Prof. Dr. Eirini Ntoutsi
M.Sc. Philip Naumann & M.Sc. Arjun Roy
AG Künstliche Intelligenz und Maschinelles Lernen

21.06.2021

Artificial Intelligence
Project 2

Summer Semester 2021

Organization

- In order to pass, you need to complete the questions Q1-Q3.
- Some questions also have additional sub-parts requiring intuitive *written* answers **IQ:** ***highlighted by color light blue.*** You need to write these answers in a **separate document**.
- You are not allowed to use additional libraries (apart from what is provided in the existing files). In particular, don't use an existing implementation and just call it. You are expected to implement the algorithms on your own. Moreover, plagiarism is of course not allowed!
- You may participate in this project as a group of 2 persons.
- Submit your solution no later than **19.07.2021, 23:59**.
- You can make re-submissions until the deadline, but the latest received submission will be graded only.
- Submit your solution (only the `multiAgents.py` file) to ai-sose21@lists.fu-berlin.de using the subject `"MatriculationNr_YourNames_AI21_Project2"`. **Please be sure to provide your matriculation numbers in the email.**

If you encounter any bugs, please report them to: philip.naumann@fu-berlin.de

Disclaimer

We are reusing a project from UC Berkeley[1]. The whole documentation can be found at `https://inst.eecs.berkeley.edu/~cs188/fa20/project2/`. The required code is available at `https://inst.eecs.berkeley.edu/~cs188/fa20/assets/files/multiagent.zip`.

[1] `http://ai.berkeley.edu`

Project Structure & Autograding

To run the project you need a **Python 3.6** environment.[2] You can check you progress by running the following command:

```
python autograder.py
```

If you encounter problems doing this, it could possibly be that your operating system uses Python 2.X instead of 3.X. In this case make sure to try out the command `python3`, i.e.:

```
python3 autograder.py
```

Files you will edit: `multiAgents.py`.
Files you might want to look at: `pacman.py`, `game.py` and `util.py`.

Fix the bug in grading.py

Running the `autograder.py` may result in the following error for you:

```
AttributeError: module 'cgi' has no attribute 'escape'
```

To fix this, make sure to replace `import cgi` with `import html` in `grading.py` and change `cgi.escape` to `html.escape` in the same file.

Introduction

In the last project, you had to design (un-)informed agents that are able to find paths in a Pacman game environment. In this case, you didn't need to pay attention to potential adversaries (aka *ghosts*). We will continue with this familiar environment with the addition of *adversaries* now. More specifically, you will design agents for the classic version of Pacman, *including ghosts*, that actively try to defeat you and make it harder to collect food. This means, the tasks are about the design of adversarial agents that can handle *multiple* opponents and not about simply finding shortest paths anymore. A particular difficulty and challenge for you will thus be the handling of more than one adversary, so you need to think about how you will implement this. 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 the last project, there are powerups which make Pacman immune to ghosts (and scare them; making them run away from Pacman) for some time. However, as last time, you don't need to consider these in this project.
As in project 1, this project includes an autograder for you to grade your answers on your machine. Play a game of Pacman by running the following command and using the arrow keys to move:

```
python pacman.py
```

[2] If you have difficulties regarding the technical requirements, please refer to https://inst.eecs.berkeley.edu/~cs188/fa20/project0/.

Question 1: 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 (Pacman). Think of it as a path like this:

$$\underbrace{\text{max} \rightarrow \text{min} \rightarrow \text{min} \rightarrow \text{min}}_{\text{Ply 1}} \rightarrow \underbrace{\text{max} \rightarrow \text{min} \rightarrow \ldots}_{\text{Ply 2}} \rightarrow \underbrace{\text{max} \rightarrow \text{min} \rightarrow \ldots}_{\text{Ply 3}}$$

in case of 3 ghosts. So here you would always pick the min of the three mins and pass it to max again. Or in other words, pick the minimum value of all ghost in the current step (i.e. iterate through all agents): $\min\{\text{Ghost}_1, \text{Ghost}_2, \text{Ghost}_3\}$. This will ensure that your action depends on the most "dangerous" ghost. Of course you need to consider all available actions of each agent still and furthermore the order you traverse the ghosts matters (keep the order that `GameState` already provides for you).
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.
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.
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
```

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
```

IQ: ***Why does Pacman rush to the closest ghost in this case?***

Question 2: 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.
You must not prune on equality in order to match the set of states explored by our autograder. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)
To test and debug your code, run

```
python autograder.py -q q3
```

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 q3 --no-graphics
```

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.

IQ: ***Compare this solution with the previous minimax and comment on that.***

Question 3: 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 constraint satisfaction problems covered so far 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.
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. **IQ:** ***Investigate the results of these two scenarios and comment on them:***

```
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. **IQ:** ***Make sure you understand why the behavior here differs from the minimax case and comment on it.***
The correct implementation of expectimax will lead to Pacman losing some tests. This is not a problem: as it is correct behaviour, it will pass the tests.