Artificial Intelligence

Assignment 4: Multi-Agent Search

Dahuin Jung
School of Computer Science and Engineering
Soongsil University

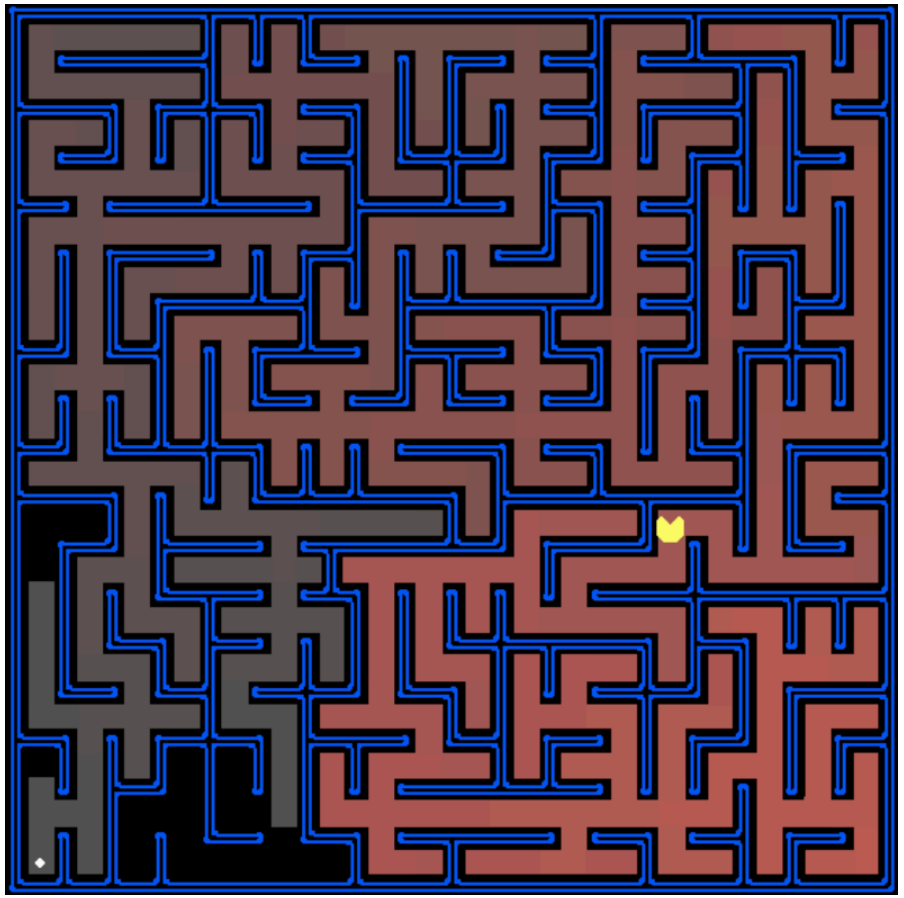
2024

Table of Contents

Table of Contents

Introduction

- In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement minimax search.



Files

- The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

Files you'll edit:	
`multiAgents.py`	Where all of your multi-agent search agents will reside.

Files you might want to look at:	
`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:	
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts

Supporting files you can ignore: (cont'd)	
`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`	Assignment 4 specific autograding test classes

Files

- **Files to Edit and Submit:**
  - You will fill in portions of `multiAgents.py` during the assignment. You should submit these files with your code.
  - Please upload only the specified file to the LMS assignment submission section.

Files you'll edit:	
`multiAgents.py`	Where all of your multi-agent search agents will reside.

Autograding

- The command

  `python autograder.py`

  grades your solution to three problems.
- If we run it before editing any files we get a page or two of output:
- Once the implementation for each solution is completed, you can remove the raiseNotDefine() function.

```
Question q1
===========

Pacman died! Score: -749
Pacman died! Score: 28
Pacman died! Score: -403
Pacman died! Score: -58
Pacman died! Score: -226
Pacman died! Score: -364
Pacman emerges victorious! Score: 479
Pacman died! Score: -82
Pacman died! Score: -203
Pacman died! Score: -302
Average Score: -188.0
Scores:        -749.0, 28.0, -403.0, -58.0, -226.0, -364.0, 479.0, -82.0, -203.0, -302.0
Win Rate:      1/10 (0.10)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss
*** FAIL: test_cases/q1/grade-agent.test (0 of 4 points)
***     -188.0 average score (0 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (0 of 0 points)
***         Grading scheme:
***          < 10:  fail
***         >= 10:  0 points
***     1 wins (0 of 2 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  0 points
***         >= 5:  1 points
***         >= 10:  2 points

### Question q1: 0/4 ###


Question q2
===========

*** Method not implemented: getAction at line 138 of multiAgents.py
*** FAIL: Terminated with a string exception.

### Question q2: 0/5 ###


Question q3
===========

*** Method not implemented: getAction at line 150 of multiAgents.py
*** FAIL: Terminated with a string exception.

### Question q3: 0/5 ###
```

Autograding

- For each of the four questions, this shows the results of that question's tests, the questions grade, and a final summary at the end.

