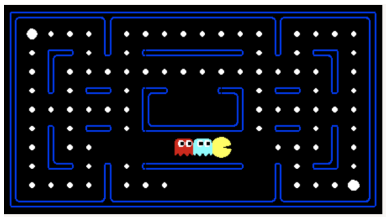
CSCI 360 Project #2: Multi-Agent Search

Released: February 5, 2021

Due: February 19, 2021



Contents

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. This project also has a written part that includes a short report to describe the algorithms you have implemented and a Markov Decision Processes (MDP) question to help you prepare for the next project.

Getting Started: 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. You can also reuse the environment setup in project 1.

Download *multiagent.zip*, unzip it, and change to the root directory. Like in project 1, we also provides an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

If you want to run for one particular question, such as q2, by:

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

If you want to run for a particular test case, here's an example

```
python autograder.py -t test_cases/q2/0-small-tree
```

Files you will 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
- *submit.py:* file that generate submit token. Please **do not change** this file !

Question 1: Minimax (5 points)

In this question, 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.

The actual ghosts operating in the environment may act partially randomly, but Pacman's minimax algorithm assumes the worst:

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

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

Important note: A single level of the search is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving twice.

To test and debug your code (and also we will grade based on these test cases), run

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

The autograder will be very picky about how many times you call GameState.getNextState. If you call it any more or less than necessary, the autograder will complain. If you are sure that your implementation is correct and you want to run the test cases without graphics, use no-graphics flags as it will save you a lot of time.

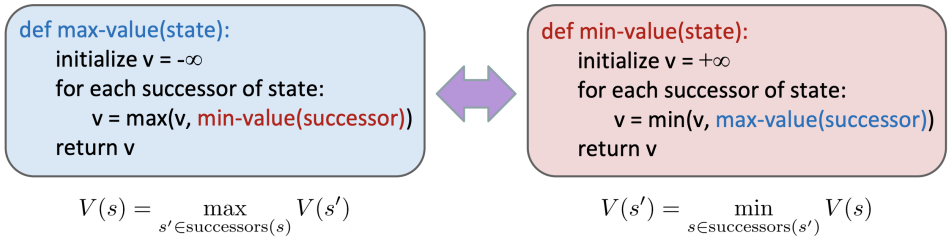
Hint:

- Recursion is extremely useful in implementing the trees !
- Even though your algorithm is correct, Pacman might lose during the game. Don't worry too much about it. Correct implementation will pass the tests.
- Pacman is always agent 0, and the agents move in order of increasing agent index.
- 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*

Hint: Here are some method calls that might be useful

- *gameState.getNextState* will return the next game state after an agent takes an action
- *gameState.getNumGhost* will return the number of ghost in the game (Note in the game you might have more than one ghost and the number does not include Pacman itself!)
- *gameState.getLegalActions* will return a list of legal actions for an agent. Please generate child states in the order returned by *GameState.getLegalActions*

The pseudo-code below represents the algorithm you should implement for this question.



Question 2: Alpha-Beta Pruning (5 points)

In this question, you will need implement Alpha-Beta Pruning algorithm in the *AlphaBetaAgent* function in *multiAgents.py*. You can implement Alpha-Beta Pruning algorithm based on Minmax search, but more efficient than Minmax. Similarly, your algorithm should be extend to multiple agents.

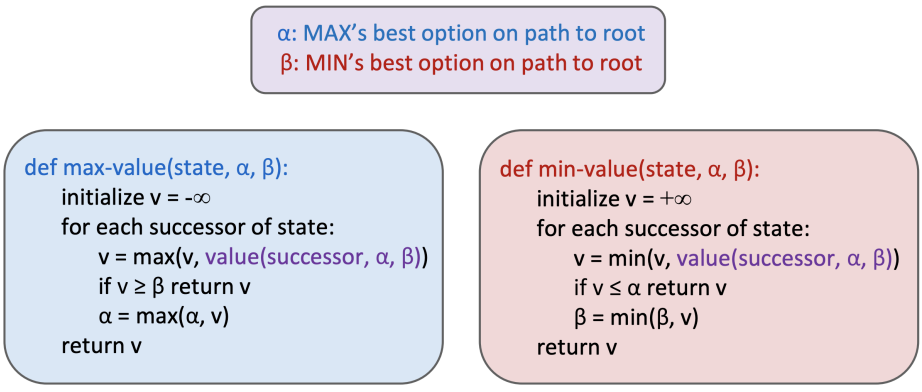
To test your code, you can run

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

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
```

You must not prune on equality in order to match the set of states explored by our autograder. The pseudo-code below represents the algorithm you should implement for this question.



Question 3: Expectimax(5 points)

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.

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.

Question 4: Evaluation Function(6 points)

Write a better evaluation function for pacman in the provided function betterEvaluationFunction. The evaluation function should evaluate states, rather than actions. 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).

You can test your evalution function with the following command.The autograder will run your agent on the smallClassic layout 10 times, therefore it may take a few seconds even you run without graphics.

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

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 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. Please do not copy any files from Project 1, as it will not pass the autograder.

Question 5 : Report and Written questions (3 points)

A. Report for adversial search algorithm (1 points)

In the first part, you need to write a report about the algorithm you implemented

1. Test the different agent 10 times with smallClassic maze with the following command line.

