# Assignment 1

Part A

Gary Geunbae Lee

CSED342 - Artificial Intelligence

Contact: TA Yejin Jeon (jeonyj0612@postech.ac.kr)

Before moving onto the main part of the assignment (Part B), you will complete several simple review questions about heuristic search. Refer to the class slides before answering the questions. Write your answers to Word and save the file as "PartA_studentID.docx".

Note: There is no partial credit in this section of the assignment.
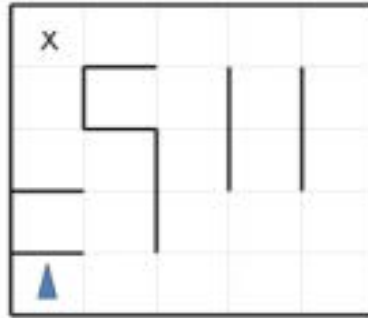
Image a car-like agent that wishes to exit a maze like the one shown above.

This agent is directional and at all times faces some direction $d \in$ (N, S, E, W). With a single action, the agent can *either* move forward at an adjustable velocity v or turn. The turning actions are *left* and *right*, which change the agent's direction by 90 degrees. Turning is only permitted when the velocity is zero (and leaves it at zero). The moving actions are *fast* and *slow*. *Fast* increments the velocity by 1 and *slow* decrements the velocity by 1; in both cases, the agent then moves a number of squares equal to its NEW adjusted velocity. Any action that would result in a collision with a wall crashes the agent and is illegal. Any action that would reduce v below 0 or above a maximum speed V_{max} is also illegal. The agent's goal is to find a plan which parks it (stationary) on the exit square using as few actions (time steps) as possible.

As an example, if the agent shown were initially stationary, it might first turn to the east using (*right*), then move one square east using *fast*, then two more squares east using *fast* again. The agent will of course have to *slow* to turn.

Q1) What is the size of the state space when the grid is M by N? Justify your answer. You should assume that all configurations are reachable from the start state. (2 pts)

Q2) What is the maximum branching factor of this problem? You may assume that illegal actions are simply not returned by the successor function. Briefly justify your answer. (2 pts)

Q3) If we used an inadmissible heuristic in A* tree search, could it change the completeness of the search? (2 pts)

Q4) If we used an inadmissible heuristic in A* tree search, could it change the optimality of the search? (2 pts)

Part B

General Instructions

This (and every) assignment has both a written and a programming section.

Write/implement your answers in **submission.py** between

`# BEGIN_YOUR_ANSWER`

and

`# END_YOUR_ANSWER`

✎This icon means a written answer is expected. Some of these problems are multiple choice questions that impose negative scores if the answers are incorrect. So, don't write answers unless you are confident.

⌨This icon means you should write code. you can add other helper functions outside the answer block if you want. Do not make changes to files other than **submission.py**. Rename the file to submission_studentid.py before submission (Replace the studentid with your student ID)

Your code will be evaluated on two types of test cases, *basic* and *hidden*, which you can see in **grader.py**. Basic tests are fully provided to you, so do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in grader.py, but the correct outputs are not. To run all the tests, type

`python grader.py`

This will only tell you whether you have passed the basic tests. For the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., 3a-0-basic) by typing

`python grader.py 3a-0-basic`

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run grader.py.
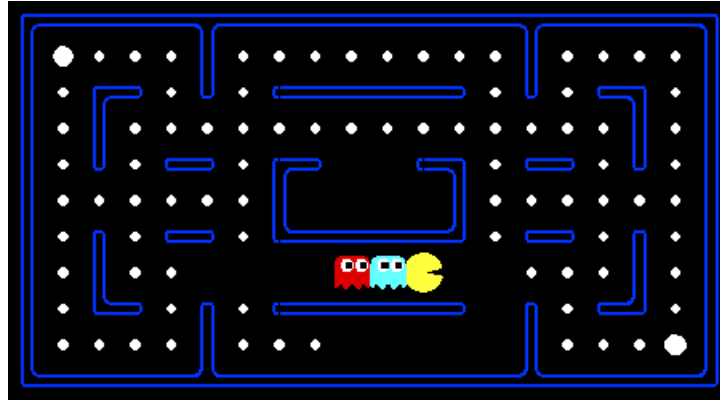
Figure 1: Pac-Man, now with ghosts.

For those of you not familiar with Pac-Man, it's a game where Pac-Man (the yellow circle with a mouth in the above figure) moves around in a maze and tries to eat as many food pellets (the small white dots) as possible, while avoiding the ghosts (the other two agents with eyes in the above figure). If Pac-Man eats all the food in a maze, it wins. The big white dots at the top-left and bottom-right corner are capsules, which give Pac-Man power to eat ghosts in a limited time window (but you won't be worrying about them for the required part of the assignment). You can get familiar with the setting by playing a few games of classic Pac-Man, which we come to just after this introduction.

