CSCD84: Artificial Intelligence

Programming Assignment 2: Multi-Agent Search

- You will be asked to type and submit the following statement acknowledging that the code you submitted was written by you. Name the file as honorCodeAgreement.txt and submit it together with search.py and searchAgents.py to MarkUs.
  By turning in this assignment, I, [First and Last name], agree by the University of Toronto honor code and declare that all of this is my own work.
- Do not add any non-standard imports in the python files you submit (all imports already in the starter code must remain). All imports that are available on teach.cs are considered to be standard.
- Make certain that your code runs on mathlab.utsc.utoronto.ca using python3.10. You should all have an account on mathlab.utsc.utoronto.ca and you can log in, download your code and test it there before you submit.

Collaboration

All programming assignments should be done individually. All codes that you use in your submissions should be written by you. Do NOT simply copy-and-paste code from websites such as Stack Overflow, GitHub, or even AI generating agents. It is OK to see the sample code snippets on such places, but you should write the code yourself and give the credit to them by citing the source that you used (*e.g.*, by mentioning the page URL).

You should not share your code with others or see other student codes. It is OK to discuss the assignment or potential solutions for them with another student. However, in addition to writing the whole code yourself, you should also mention this (the discussion with other students in finding the answer to the assignments) in the submissions.

You can also ask your questions on the course discussion board. Please do NOT copy your code there and ask questions about it publicly. If there is a specific question related to part of your code that needs revealing part of your code, either ask it in a tutorial session or an office hour from a TA or via a private question on the discussion website.

Introduction

In this 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 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, or for one particular question (*e.g.*, q2), or for one particular test such as q2's test "0-small-tree" with the following commands:

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

The code for this project contains the following files, available as a zip file, named multi_agent.zip, under the "Files/Programming Assignments/PA2" section of the course page. The code for this project contains the following files

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 Pac-Man 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 Pac-Man
graphicsUtils.py	Support for Pac-Man graphics
textDisplay.py	ASCII graphics for Pac-Man
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pac-Man
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 3 specific autograding test classes

Table 1: Files Overview

Files to Edit and Submit: You will fill in portions of multiAgents.py during the assignment. Once you have completed the assignment, you will submit these files to MarkUs. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

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.

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.

Q1 (4 pts): 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 additional 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: conditions with

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

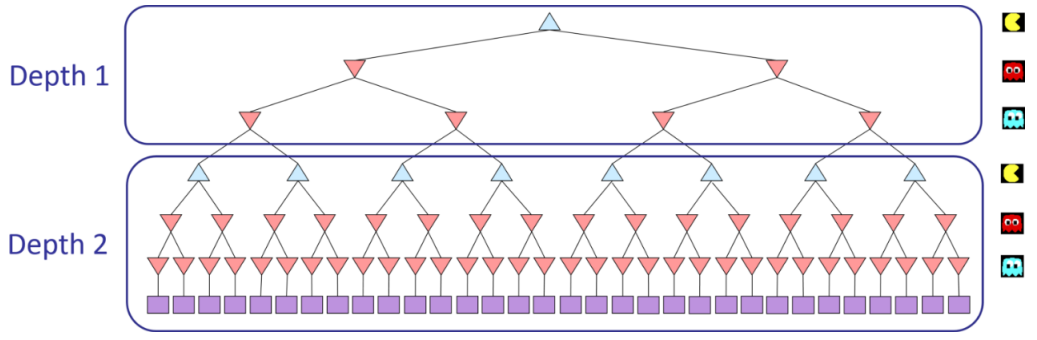
Don't spend too much time on this question, though, as the meat of the project lies ahead.

Q2 (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 (see diagram below).



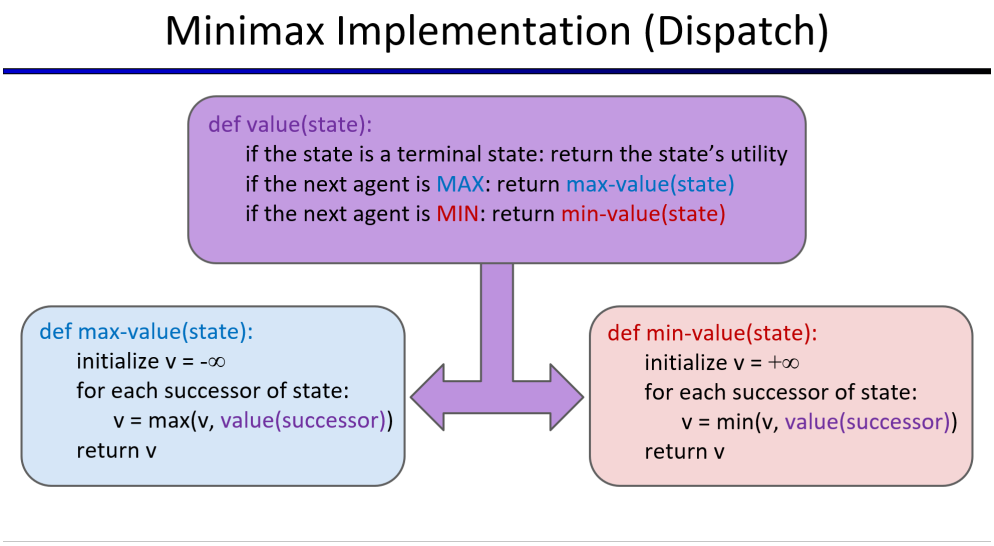
Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call GameState.generateSuccessor. 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 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:

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

The pseudo-code below may help you understand the algorithm you should implement for this problem.



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.
- 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 (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.
- 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. 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, question 5 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:

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

Make sure you understand why Pacman rushes the closest ghost in this case.

Q3 (5 pts): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in AlphaBetaAgent. Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on smallClassic should run in just a few seconds per move or faster.

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

The AlphaBetaAgent minimax values should be identical to the MinimaxAgent minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the minimaxClassic layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4 respectively.

Grading: Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.generateSuccessor more than necessary.

For this question, you should not prune on equality.

The pseudo-code below represents the algorithm you should implement for this question. To test and debug your code with or without graphics, run

```
python autograder.py -q q3
python autograder.py -q q3 --no-graphics
```

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

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.

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

Q4 (5 pts): Expectimax

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.

As with the search and problems yet to be 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.

The pseudo-code below may help you understand the algorithm you should implement for this problem.

Expectimax Pseudocode

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of

course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. 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
```

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:

```
python pacman.py -p AlphaBetaAgent -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 AlphaBetaAgent always loses. Make sure you understand why the behavior here differs from the minimax case.

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.

Q5 (6 pts): Evaluation Function

Write a better evaluation function for Pacman in the provided function betterEvaluationFunction. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. 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).
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, when run with --no-graphics.
- The additional points for average score and computation time will only be awarded if you win at least 5 times.
- Please do not copy any files from PA1, as it will not pass the autograder on Gradescope.

You can try your agent out under these conditions with or without graphics with the following commands.

```
python autograder.py -q q5
python autograder.py -q q5 --no-graphics
```

Submission

In order to submit your project upload the Python files you edit (*i.e,* multiAgents.py), together with the honor code agreement to MarkUs. Do not make any changes to the other files in the directory. The honor code agreement file should be named as "honorCodeAgreement.txt" and its content must be the following sentence with your first and last name.
"By turning in this assignment, I, [First and Last name], agree by the University of Toronto honor code and declare that all of this is my own work."