Assignment 2

Introduction

In this assignment, you will implement AI techniques learned in class for Pacman and Tic-Tac-Toe.

As in Assignment 1, Assignment 2 includes an autograder for you to grade your answers on your machine. This can be run on all questions of Pacman with the command:

```
python autograder.py
```

It can be run for one particular question:

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

It can be run for one particular test by commands of the form:

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

By default, the autograder displays graphics with the `-t` option, but doesn't with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.

See the autograder tutorial in Assignment 0 for more information about using the autograder.

Related Files

Files you'll edit and submit

File	Description
`multiAgents.py`	Where most of your multi-agent search agents will reside.
`solveTicTacToe.py`	Tic-Tac-Toe code for question 6.

Files you might want to look at

File	Description
`pacman.py`	The main file that runs Pacman games. This file describes a Pacman `GameState` type, which you use 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

Supporting files you can ignore

File	Description
`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
`searchTestClasses.py`	Autograding test classes

Requirements

Files to Edit and Submit: You will fill in portions of `multiAgents.py` and `solveTicTacToe.py` during the assignment. You should submit these two files with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than `multiAgents.py` and `solveTicTacToe.py`.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you may wreak havoc on the autograder.

Academic Dishonesty: We will be checking your code against other submissions in the class and from the Internet for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, please submit questions to the forum.If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Discussion Forum: Please be careful not to post spoilers. Please don't post any code that is directly related to the assignments. However you are welcome and encouraged to discuss general ideas.

Submission: Submit your code `multiAgents.py` and `solveTicTacToe.py` as `a2_ID.zip` file to moodle (ID is your university number).

You will get zero mark if

- you submit the wrong files
- you copy another student's answer
- your zip file's name does not follow the format `a2_ID.zip`
- Your zip uses any other file format than ZIP
- your program contains an infinite loop

Check your files before the submission.

Multi-Agent Pacman

First, play a game of classic Pacman:

```
python pacman.py
```

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

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

Note that it plays quite poorly even on simple layouts:

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

Inspect its code (in `multiAgents.py`) and make sure you understand what it's doing.

Question 1 (3 points): Reflex Agent

Improve the `ReflexAgent` in `multiAgents.py` to play well. The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the `testClassic` layout:

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

Try out your reflex agent on the default `mediumClassic` layout with one ghost or two (and animation off to speed up the display):

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

How does your agent fare? It will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good.

Note: You can never have more ghosts than the layout permits.

Note: As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.

Note: The evaluation function you're writing is evaluating state-action pairs; in later parts of the assignment, you'll be evaluating states.

Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using `-g DirectionalGhost`. If the randomness is preventing you from telling whether your agent is improving, you can use `-f` to run with a fixed random seed (same random choices every game). You can also play multiple games in a row with `-n`. Turn off graphics with `-q` to run lots of games quickly.

Grading: we will run your agent on the `openClassic` layout 10 times. You will receive 0 points if

your agent times out, or never wins. You will receive 1 point if your agent wins at least 5 times. You will receive an additional 1 point if your agent's average score is greater than 500, or 2 points if it is greater than 1000. You can try your agent out under these conditions with

```
python autograder.py -q q1
```

To run it without graphics, use:

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

Don't spend too much time on this question, though, as the meat of the assignment lies ahead.

Question 2 (4 points): 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.

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.

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.

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.getLegalActions`. 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 assignment, 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`
- Make sure you understand why Pacman rushes the closest ghost in this case.

Question 3 (4 points): 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 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.

