CSE 537 Project 2:
Multi-Agent Pac-Man and Sudoku.
Due Oct. 16th 11:59PM

What to submit: Please submit a zip file containing two folders separately for two problems: PacMan and Sudoku. For the Multi-Agent PacMan problem, you should submit modified source files only. Please *do not* change the other files in this distribution or submit any of the original files. For the Sudoku problem, if you wrote a stand-alone program, please submit all the source code of your progam. If you implemented sudokuSolver.py, please submit sudokuSolver.py only. All the project submissions must be made through blackboard.

Report (10 points) Please write two separate reports for Multi-Agent Pac-Man and Sudoku and put them in corresponding folders. Two reports should both include stats such as number of nodes expanded and running time for each search strategy that you have used in this project, and conclude with critical analysis of the search methods based on your collected stats.

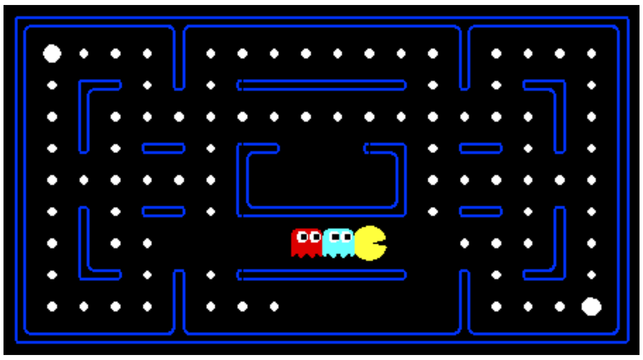
Document (5 points) Your code will be reviewed for good documentation.

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 will wreak havoc on the autograder. However, the correctness of your implementation --not the autograder's judgements -- will be the final judge of your score.

Team up: You can have the same team as before.

Demo with TA: The presenter in each team will be randomly selected by Oct. 17th. The spreadsheet of the appointment will be out on Oct. 17th.

Multi-Agent Pac-Man



I'm gonna fake it to the left, and move to the right;
'Cause Pokey's too slow, and Blinky's out of sight.
I've got Pac-Man fever; It's driving me crazy.
I've got Pac-Man fever; I'm going out of my mind.
(from Pac-Man Fever by Buckner and Garcia)

In this problem, you will design agents for the classic version of Pac-Man, including ghosts. 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. You can, however, use your search.py and searchAgents.py in any way you want.

The code for this project contains the following files, available as multiagent.zip.

Key files to read

multiAgents.py	Where all of your multi-agent search agents will reside.
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, Agent, Direction, and Grid.
util.py	Useful data structures for implementing search algorithms.

Files you can ignore

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

First, play a game of classic Pac-Man, preferably while listening to Pac-Man Fever:

```
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 (5 points) Improve the ReflexAgent in multiAgents.py to play respectably. 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 project, 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.

The autograder will check that your agent can rapidly clear the openClassic layout ten times without dying more than twice or thrashing around infinitely (i.e. repeatedly moving back and forth between two positions, making no progress).

```
python pacman.py -p ReflexAgent -l openClassic -n 10 -q
```

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

Question 2 (20 points) 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 appears in the textbook. 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 a fix depth, which will be specified at the command line. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction. MinimaxAgent extends MultiAgentAgent, 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 Pac-Man move and all the ghosts' responses, so depth 2 search will involve Pac-Man and each ghost moving two times.

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.
- 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
  ```

- To increase the search depth achievable by your agent, remove the Directions.STOP action from Pac-Man's list of possible actions. Depth 2 should be pretty quick, but depth 3 or 4 will be slow. Don't worry, the next question will speed up the search somewhat.

- Pac-Man 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 Pac-Man 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 Pac-Man 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 Pac-Man rushes the closest ghost in this case.

Question 3 (20 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 textbook, 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.

Acknowledgement

Thanks for Mohammad Mahdi Javanmard and Chaitanya Kommini for putting this project together.

Sudoku

Call me the "Sudokiller".
(Image: wikimedia.org)

In this problem, we solve Sudoku puzzles. Starting with backtracking, you will need to add MRV heuristic + forward checking + arc consistency to the algorithm. You can write your own stand-alone program in any programming languages you like. Or you can implement a provided Python function.

The code for this project contains the following files, available as CSP.zip.

Key files to read

puzzle.txt	Get to know the format of the input Sudoku.
sudokuSolver.py	The main file that solves the Sudoku puzzle. If you want to do use Python for this problem, you will fill out the solve_puzzle function.
solution.txt	The output of sudokuSolver.py.

Files you can ignore

sudokuGenerator.py	This file generates a random input Sudoku: puzzle.txt
sudokuChecker.py	This file validates the output of the Sudoku solver: solution.txt
sudokuUtil.py	Contains some supporting functions.

You can run the following command in sequence to get an idea of the pipeline:

Generate a puzzle. Write the puzzle to puzzle.txt.

```
python sudokuGenerator.py
```

You can change the number of given digits in the puzzle (default 30):

```
python sudokuGenerator.py 45
```

Read puzzle.txt and solve it. Write the solution to solution.txt

```
python sudokuSolver.py
```

Read solution.txt and check it

```
python sudokuChecker.py
```

You have two options: write your only stand-alone program with any programming languages you like, or implement the solve_puzzle function in sudokuSolver.py.

If you are going to write your own program, your should ignore sudokuSolver.py completely and replace it with your own program. You program should take puzzle.txt as the input file and write to solution.txt as output. Remember to check your solution by sudokuChecker.py.

If you like to deal with Python, we've done the I/O part in sudokuSolver.py for you and you can focus on the algorithm. You can also use data structures in util.py or add your utilities in sudokuSolver.py.

Question 4 (20 points) Implement the basic backtracking algorithm. Generate 10 random Sudokus with puzzle.txt and evaluate the solver. Report the average and variance of the performance.

Question 5 (20 points) Add minimum remaining values (MRV) heuristic + forward checking + arc consistency to the algorithm. To make arc consistency simpler, you can trigger constraint propagation only when a variable with just one valid value left is discovered. Generate 10 random Sudokus with puzzle.txt and evaluate it. Report the average and variance of the performance.