Koç University
COMP341
Introduction to Artificial Intelligence
Assignment 2

Instructor: Barış Akgün
Due Date: November 25 2020, 23:59
Submission Through: Blackboard
Make sure you read and understand every part of this document

This programming assignment will test your knowledge and your implementation abilities of what you have learned in the adversarial search part of the class. You are asked to complete a coding part and answer a few questions about how it runs. The coding part of the homework will follow the Berkeley CS188 Spring 2020 pacman project P2: Multi-agent Search at `https://inst.eecs.berkeley.edu/~cs188/sp20/project2/`. The questions for the report part are given in this document.

This homework must be completed individually. Discussion about algorithms, algorithm properties, code structure, and Python is allowed but group work is not. Coming up with the same approach and talking about the ways of implementation leads to very similar code which is treated as plagiarism! Furthermore, do not discuss the answers directly as it will lead to similar sentences which is treated as plagiarism. Any academic dishonesty, will not be tolerated. **By submitting your assignment, you agree to abide by the Koç University codes of conduct.**

You may find yourself having trouble implementing the coding part. In this case, we are going to let you use someone else's code to answer the given questions, as long as you credit the person or the website you take the code from. If you chose this option, we are only going to grade your report.

Grading

You are given two options about submitting your homework: (1) both code and report, and (2) only report. The second option is given to you in case you are not able to implement the programming part. These options will be graded differently:

- **Code and Report:** The code part and the report will have 2:1 weight ratio in your submission (programming 2/3, report 1/3).
- **Only Report:** The report part will be treated as 1/2 of the total grade. You **must** credit the code you used and **must not** submit a code part.

The solution code for the homeworks can be found online. We are going to compare your submission with these sources. We are also going to compare your code to previous submissions of Koç students. If your code's similarity level is above a certain threshold, your code will be scrutinized. If we see any plagiarism, you will lose points in the best case and disciplinary action will be taken against you in the worst.

Part 1: Programming

You are going to do the 5 programming questions about adversarial search given in the website. You are only required to change *multiAgents.py*. If you have any issues with other parts of the code let your instructor or TA know ASAP, even if you manage to solve your problem. Use the data structures in *util.py* for the autograder to work properly. If you think you have the right answer but the autograder is not giving you any points, try to run it on individual questions (look at P0 for details on how to use the *autograder.py*).

Hints

There are hints for the programming part, especially about what features to use in your evaluation functions, in the project description website. Read those carefully, as they generally provide good ideas. We have also discussed example features during the class. The first things to try are the distance to ghosts, distance to food, distance to capsules, scared timer etc. More information is given below. Note that your evaluation function for question 1 is a good starting point for question 5. As noted in the website, do not waste too much time on both of these questions and only get back to them if you have time left.

Evaluation Functions

The `evaluationFunction` method of the `ReflexAgent` class already shows you how to get important information from the `GameState` class. As an addition, you can use the `getCapsules` method to get the locations of the remaining capsules as a list. These capsules let the pacman eat the ghosts for significant additional points!

You can call the `asList` method of the `Food` class to get the locations of the foods as a list. You can use the built-in `len` function to get the remaining number of foods or capsules. The `scaredTimes` gives you the remaining time for pacman to eat the ghosts. A 0 value means that the ghosts are not scared so watch out!

You are potentially going to use the distances to the food as a feature (e.g. the distance to the closest food). Your agent, pacman, might get right next to the food and not eat it! This is due to the fact that your evaluation function might return a lower value after eating the food and the next distance to the closest food or the average distance to the foods becomes higher. Make sure your evalation function takes this into account. The same thing might happen with eating the ghosts, pacman might chase the ghost but never actually eat it.

In the previous homework, HW1, you had the option to use the `mazeDistance`, which gives you the cost to go between 2 locations in the maze taking the walls into account, i.e., the true graph cost. This results in better cost estimates at the expense of more computation. This resulted in lower number of node expansions but potentially made the algorithms run slower than if you used the `manhattanDistance`. Feel free to use either, however, make sure that your code does not take too long!

Your pacman can get to a state of thrashing, which means that it either stops or keeps moving between two states, until a ghost comes nearby. In your `ReflexAgent`, you might think of penalizing the *Stop* action. This is not the best idea and there are other things you can do. If you happen to have this problem, try to tune your feature weights. Finally, it is okay if you do not have the best agent behavior as we are not sending a rover to Mars, yet.

Programming for Multiple Ghosts

Remember that in the class, we talked about the multiplayer case, where all the agents are trying to maximize their utility. Think of all the ghosts as trying to minimize your evaluation function, i.e., all the ghosts are *MIN* agents. This means that you need to call the *min-value* function for all of the ghosts. You can do it by passing an agent index to your value function. Note that pacman will always be agent 0. Look at the comments in the *multiAgents.py* file to see which functions take an agent index (look for `agentIndex`) as a variable. You are free to define your own functions, either externally or within the classes, but make sure the entry points are in the existing class methods (e.g. the `getAction` methods).