In this project, you will design agents for the classic version of Pac-Man, including ghosts. Along the way, you will implement both minimax and expectimax search.

Files
- submission.py : Where all of your multi-agent search agents will reside and the only file you need to concern yourself with for this assignment. Write your student ID and name at the top of this file. Rename the file to **submission_studentid.py** before submission (Replace the studentid with your student ID).
- pacman.py : The main file that runs Pac-Man games. This file also describes a Pac-Man GameState type, which you will use extensively in this project
- game.py : The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, A gent, D irection, and G rid.
- util.py : Useful data structures for implementing search algorithms.
- graphicsDisplay.py : Graphics for Pac-Man
- graphicsUtils.py : Support for Pac-Man graphics
- textDisplay.py : ASCII graphics for Pac-Man
- ghostAgents.py : Agents to control ghosts
- keyboardAgents.py : Keyboard interfaces to control Pac-Man
- layout.py : Code for reading layout files and storing their contents

Warmup

First, play a game of classic Pac-Man by running the following command:

    python pacman.py

You can always add `--frameTime 1` to the command line to run in "demo mode" where the game pauses after every frame.

Now, run the provided `ReflexAgent` in `submission.py`:

    python pacman.py -p ReflexAgent

Note that it plays quite poorly even on simple layouts:

    python pacman.py -p ReflexAgent -l testClassic

You can also try out the reflex agent on the default `mediumClassic` layout with one ghost or two.

    python pacman.py -p ReflexAgent -k 1
    python pacman.py -p ReflexAgent -k 2

Note: you can never have more ghosts than the layout permits.
Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using `-g DirectionalGhost`. You can also play multiple games in a row with `-n`. Turn off graphics with `-q` to run lots of games quickly.

So, now that you are familiar enough with the interface, inspect the `ReflexAgent` code carefully (in `submission.py`) and make sure you understand what it's doing. The reflex agent code provides some helpful examples of methods that query the `GameState` (a `GameState` specifies the full game state, including the food, capsules, agent configurations and score changes: see `submission.py` for further information and helper methods) for information, which you will be using in the actual coding part. We are giving an exhaustive and very detailed description below, for the sake of completeness and to save you from digging deeper into the starter code. The actual coding part is very small - so please be patient if you think there is too much writing.
Note: if you wish to run Pac-Man in the terminal using a text-based interface, check out the `terminal` directory.

Problems

Problem 1: Minimax

Before you code up Pac-Man as a minimax agent, notice that instead of just one adversary, Pac-Man could have multiple ghosts as adversaries. So we will extend the minimax algorithm from class (which had only one min stage for a single adversary) to the more general case of multiple adversaries. In particular, *your minimax tree will have multiple min layers (one for each ghost) for every max layer.*

Specifically, consider the limited depth tree minimax search with evaluation functions taught in class. Suppose there are $n + 1$ agents on the board, $a_0, \ldots, a_n$, where a_0 is Pac-Man and the rest are ghosts. Pac-Man acts as a max agent, and the ghosts act as min agents. A single *depth* consists of all $n + 1$ agents making a move, so depth 2 search will involve Pac-Man and each ghost moving two times. In other words, a depth of 2 corresponds to a height of $2(n + 1)$ in the minimax game tree.

Problem 1a [6 points] ⌨

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `submission.py`. Remember that your minimax agent should work with any number of ghosts, and your minimax tree should have multiple min layers (one for each ghost) for every max layer.

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`. The class `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 from the command line options.

Other functions that you might use in the code: `GameState.getLegalActions()` which returns all the possible legal moves, where each move is `Directions.X` for some X in the set `NORTH, SOUTH, WEST, EAST, STOP`. Go through `ReflexAgent` code as suggested before to see how the above are used and also for other important methods like `GameState.getPacmanState()`, `GameState.getGhostStates()`, `GameState.getScore()` etc. These are further documented inside the `MinimaxAgent` class.

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. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state. Use `self.evaluationFunction` in your definition of $V_{\text{max,min}}$.

- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. **You can use these numbers to verify if**

your implementation is correct. Note that your minimax agent will often win (just under 50% of the time for us–be sure to test on a large number of games using the `-n` and `-q` flags) despite the dire prediction of depth 4 minimax.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- Pac-Man is always agent 0, and the agents move in order of increasing agent index. Use `self.index` in your minimax implementation, but only Pac-Man will actually be running your `MinimaxAgent`.

- Functions are provided to get legal moves for Pac-Man or the ghosts and to execute a move by any agent. See `GameState` in `pacman.py` for details.

- 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.

