ARTIFICIAL INTELLIGENCE (E016330)
ALEKSANDRA PIZURICA
GHENT UNIVERSITY
2022/2023 AY

Assistants: Xianlu Li, Nicolas Nadisic,
Ide Van den Borre
Professor: Aleksandra Pizurica

Practical Session 1: Multi-agent search

1 Introduction

The goal of this project is to better understand and learn how to implement multi-agent search strategies. We use the familiar game environment – the Pacman game. The project is based on the material from the Berkeley course CS188 in Artificial Intelligence[1].

The code for this project consists of several Python files, which are available on Ufora. Please download the file `multiagent.zip` in the section "Project 1" on Ufora, and do not reuse the code from previous assignments as we made some changes.

You will need to modify the file `multiAgents.py`. You may want to have a look at `pacman.py` and `game.py` to understand how the game works, and `util.py` for functions and structures that may be useful. However, you should not modify them. The other files serve only as supporting files and you should not need to look at them.

This project is done in groups of 2 students (with the exceptions we accepted). Each group will have to submit their code and a small report, see Section 3.

2 Assignment

Welcome to Multi-Agent Pacman

To make sure your setup is working, play a game of Pacman by running the following command:

```
python pacman.py
```

Then, have a look at a basic reflex agent playing Pacman.

```
python pacman.py -p ReflexAgent
```

You can try it on different layouts (see the folder `/layouts` for more options).

```
python pacman.py -p ReflexAgent -l testClassic
python pacman.py -p ReflexAgent -l originalClassic
```

As you can see, this is not the smartest agent. Inspect its code (function `ReflexAgent` in `multiAgents.py`) and try to understand its strategy.

[1] https://inst.eecs.berkeley.edu/~cs188/fa22/projects/

Question 1 *Explain in your own words the strategy of this reflex agent. What are the limitations of the reflex agent? Write this compactly, in one paragraph.*

Question 2 *Propose a simple modification of the code that would help this reflex agent to lose less often. Which function should be modified? Explain your the proposed improvement in a couple of sentences (you do not need to actually code it).*

You may see that, when Pacman is trapped between two ghosts, he tries to die as fast as possible.

```
python pacman.py -p ReflexAgent -l trappedClassic
```

Question 3 *Explain in a few sentences why the reflex agent tries to die fast when it is trapped.*

Task 1: Minimax

The **minimax** algorithm consists in, for a given game, minimizing the possible loss in the worst case (maximum-loss) scenario. This strategy is equivalent to maximizing the minimum possible gain. In the context of Pacman, we will thus try to maximize the minimum score for Pacman among the possible moves. For this task, you will write your code in the class `MinimaxAgent`.

Two-agent case: First, you need to implement the minimax algorithm as seen in the lecture to play Pacman with only one ghost (Pacman + one ghost = two agents). You will find the pseudo-code in the slides of the lecture. To play your MinimaxAgent with only one ghost, add the `-k 1` option:

```
python pacman.py -p MinimaxAgent -k 1
```

Question 4 *Describe and explain in a couple of sentences the strategy of this agent with only one ghost.*

Generalization to multiple agents: Now you need to generalize your code to manage multiple ghosts. Hint: your minimax tree must have several min layers (one for each ghost) for every max layer.

General remarks: Your code should be able to expand the minimax tree to any given depth. Note that one search layer corresponds to one move of every agent (both Pacman and ghosts). Therefore, depth-2 search corresponds to Pacman and each ghost moving twice. Pacman is always agent 0, he moves first and then the ghosts move in the order of increasing index.

To score the leaves of the minimax tree, you may use the function `self.evaluationFunction`, which by default calls `scoreEvaluationFunction`. The class `MinimaxAgent` is a subclass of `MultiAgentSearchAgent`, therefore you have access to the variables `self.depth` and `self.evaluationFunction`. You must use these variables in your code, because their values represent parameters set by the user (they are defined by command line options). You should not need to change `self.evaluationFunction`.

