Artificial Intelligence

Assignment 3: Search

Dahuin Jung

School of Computer Science and Engineering

Soongsil University

2024

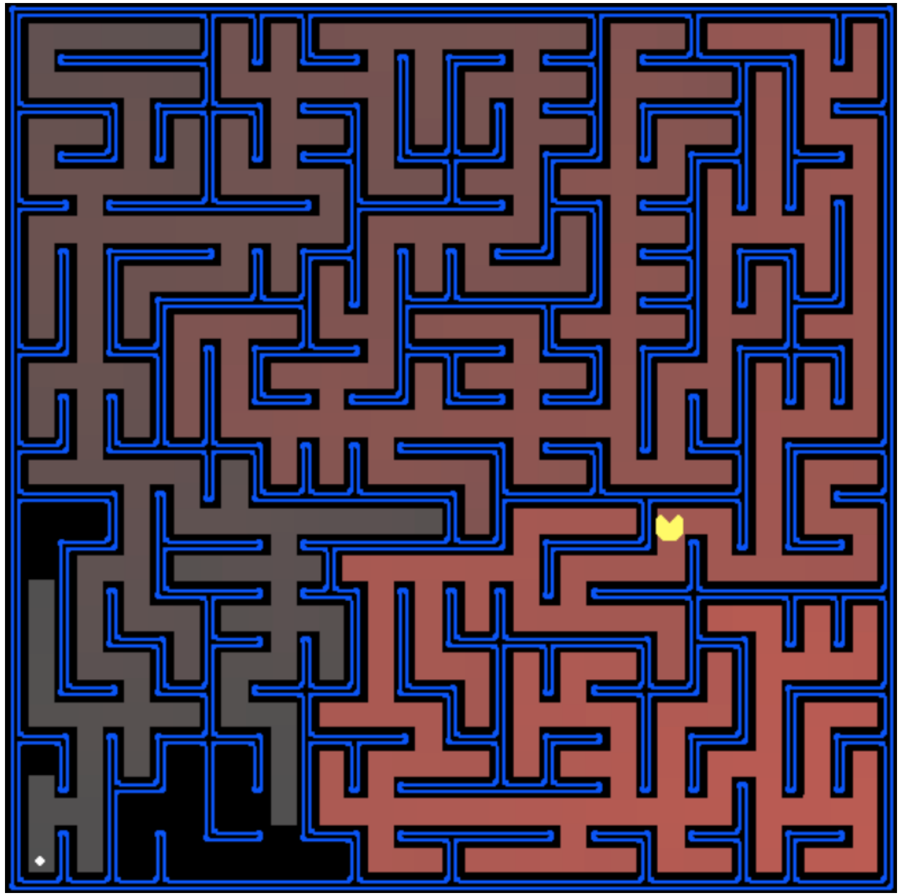
Soongsil University

Table of Contents

Table of Contents

Introduction

- In this project, your Pacman agent will find paths through his maze world. You will build general search algorithms and apply them to Pacman scenarios.



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:	
`search.py`	Where all of your search algorithms 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.
`searchAgents.py`	Where all of your search-based agents will reside.
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 3 specific autograding test classes

Files

- **Files to Edit and Submit:**
  - You will fill in portions of `search.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:	
`search.py`	Where all of your search algorithms will reside.

Autograding

- The command

  `python autograder.py`

  grades your solution to four 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.

```
Starting on 10-29 at 13:18:31

Question q1
===========

*** Method not implemented: depthFirstSearch at line 90 of search.py
*** FAIL: Terminated with a string exception.

### Question q1: 0/3 ###


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

*** Method not implemented: breadthFirstSearch at line 95 of search.py
*** FAIL: Terminated with a string exception.

### Question q2: 0/3 ###


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

*** Method not implemented: uniformCostSearch at line 100 of search.py
*** FAIL: Terminated with a string exception.

### Question q3: 0/3 ###


Question q4
===========

*** Method not implemented: aStarSearch at line 112 of search.py
*** FAIL: Terminated with a string exception.

### Question q4: 0/3 ###


Finished at 13:18:31

Provisional grades
==================
Question q1: 0/3
Question q2: 0/3
Question q3: 0/3
Question q4: 0/3
------------------
Total: 0/12
```

