

Problem A

Please refer to main Assignment 1 document for all submission instructions and dates

Introduction

In this homework, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently.

On Canvas there is a zip file with the contents of the Pacman game around which you will be developing code to implement various search techniques. You should unzip the file and run it to get an idea of how it works. For instance, to play a game of classic Pacman, run:

```
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.

Note: A description of the files involved in this project can be found in Appendix A at the end of this document.

Agents

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` (or equivalent) into your terminal.

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 (e.g., `--layout`) or a short way (e.g., `-l`). 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`.

Useful Information

Checking Your Work, you can check your code by running

```
python autograder.py
```

This will run your functions through a series of test cases, similar to those used on gradescope. Questions 4b and 5 are not autograded, so the autograder will only give you a score out of 7, the remaining 3 points will be manually graded.

Note. Gradescope is slow and will not give you as detailed feedback as `autograder.py` so its a good idea to test offline first!

Files to Edit and Submit: You will fill in portions of `search.py` and `serchAgents.py`. You should submit these files with your code and comments, as well as your final report. Please do not change the other files in this distribution or submit any of our original files other than these files. You may want to review other files in the project, these have been shown in Appendix A

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. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Q1. [2 pts] Depth First

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. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

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.

Now it’s time to write full-fledged generic search functions to help Pacman plan routes! We have discussed each of these methods in class, please refer to the slides for implementation guidelines. 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).

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.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).

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.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

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?

Hint: If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMaze should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order).

Q2. [1 pt] 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 -p SearchAgent -a fn=bfs -z .5
```

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Q3. [2 pts] Uniform Cost

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.

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. 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).

Q4. [4 pts] A* Search

- (a) [2 pts] 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.

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 (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on openMaze for the various search strategies?

- (b) [2 pts] Implement two new heuristics in `searchAgents.py` in the empty functions `euclideanHeuristic` and

`randomHeuristic`.

`euclideanHeuristic` should return the euclidian distance to the target.

`randomHeuristic` should return a random integer value between 1 and 10.

Use the following commands to test your heuristics:

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

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

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

In your report, Discuss your new heuristics, are they both admissible and consistent? Why? Then compare the three heuristics (manhattan, euclidean, and random) in terms of performance and explain the differences in performance.

Q5. [1 pt] Algorithm Comparison

In your report answer the following questions: What happens on `openMaze` for the various search strategies? Describe the behaviour seen and explain why it occurs.

A Files in the project

Files you'll edit:

| | |
|------------------------------|--|
| <code>search.py</code> | Where all of your search algorithms will reside. |
| <code>searchAgents.py</code> | Where all of your search-based agents will reside. |

Files you might want to look at:

| | |
|------------------------|--|
| <code>pacman.py</code> | The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project. |
| <code>game.py</code> | The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid. |
| <code>util.py</code> | Useful data structures for implementing search algorithms. |

Supporting files you can ignore:

| | |
|-----------------------------------|--|
| <code>graphicsDisplay.py</code> | Graphics for Pacman |
| <code>graphicsUtils.py</code> | Support for Pacman graphics |
| <code>textDisplay.py</code> | ASCII graphics for Pacman |
| <code>ghostAgents.py</code> | Agents to control ghosts |
| <code>keyboardAgents.py</code> | Keyboard interfaces to control Pacman |
| <code>layout.py</code> | Code for reading layout files and storing their contents |
| <code>autograder.py</code> | Project autograder |
| <code>testParser.py</code> | Parses autograder test and solution files |
| <code>testClasses.py</code> | General autograding test classes |
| <code>test_cases/</code> | Directory containing the test cases for each question |
| <code>searchTestClasses.py</code> | Project 1 specific autograding test classes |

Table 1: Files in Project