Grading: To check the correctness of your code, we use `autograder.py`, that is also provided to you. This autograder script counts the number of game states explored by your agent, which is in practice the number of times you call the function `GameState.generateSuccessor`, so make sure to call it only the number of times necessary. To use the autograder to test your minimax solution, use the following command:

```
python autograder.py -q t1
```

To run the autograder without graphics, use:

```
python autograder.py -q t1 --no-graphics
```

A correct minimax agent should score 5/5. Note that the minimax agent will sometimes get stuck or lose, this is a normal behavior.

Question 5 *Both the reflex agent and the minimax agent use the function* `self.evaluationFunction`. *What are the differences in the way how they use this function? Explain in one paragraph.*

Question 6 *You should see that the minimax agent is good at avoiding ghosts, but not good at winning. Explain this behavior, and propose a way to improve it. Describe this all compactly, in a few sentences.*

Task 2: Alpha-beta pruning

Now let us explore the minimax tree more efficiently by alpha-beta pruning in `AlphaBetaAgent`. Alpha-beta pruning will reduce the branching of the search tree, spend search time on "more promising" subtrees, and increase search depth. You can find the pseudocode in the slide show of the theory class. But your algorithm will be more general than the pseudocode, since you need to extend the logic to an arbitrary number of minimization agents.

Type the command below, you will see a speed-up. Ideally, depth 3 on `smallClassic` should run in just few seconds per move or even faster.

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

The `AlphaBetaAgent` should get the same result as the `MinimaxAgent`, although the actions it selects can vary because of different tie-breaking behavior.

Attention! Because we will check if your code explores the correct number of states, it is important that you perform alpha-beta pruning *without reordering children*. In other words, you can't change successor states order returned by `GameState.getLegalActions`.

You must not prune on equality in order to match the set of states explored by our autograder.

As for the previous task, we will use the autograder to check your code. You can test and debug your code by the command below:

```
python autograder.py -q t2
```

It will show how your code works in the game. To run it without graphics, use:

```
python autograder.py -q t2 --no-graphics
```

Question 7 *Explain in one paragraph how the alpha-beta pruning makes exploring more efficient, and why the result is identical to the minimax agent?*

Task 3: Expectimax

The two previous implemented algorithms, minimax and alpha-beta pruning, rely on the assumption that the opponent always chooses the optimal decision. However, in reality this is certainly not always the case. In this task you will implement the `ExpectimaxAgent`. This agent can deal with uncertainty of agents who make suboptimal decisions. As for the previous tasks, you can use the autograder to debug your code:

```
python autograder.py -q t3
```

The `ExpectimaxAgent` will calculate the expectation according to the agent's model of how the ghosts act. To simplify your code, you can assume that you will be playing against an opponent that chooses uniformly at random `getLegalActions`.

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 will at least try. Investigate the results of these two scenarios by running them a few times:

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

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

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

Question 8 *Explain in a few sentences why you observe this more cavalier and more successful behavior of the* `ExpectimaxAgent`*.*

3 Submission

You need to submit 2 files:

- Your final version of the file `multiAgents.py`.
- A report in PDF containing the answers to our questions. Please answer the questions separately, indicating the questions numbers.

A few remarks:

- Deadline: **23 November 23:59 CET**
- Submit on the Ufora course page, Ufora-tools → Assignments → Project 1 - Multi-agent
- These files must be submitted as one zip file named after both authors. For example, for the group of Xianlu Li and Ide Van den Borre, name the file `multiagent_xli_ivandenborre.zip`

- The evaluation of the code is based on its execution. Make sure your code works with the original Pacman framework provided and with the autograder.
- Write clear and concise answers, using the right terminology (as in the theory course).
- Plagiarism will be very heavily penalized. We use automatic tools to check for plagiarism, and they are quite hard to fool. Helping each other is a good thing, but do not share your code or answers.