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

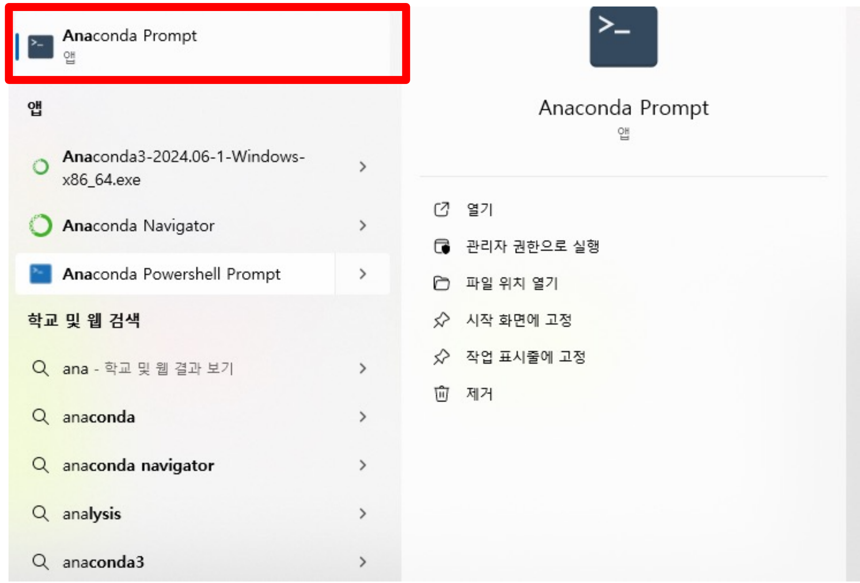
Setup

- 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.)

```
(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
```

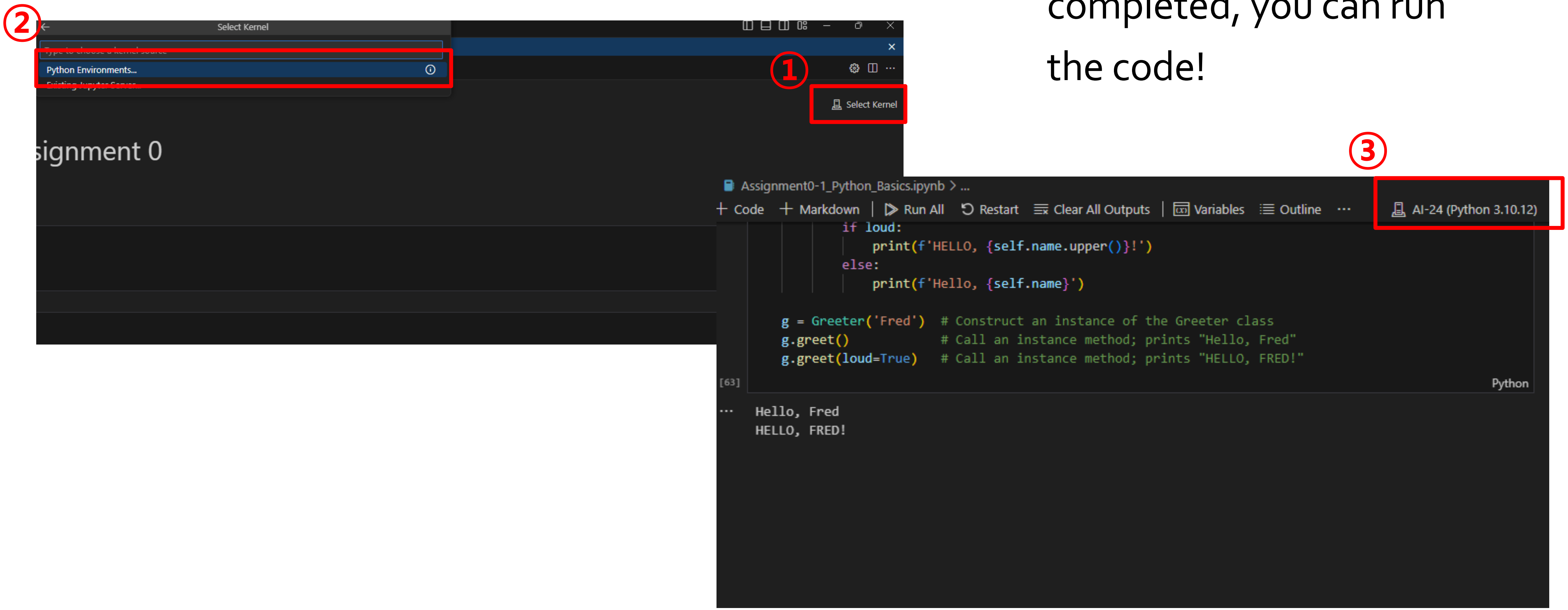
  - 2. conda activate AI-24

```
(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 Pacman

- After downloading the code (`AS3_search.zip`), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

- Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

Welcome to Pacman

- The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

- But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

- If Pacman gets stuck, you can exit the game by typing `CTRL−c` into your terminal.

Welcome to Pacman

- Soon, your agent will solve not only `tinyMaze`, but any maze you want.

- Note that `pacman.py` supports a number of options that can each be expressed in a long way or a short way. You can see the list of all options and their default values via:

```
python pacman.py -h
```

- Also, all of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with `bash commands.txt`.

Table of Contents

Q1: Finding a Fixed Food Dot using Depth First Search

- In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step.
  - The search algorithms for formulating a plan are not implemented -- that's your job.

- First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

- The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Q1: Finding a Fixed Food Dot using Depth First Search

- Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture notes.

- Remember that a search node must contain not only a **state** but also the **information necessary to reconstruct the path (plan) which gets to that state**.

- *Important note:* All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Q1: Finding a Fixed Food Dot using Depth First Search

- Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`.
  - To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

- *Important note:* Make sure to **use** the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Q1: Finding a Fixed Food Dot using Depth First Search

- Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=dfs
```