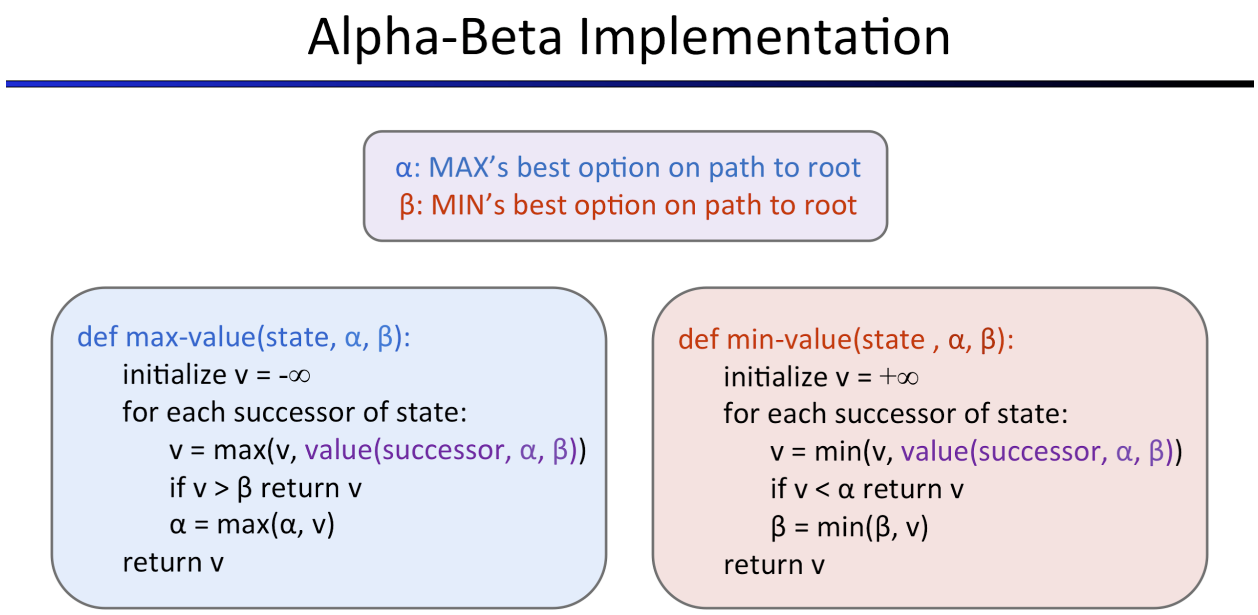
```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

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`.

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.)

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



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
```

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.
- Avoid using system specific values, such as `sys.maxint`, as it could lead to inconsistencies between your machine and the autograders.

Question 4 (4 points): 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 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. 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 5 (5 points): A Better Evaluation Function

Write a better evaluation function for pacman in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. You may use any tools at your disposal for evaluation, including your search code from the last assignment. With depth 2 search, your evaluation function should clear the `smallClassic` layout with two random ghosts 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).

```
python autograder.py -q q5
```

Grading: we will run your agent on the `smallClassic` layout 10 times. 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. +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.
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

Hints and Observations

- As for your reflex agent evaluation function, you may want to use the reciprocal of important values (such as distance to food) rather than the values themselves.
- One way you might want to write your evaluation function is to use a linear combination

of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values and adding the results together. You might decide what to multiply each feature by based on how important you think it is.

Question 6 (6 points):
3-Board Misere Tic-Tac-Toe (Notakto)

Tic-Tac-Toe (also known as Noughts and crosses or Xs and Os) is a two player paper-and-pencil game. Watch this video to learn about the game and its X-Only and Misere variants. In this part of the assignment, you will implement the 3-Board Misere Tic-Tac-Toe version described in the beginning of this video. Assume your AI makes the first move. Implement an AI search algorithm that plays the 3-Board Misere Tic-Tac-Toe version optimally, i.e., never loses.

