CS-331: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

ASSIGNMENT 3: Adversarial Search

Instructor: Mian Muhammad Awais

TAs: Habiba Farrukh, Aiza Anjum Chaudhry, Fatima Tariq, Hamza Mahmood and Maryam Khalid

Submission Deadline: Monday, October 19, 2015:

As before, this assignment can be done in groups of 2 (same groups as last time). The evaluation of the assignment will be on individual basis. Members of the same group will get marks depending on the working of tasks they have implemented and their performance in viva. So make sure you contribute equally in the assignment.

All the submitted codes will be tested against each other and the solutions present online for plagiarism and in case of slightest doubt the concerned case will be forwarded to the Disciplinary Committee.

As always please reach out to the course staff for any help regarding the assignment and its configuration.

Adversarial Search

Introduction

In this assignment you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search. Download the zipped folder multiagent.zip which contains all the necessary files to complete this assignment.

We are introducing a new ‘pacman’ framework for this assignment. You will be required to edit specific portions of the following file during your implementation:

multiAgents.py Where all of your multiagent search agents will reside. Files you might want to look at:

pacman.py The main file that runs Pacman games. This file describes a Pacman GameState type, which you 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 can ignore rest of the files.

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.

Warm Up:

First, play a game of classic Pacman:

>> python pacman.py

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.

Task 1 (20 points):

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!

Hint: When writing your evaluation function, 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 assignment, you'll be evaluating states.

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. 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 evaluate your agent with the autograder using these commands:-

>> ***python autograder.py -q q1***

To run it without graphics, use:

>> ***python autograder.py -q q1 --nographics***

Don't spend too much time on this question as it's just meant to get you acquainted with the framework.

Task 2 (30 points):

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 general for any number of agents. 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 a fixed depth, which will be specified at the command line. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction. MinimaxAgent extends MultiAgentAgent, 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 Pac-Man move and all the ghosts' responses, so depth 2 search will involve Pac-Man and each ghost moving two times.

Hints and Observations

• The evaluation function 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. Lookahead 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***

• To increase the search depth achievable by your agent, remove the Directions.STOP action from PacMan's list of possible actions. Depth 2 should be pretty quick, but depth 3 or 4 will be slow.
• 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 assignment, 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.

• 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.

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.getLegalActions. 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 --nographics

Don’t take the score output by the autograder too seriously. We will calculate the number of test cases passed by your agent, and your final score will mostly depend on the number of test cases successfully passed by your agent.

Task 3 (30 points):

Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. Fill in **ExpectimaxAgent**, where your agent 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 RandomGhost ghosts, which choose amongst their getLegalActions uniformly at random.

Note: Make sure when you compute your averages that you use floats. Integer division in Python truncates, so that 1/2 = 0, unlike the case with floats where 1.0/2.0 = 0.5.

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 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. Make sure you understand why the behavior here differs from the minimax case.

Grading: To expedite your own development, we've supplied some test cases based on generic trees. You can debug your implementation on small 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.

Task 4 (20 points):

For the last task of the assignment you are required to implement a variant of **Expectiminimax** search. While minimax assumes that the agents are completely rational and expectimax search assumes that the agents are random, this strategy is not suitable in a desired case where one of the ghosts is rational while the other is random. Fill in ExpectiminimaxAgent, where your

Pacman agent makes its decision based upon the assumption that one of the ghosts is rational while the other is random.

Note: You can assume that the layout contains exactly two ghosts and test your agent on any provided layout which contains two ghosts. For more details about each layout, you can access the layouts present in the layouts folder using any text editor.

Use -g DirectionalGhost to switch from playing against default random ghosts to slightly smarter directional ghosts. What difference do you observe?

The assignment will require more effort and reading of the source files as compared to the previous assignment but the good news is that it requires relatively less effort overall. So make sure you start early to get it finished by the deadline.

Good luck! :)