- Because you haven't yet solved the questions, all the tests fail.
  - As you solve each question you may find some tests pass while other fail.
  - When all tests pass for a question, you get full marks.

Table of Contents

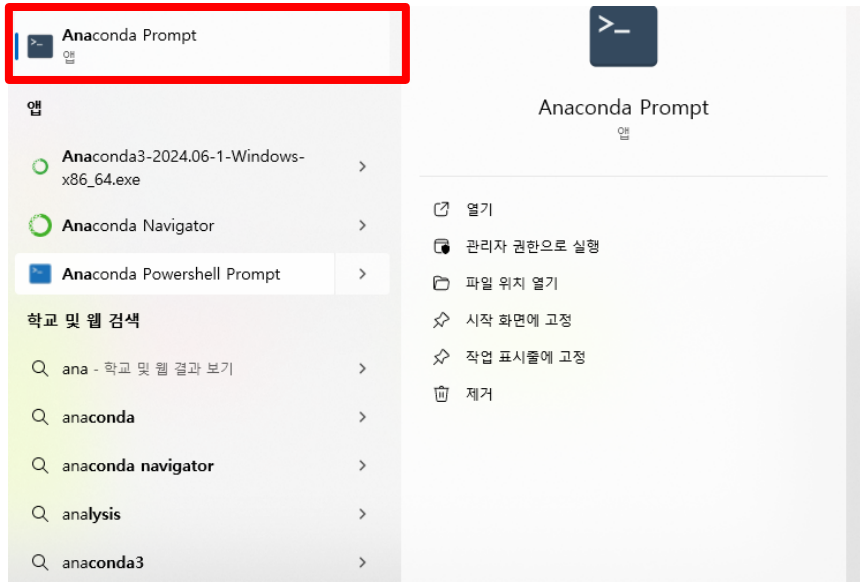
Setup

- At this assignment, we do not need GPUs. Please work on the assignment on your own computer/ laptop (Environment settings can be referenced later in the local setup).
  - To run the code, first run the following command, `conda activate AI-24`

```
(base) C:\Users\ssu_hai>conda activate AI-24

(AI-24) C:\Users\ssu_hai>
```

Local setup

- Step 1 Anaconda download
  - https://www.anaconda.com/download/success (download)



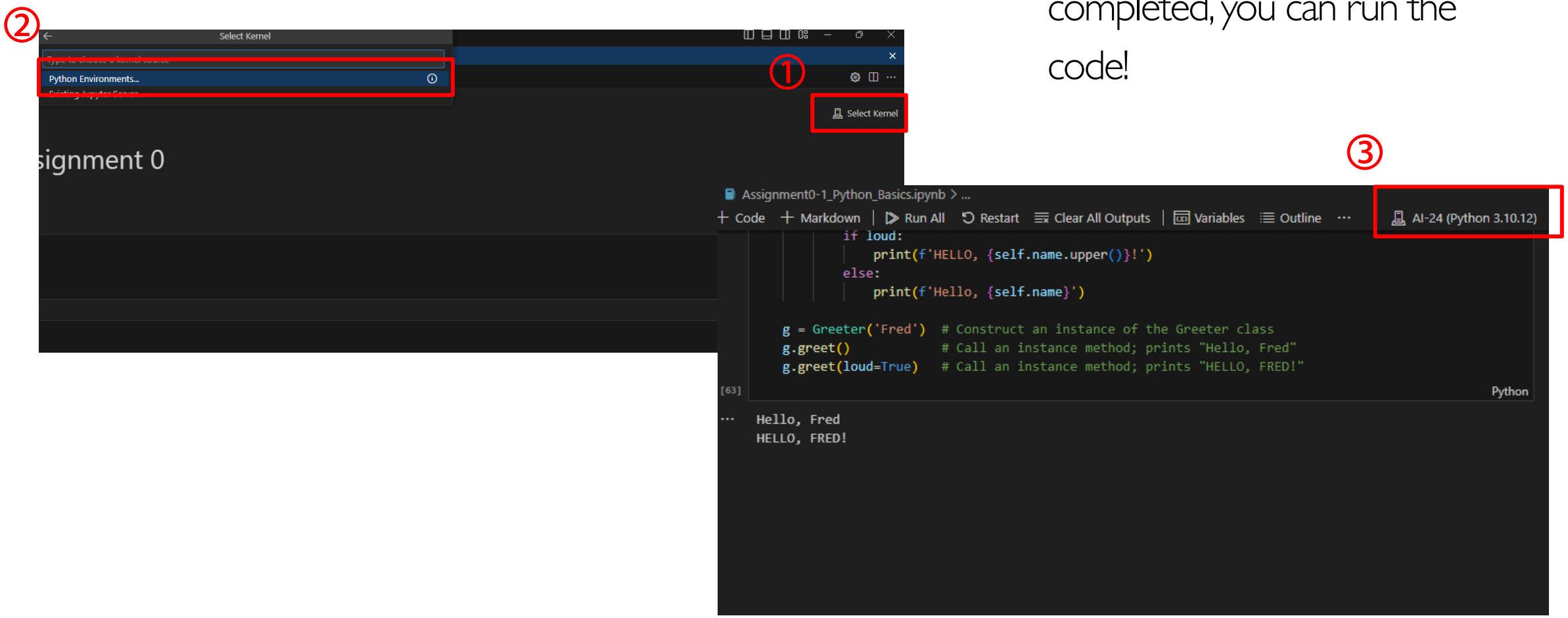
- Step 2 Join to Anaconda prompt and activate AI-24
  - 1. conda env create -f environment.yml(Use the cd command to navigate to the env folder and then run this command.)
  - 2

```
(base) C:\Users\ssu_hai\Desktop\인공지능\Assignment0>cd env

(base) C:\Users\ssu_hai\Desktop\인공지능\Assignment0\env>conda env create -f environment.yml
```

```
(base) C:\Users\ssu_hai>conda activate AI-24

(AI-24) C:\Users\ssu_hai>
```