We provide you with a basic code structure in the `solveTicTacToe.py`. There are six classes in this file:

- `Game`: manages the control flow of the game
- `GameRules`: defines game rules
- `GameState`: game state for 3-Board Misere Tic-Tac-Toe
- `TicTacToeAgent`: the search agent you need to implement
- `KeyboardAgent`: a provided agent decides the action based on your input
- `randomAgent`: a provided agent decides the action randomly

You may try the options when running the program:

- `-n`: Indicates the number of games, default 1
- `-m`: If specified, the program will mute the output
- `-r`: If specified, the first player will be the `randomAgent`, otherwise, use `TicTacToeAgent`
- `-a`: If specified, the second player will be the `randomAgent`, otherwise, use `keyboardAgent`

You can type

```
python solveTicTacToe.py -r
```

to play a game with the random AI.

```
Player 1 (AI): A7
```

```
A      B      C
0 1 2  0 1 2  0 1 2
3 4 5  3 4 5  3 4 5
6 X 8  6 7 8  6 7 8

Your move:
```

You can also type

```
python solveTicTacToe.py -r -a
```

to see two random AI fight with each other. The following shows a full game between two random AI. We by default call the first player to be (AI), the second player to (Human), even though both of them are AI.

```
Player 1 (AI): A7
A      B      C
0 1 2  0 1 2  0 1 2
3 4 5  3 4 5  3 4 5
6 X 8  6 7 8  6 7 8

Player 2 (Human): C4
A      B      C
0 1 2  0 1 2  0 1 2
3 4 5  3 4 5  3 X 5
6 X 8  6 7 8  6 7 8

Player 1 (AI): A1
A      B      C
0 X 2  0 1 2  0 1 2
3 4 5  3 4 5  3 X 5
6 X 8  6 7 8  6 7 8

Player 2 (Human): B2
A      B      C
0 X 2  0 1 X  0 1 2
3 4 5  3 4 5  3 X 5
6 X 8  6 7 8  6 7 8

Player 1 (AI): A0
A      B      C
```

```
X X 2  0 1 X  0 1 2
3 4 5  3 4 5  3 X 5
6 X 8  6 7 8  6 7 8

Player 2 (Human): C5
A      B      C
X X 2  0 1 X  0 1 2
3 4 5  3 4 5  3 X X
6 X 8  6 7 8  6 7 8

Player 1 (AI): A6
A      B      C
X X 2  0 1 X  0 1 2
3 4 5  3 4 5  3 X X
X X 8  6 7 8  6 7 8

Player 2 (Human): A2
B      C
0 1 X  0 1 2
3 4 5  3 X X
6 7 8  6 7 8

Player 1 (AI): C6
B      C
0 1 X  0 1 2
3 4 5  3 X X
6 7 8  X 7 8

Player 2 (Human): B7
B      C
0 1 X  0 1 2
3 4 5  3 X X
6 X 8  X 7 8

Player 1 (AI): C0
B      C
0 1 X  X 1 2
3 4 5  3 X X
6 X 8  X 7 8

Player 2 (Human): C3
B
```

```
0 1 X
3 4 5
6 X 8

Player 1 (AI): B1
B
0 X X
3 4 5
6 X 8

Player 2 (Human): B0
****Player 1 wins game 1!!****

****Player 1 wins 1/1 games.****
```

Requirements

- Implement the `getAction()` function of the class `TicTacToeAgent` in `solveTicTacToe.py`.
- You can add helper functions inside the `GameRules` and `TicTacToeAgent`, but do not remove anything.
- Your AI should not take unreasonable time (defined as > 30 seconds on our grading machine) for making a move. When your AI exceeds this time limit, the program will throw an error and stop.

Evaluation

- An autograder for this question will be provided. The provided autograder will play 30 games with your agent using a `randomAgent`.
- Obviously, it's hard to say your implementation is correct even though your agent can always beat a randomAgent. However, we can not provide an optimal agent in the autograder (We can not give you the answer!). When we evaluate your code, we will use an optimal agent to play with yours.
- If you want to make sure your agent is optimal (with high likelihood), you can initialize both of the players to be the `TicTacToeAgent` and play lots of games. If the first player wins all the games, your implementation is probably correct.

Acknowledgments

This work is based on previous work by John DeNero and Dan Klein et al. of berkeley.edu.