Keeping Track of Depth

In this assignment, 1-level of depth is interpreted as including all the agents' actions. In the 2-player version, this means 1-level of depth is completed after you check the actions of both the *MAX* agent and the *MIN* agent. If you have more than 1 ghost, the depth increases after you look at the actions of pacman and all the ghosts.

In the class, we have discussed why we are not able to search as deep as the end of the game. The assignment requires you to implement a depth limit. In a recursive algorithm, it might not be obvious how to implement this. There are multiple ways to do this. You can pass down a depth variable, along with your state to your functions.

Alpha-Beta Pruning

The α and the β values are not for the entire search tree but for a certain node. This means that you should not keep the same variables for the entire search.

Pacman Not Eating the Last Food

Due to the setup of the environment, you might not be able to force the pacman to eat the last food for the programming question 5. This is because the action sequences *Stop* and *Move*, where the latter is moving in any legal direction, and *Move* and *Stop* will potentially have the same value at the end of the game depending on the location of the ghosts. Since the *Stop* always comes before the *Move* actions when getting the legal actions, your agent might always chose it first. Keep this in mind if your pacman stops right next to a food at the end of a game. A way to solve this is to force pacman to chose the *Move* action in case two actions have the same value.

Part 2: Report

This part includes answering the following questions based on your program's output on the given pacman tests. You are expected to answer the questions concisely. Five sentences is more than enough for most of them. Limit yourself to 200 words per question. It is okay if you over-generalize, as long as your direction is clear and correct. Note that some answers are already in the provided link to the Berkeley site. For some questions, you do not even need to write that much (*Hint:* Q2, Q3 and Q4).

Create a PDF file named *report.pdf* containing your answers for submission. **Write your name and your ID on the report as well!**

Written Q1:
What are the features you used in your evaluation function for your reflex agent? Why did you chose them? If you have too many, limit yourself to at most 5. Do you think using the reciprocals or negatives of some values is a good idea and why?

Written Q2:
Try the following lines of code:

```
python pacman.py -p MinimaxAgent -l trickyClassic -a depth=2 -f
python pacman.py -p AlphaBetaAgent -l trickyClassic -a depth=2 -f
```

Run both them until 20 seconds (or less if pacman ends up dieing) and see how far the pacman has got. You do not need to write additional time keeping code, a stopwatch should suffice.

In your tests, were you able to see any speed difference between the `MinimaxAgent` and `AlphaBetaAgent`, between pacman actions? If so, why and if not why not? Is there any situation you came across that highlights this?

Written Q3:
When you were running the tests in the previous question, did your pacman behave exactly in both cases? Why?

Written Q4:
Now try the same with the ExpectimaxAgent;

```
python pacman.py -p ExpectimaxAgent -l trickyClassic -a depth=2 -f
```

Comment on how fast your code runs. Compare it with the `MinimaxAgent` and `AlphaBetaAgent`. Note that this comparison is trickier to do. If you are not able to conclusively see anything, write what you would have expected.

Written Q5:
We are sure that you were able to write a better evaluation function than the one we used for the programming questions 2-4. Did you change anything from your evaluation function for the ReflexAgent? If so, what were the changes? What, if anything, is different in this case? If you have written something entirely different, comment on your new evaluation function.

Written Q6:
For both the programming questions 1 and 5, you probably needed to tune your feature weights. If so,

comment on how you selected your weights, what did you prioritize and why.

Submission

You are going to submit a compressed archive through the blackboard site. This archive should only contain *report.pdf* and *multiAgents.py*. Make sure **you put your name and ID inside the report** as well. Other files will be deleted and/or overwritten. Do not submit any code if you only want us to grade your report.

- **Important:** Download your submission to make sure it is not corrupted and it has your latest report/code. You are only going to be graded by your blackboard submission. We are not going to accept a version on your harddrive or cloud storage
- You are going to submit a **single** compressed archive through the blackboard site. The file can have *zip*, *rar*, *tar*, *tar.gz* or *7z* format.
- You are fine as long as the compressed archive has the required files (within 4 folder levels).
- Code that does not run (e.g. due to syntax errors) or that does not terminate (e.g. due to infinite loops) will not receive any credits.
- Once you are sure about your assignment and the compressed file, submit it through Blackboard.
- **DO NOT SUBMIT CODE THAT DOES NOT TERMINATE OR THAT BLOWS UP THE MEMORY.**

Best of luck and happy coding!