Local setup

- Step 3 Verification of activation



- Once steps 1 to 3 are completed, you can run the code!

Table of Contents

Welcome to Multi-Agent Pacman

- After downloading the code (`AS4_multiagentsearch.zip`), unzipping it, and changing to the directory, you should be able to play a game of classic Pacman by running the following command:

```
python pacman.py
```

- and using the arrow keys to move

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

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

Welcome to Multi-Agent Pacman

- 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

Table of Contents

Q1: Reflex Agent

- 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
```

Q1: Reflex Agent

- 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.

Q1: Reflex Agent

- *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, or 2 points if your agent wins all 10 games.
- You will receive an addition 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
```

Table of Contents

Q2: Minimax

- Now you will write an adversarial search agent in the provided `MinimaxAgent` class 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.

Q2: Minimax

- 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.

Q2: Minimax

- *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
```

- *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.

Q2: Minimax

- *Hint*:
  - The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behavior, 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
```

Table of Contents

Q3: 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
```

Q3: Alpha-Beta Pruning

- 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.

Q3: Alpha-Beta Pruning

- *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.
  - 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.

- *You must not prune on equality in order to match the set of states explored by our autograder.*

Q3: Alpha-Beta Pruning

- 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
```

Table of Contents

Autograding

- As you solve each question you may find all tests pass.
  - When all tests pass for a question, you get full marks.

```
*** PASS: test_cases/q3/0-eval-function-lose-states-1.test
*** PASS: test_cases/q3/0-eval-function-lose-states-2.test
*** PASS: test_cases/q3/0-eval-function-win-states-1.test
*** PASS: test_cases/q3/0-eval-function-win-states-2.test
*** PASS: test_cases/q3/0-lecture-6-tree.test
*** PASS: test_cases/q3/0-small-tree.test
*** PASS: test_cases/q3/1-1-minmax.test
*** PASS: test_cases/q3/1-2-minmax.test
*** PASS: test_cases/q3/1-3-minmax.test
*** PASS: test_cases/q3/1-4-minmax.test
*** PASS: test_cases/q3/1-5-minmax.test
*** PASS: test_cases/q3/1-6-minmax.test
*** PASS: test_cases/q3/1-7-minmax.test
*** PASS: test_cases/q3/1-8-minmax.test
*** PASS: test_cases/q3/2-1a-vary-depth.test
*** PASS: test_cases/q3/2-1b-vary-depth.test
*** PASS: test_cases/q3/2-2a-vary-depth.test
*** PASS: test_cases/q3/2-2b-vary-depth.test
*** PASS: test_cases/q3/2-3a-vary-depth.test
*** PASS: test_cases/q3/2-3b-vary-depth.test
*** PASS: test_cases/q3/2-4a-vary-depth.test
*** PASS: test_cases/q3/2-4b-vary-depth.test
*** PASS: test_cases/q3/2-one-ghost-3level.test
*** PASS: test_cases/q3/3-one-ghost-4level.test
*** PASS: test_cases/q3/4-two-ghosts-3level.test
*** PASS: test_cases/q3/5-two-ghosts-4level.test
*** PASS: test_cases/q3/6-tied-root.test
*** PASS: test_cases/q3/7-1a-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-1b-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-1c-check-depth-one-ghost.test
*** PASS: test_cases/q3/7-2a-check-depth-two-ghosts.test
*** PASS: test_cases/q3/7-2b-check-depth-two-ghosts.test
*** PASS: test_cases/q3/7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases/q3/8-pacman-game.test

### Question q3: 5/5 ###


Finished at 13:04:16

Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
------------------
Total: 14/14
```

Files

- **Files to Edit and Submit:**
  - You will fill in portions of `multiAgents.py` during the assignment. You should submit these files with your code.
  - Please upload only the specified file to the LMS assignment submission section.

Files you'll edit:	
`multiAgents.py`	Where all of your multi-agent search agents will reside.

Submitting your work

- Submitting your work
  - `multiAgents.py`
  - You can modify this file, save it, and then submit it directly to the LMS.


Thank You!