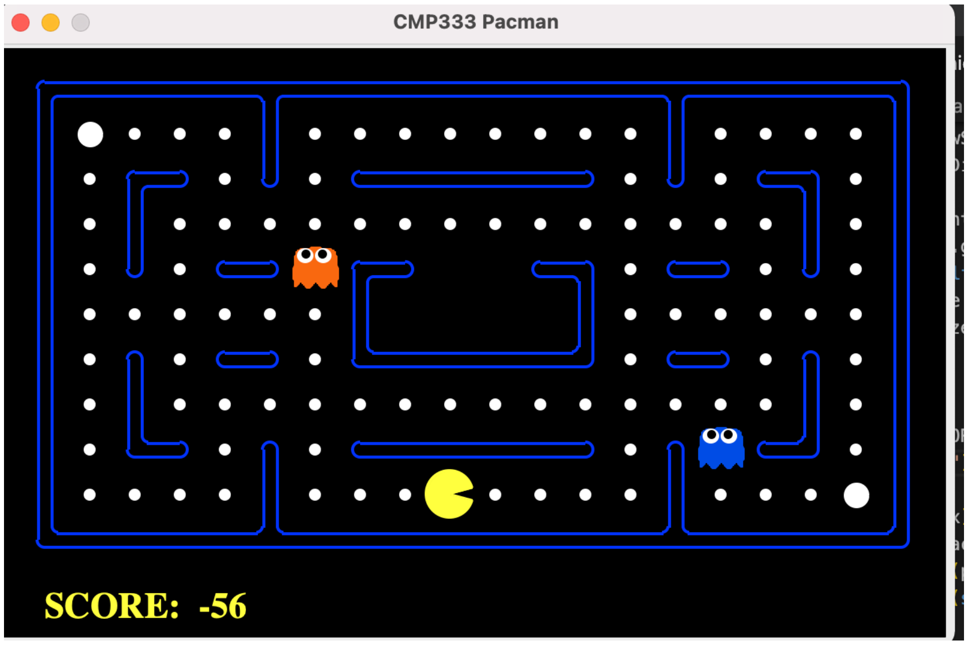

Table of Contents

- Introduction
- Welcome to Multi-Agent Pacman
- Minimax
- Expectimax

Introduction

In this project, you will design agents for the classic version of Pacman, including ghosts. You will implement both minimax and expectimax search.

The code base has not changed much from the previous project, but please start with a fresh copy of starter code provided on iLearn, rather than intermingling files from project 1.

Project adapted from University of California CS188 https://inst.eecs.berkeley.edu/~cs188/fa22/projects/proj2/ for educational purpose.

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

```
python autograder.py
```

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

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

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.

The code for this project contains the following files, available as zip file on iLearn.

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`	Project 2 specific autograding test classes

Files to Edit and Submit: You will fill in portions of multiAgents.py during the Project. Once you have completed the assignment, you will submit **only multiagent.py** on iLearn.

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.

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
```

A **simple reflex agent** is the simplest of all the agent programs. Decisions or actions are taken based on the current state. Note that reflex agent 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 (5 pts): Minimax

Now you will write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also 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 makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Important: 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 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

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

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:

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

Hints and Observations

- Implement the algorithm recursively using helper function(s).
- 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).
- 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 (665/1000 games for us) despite the dire prediction of depth 4 minimax.

```
  python 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.
- 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:

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

Question 2 (5 pts): Expectimax

Minimax is great, but it assumes that you are playing against an adversary who makes optimal decisions. 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.

ExpectimaxAgent will no longer take the min over all ghost actions, but the expectation according to your agent’s model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.

To see how the ExpectimaxAgent behaves in Pacman, run:

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

Compare with Minimax

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

You should find that your ExpectimaxAgent wins about half the time, while
your MinimaxAgent always loses.

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