- `getAction` should use $V_{\text{max,min}}$ to determine the best action for Pac-Man.

- On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pac-Man to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. Don't worry if you see this behavior.

- Consider the following run:

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

Why do you think Pac-Man rushes the closest ghost in minimax search on `trappedClassic`?

- You can assume that you will always have at least one action from which to choose in getAction.

- If there is a tie between multiple actions for the best move, you may break the tie.

Problem 2: Alpha-beta pruning

Problem 2a [3 points] ✏️

Let's do Alpha-Beta pruning manually before applying to Pac-Man game. Find the minimax value of the root node and list the label of pruned nodes. It would be helpful to review the quizzes in lecture notes (pages 75 and 76). Write your answer **at the top** of the `submission.py` file (replace the example values with your answers).
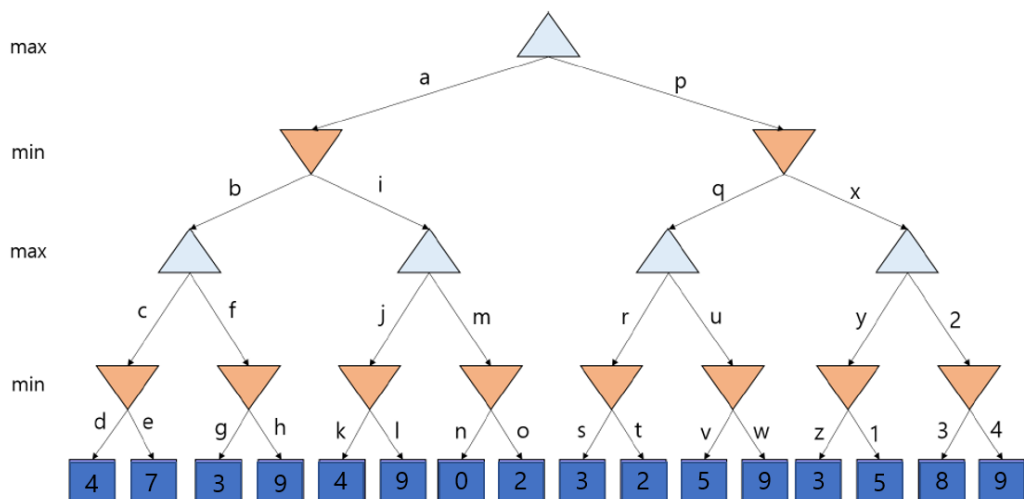
Figure 2: Search tree for Problem 2a

Problem 2b [6 points] ⌨

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 pseudo-code in the slides, so part of the challenge is to extend the alpha-beta Apruning 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 `mediumClassic` should run in just a few seconds per move or faster.

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

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. Running the command given above this paragraph, the minimax values of the initial state should be 9, 18, 27, and 36 for depths 1, 2, 3, and 4, respectively.

Problem 3: Expectimax

Random ghosts are of course not optimal minimax agents, so modeling them with minimax search is not optimal. Let's assume that ghosts follow the uniform policy, therefore ghosts take legal actions uniformly in any state. Before implementing it, first extend Expectimax recurrence, so the algorithm considers multiple ghosts as opponents.Your recurrence should resemble that of Problem 1

8

Problem 3a [6 points] ⌨

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. Assume Pac-Man is playing against `RandomGhost`s, which each choose `getLegalActions` uniformly at random.

   You should now observe a more cavalier approach to close quarters with ghosts. In particular, if Pac-Man perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try:

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

   You may have to run this scenario a few times to see Pac-Man's gamble pay off. Pac-Man would win half the time on an average and for this particular command, the final score would be -502 if Pac-Man loses and 532 or 531 (depending on your tiebreaking method and the particular trial) if it wins (you can use these numbers to validate your implementation). Why does Pac-Man's behavior in expectimax differ from the minimax case (i.e., why doesn't he head directly for the ghosts)?

Problem 4: Evaluation function

Problem 4a [5 points] ⌨

Write a better evaluation function for Pac-Man in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states (rather than actions). You may use any tools at your disposal for evaluation, including any `util.py` code . After implementing it, you can test your code with:

```
python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 20
```

   We will run your Pac-Man agent 20 times, and calculate the average score you obtained. If your average score is less than 700, you'll get no point. If your average score is more than 1500, you'll get 5 points. Check the `grader.py` to see how the scores are calculated.

Hints and Observations

- Having gone through the rest of the assignment, you should play Pac-Man again yourself and think about what kinds of features you want to add to the evaluation function. How can you add multiple features to your evaluation function?

- You may want to use the reciprocal of important values rather than the values themselves for your features (such as distances between Pac-Man and ghosts).

- For your information, our solution code gets more than 1600 scores on average with various random seeds.