- The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Table of Contents

Q2: Breadth First Search

- Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs
```

- Does BFS find a least cost solution? If not, check your implementation.

Table of Contents

Q3: Varying the Cost Function

- While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

- By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Q3: Varying the Cost Function

- Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`.

- We encourage you to look through `util.py` for some data structures that may be useful in your implementation.

Q3: Varying the Cost Function

- You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

- *Note*: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Table of Contents

Q4: A* search

- Implement A* graph search in the empty function `aStarSearch` in `search.py`.

- A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information).

- The `nullHeuristic` heuristic function in `search.py` is a trivial example.

Q4: A* search

- You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

- You should see that A* finds the optimal solution slightly faster than uniform cost search.

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.

```
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###


Question q4
===========

*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Finished at 1:38:15

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
------------------
Total: 12/12
```

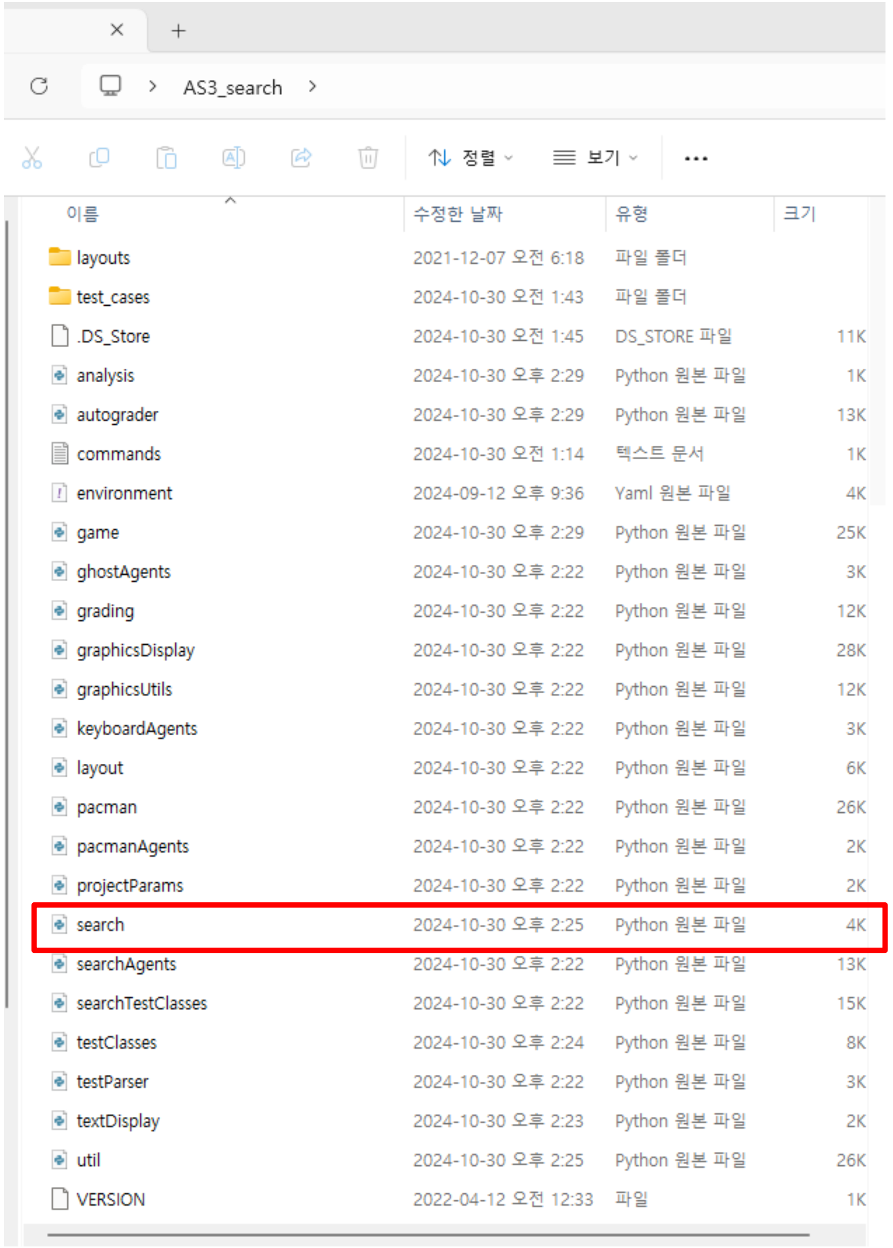
Files

- **Files to Edit and Submit:**
  - You will fill in portions of `search.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:	
`search.py`	Where all of your search algorithms will reside.

Submitting your work

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




Thank You!