```
python pacman.py -p MinimaxAgent -l smallClassic -a depth=2 -n 10 --frameTime 0
python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=2 -n 10 --frameTime 0
python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=2 -n 10 --frameTime 0
```

   Give a screenshot of the winning performance of each agent. Do you think they perform well in smallClassic maze? Please give your reason for that (We will accept any reasonable explanation).
2. Give a screenshot of your better evaluation function in Question 4 . Please explain what you did in the function and why you design in this way.

B. Written question of MDP (2 points)

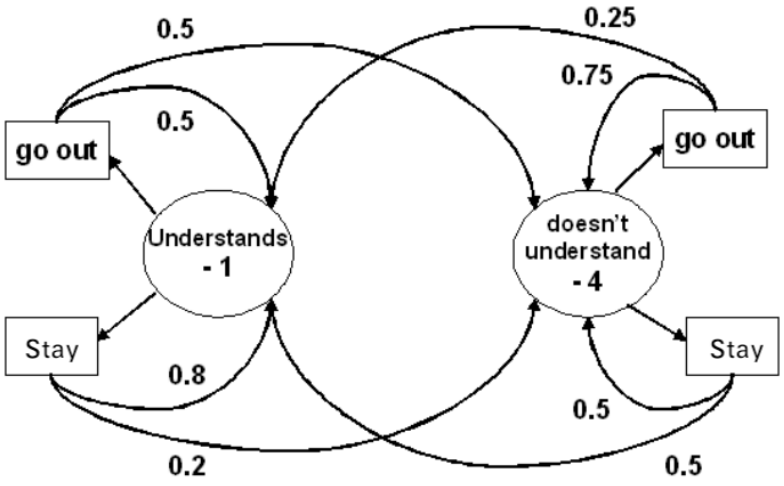
Mike has registered CSCI 360 this semester. During every weekend, he has two options: either he can go out with his friends , or he can stay at home and review the lecture videos. Suppose Mike has mastered the lectures well in class the week before. If he chooses not to go out, he has 80% possibility to understand next week's lectures. If he chooses to go out with his friends, the chance that he understand the next week's lectures drops to 50%. However, if Mike did not understand lectures in last week (As you know, CSCI 360 is not a easy class !), the chance that he understand the next week's lectures drops to 25% when he decides to go out with his friends. If he chooses to stay at home, he has 50% possibility to understand next week's lectures.

Like everyone else in class, Mike really likes CSCI 360. If he doesn't understand the lecture on a week, he loses 4 happiness points. On the other hand, if he understand the lectures, because he has to spend time reviewing notes, he will lose 1 happiness point.

1. Draw Mike's choices as a MDP diagram. The diagram should includes actions, stats and transitional probabilities.
2. **Additionally**, if Mike does not go out with his friends, he will lose 2 happiness points for that. What policy can Mike take? List all the policies.

3. Suppose at the beginning of the semester, Mike has 50% possibility to understand the first week's lectures. What is the cost value then for every policy, for three weeks ? What is the optimal policy ? Define discount factor $\gamma = 0.9$, per week . Please show your calculations.

Q1



Q2

policy	state	cost	u	d
a: go out either way	u	-1	0.5	0.5
	d	-4	0.25	0.75
b: go to sleep either way	u	-3	0.8	0.2
	d	-6	0.5	0.5
c: go out if understands	u	-1	0.5	0.5
	d	-6	0.5	0.5
d: go out if doesn't undertand	u	-3	0.8	0.2
	d	-4	0.25	0.75

Q3

policy a:

	t	**p(u)**	**p(d)**	**total cost**
Starting from u:	t_0	1	0	-1
	t_1	0.5	0.5	$0.9(-1\times0.5-4\times0.5)$
	t_2	0.25+0.125	0.25+0.375	$0.9^2(-1\times0.375-4\times0.625)$
and, from d:	t_0	0	1	-4
	t_1	0.25	0.75	$0.9(-1\times0.25-4\times0.75)$
	t_2	0.125+0.1875	0.125+0.5625	$0.9^2(-1\times0.3125-4\times0.685)$

Results for Assumption 1:

Total cost for policy a: $Cost(a, t = 3) = 0.5 \times -5.5788 + 0.5 \times -9.4056 = -7.492$

Similarly,

$Cost(b, t = 3) = -11.316$

$Cost(c, t = 3) = -9.485$

$Cost(d, t = 3) = -9.431$

Results for Assumption 2:

$Cost(a, t = 3) = -7.492$

$Cost(b, t = 3) = -9.664$

$Cost(c, t = 3) = -8.675$

$Cost(d, t = 3) = -8.558$

Results for Assumption 3:

$Cost(a, t = 3) = -7.492$

$Cost(b, t = 3) = -9.316$

$Cost(c, t = 3) = -8.485$

$Cost(d, t = 3) = -8.364$

The optimal policy is then, policy a - to go out either way (for all assumptions)

Submission (Important !)

Once you're happy with your score that the *autograder* gives you, run *python submission_autograder.py* to generate a unique *submit.token*. Again, **PLEASE ZIP** this token along with *report.pdf* and *multiagent.py* into one zip file. Name this zipfile as **Last_name_First_name_hw2.zip**. And then submit to Blackboard! You can submit multiple times and we will grade only the last submission We will deduct 3 points for incorrect submission .


I AM ONCE AGAIN ASKING...
PLEASE ZIP YOUR SUBMISSION FILES!
imgflip.com