Project 1: Search

Submit Assignment

Due Thursday by 11:59pm **Points** 21 **Submitting** a file upload **File Types** py
Available until Oct 4 at 11:59pm

Introduction

Download project code: p1_search.zip

This project applies the concepts of single-agent search to the world of Pacman. You will implement DFS, BFS, UCS, and A* Search, then encode a search problem along with a corresponding heuristic for A* Search.

An autograder is included in the code. You can run the autograder from the command line with the following prompt.

```
python3 autograder.py
```

The points you receive from the autograder reflect what you will receive for each question in this project. Keep in mind that not all questions in the autograder may be required. Be sure to consult the rubric for a breakdown of what is needed to complete the project.

You can also autograde a single question using the `-q` option with an argument of the form `q#`. See the example below, which only autogrades question 1.

```
python3 autograder.py -q q1
```

Files that you will edit and submit

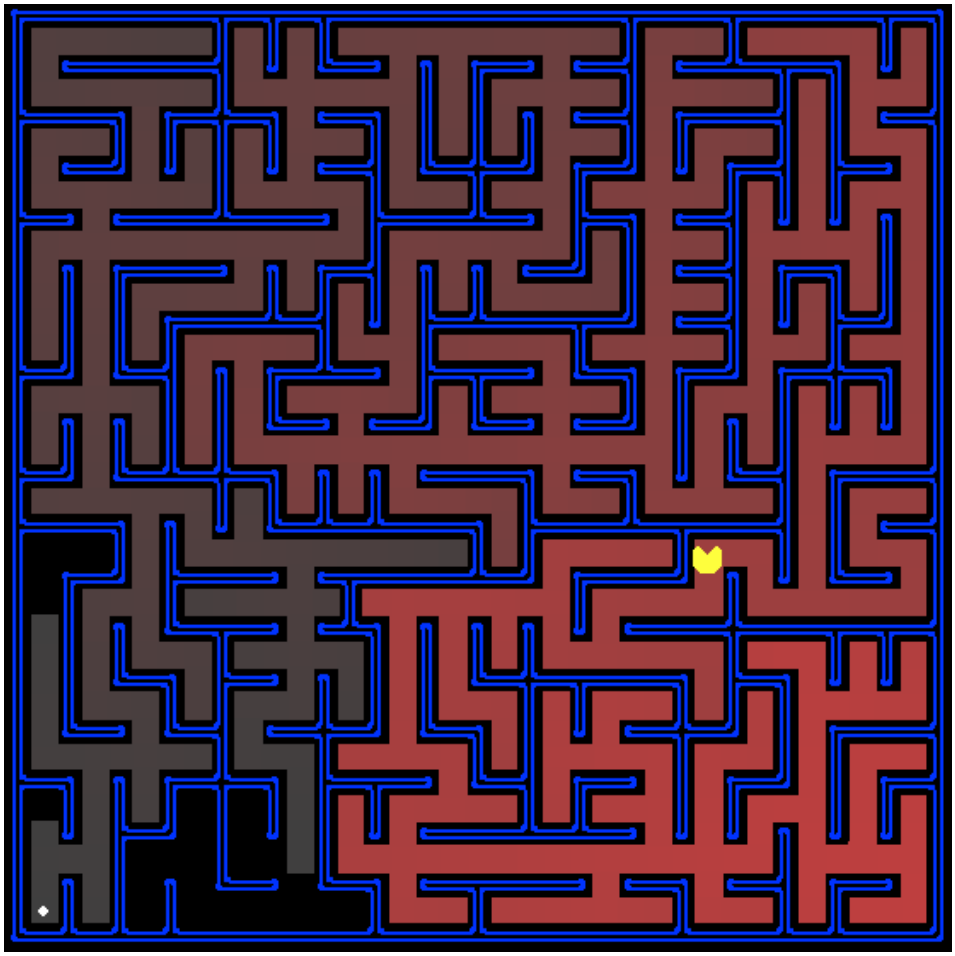
- `search.py` - Where all the search algorithm will reside
- `searchAgents.py` - Where all the search-based agents will reside

Files that you may 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

You are welcome to look at the other files included in the project, but you should only need to read through the files specified above in order to complete the project.

Do not edit any files other than the ones that you will submit. Your code will be graded against the provided project code, so if your code is dependent on changes you've made to files that are not required of you to submit, you will end up losing points.



Welcome to Pacman

After changing into the root directory of the code archive file, you should be able to play a game of Pacman by using the following command:

```
python3 pacman.py
```

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

```
python3 pacman.py --layout testMaze --pacman GoWestAgent
```

However, things don't go as well when Pacman has to turn:

```
python3 pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c 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:

```
python3 pacman.py -h
```

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.

Test that the `SearchAgent` is working correctly by running:

```
python3 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`. `tinyMazeSearch` is simply returns a hard-coded sequence of actions specifically for solving `tinyMaze`. Pacman should navigate the maze successfully.

Important Implementation Note: A crucial element of your search algorithm implementations is that the provided code does not explicitly outline what a node in the search tree looks like. Moreover, you aren't actually building a tree data structure. That is simply how we visualize what is happening in the algorithm.

You will add a node to the fringe as the first part of your search algorithm. This is where you determine what the node structure contains. In addition to a game state, a node must also contain a sequence of actions (the path so far), and – for the algorithms that need it – a cumulative cost. The return value of problem.getSuccessors() should give you a hint as to how this can be easily implemented.

Question 1: Depth First Search

This is the first of a handful of search algorithms you will implement. Getting DFS to work correctly requires the most overhead since it's the first question and there is a fair amount of provided code to look through. Rest assured that the subsequent search algorithm implementations will be much easier (perhaps trivially easy if your code is written flexibly) once you have DFS completed.

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:

```
python3 pacman.py -l tinyMaze -p SearchAgent
```

```
python3 pacman.py -l mediumMaze -p SearchAgent
```

```
python3 pacman.py -l bigMaze -z .5 -p SearchAgent
```

If Pacman moves too slowly for you, add the option `--frameTime 0` to the command.

The Pacman board will show an overlay of the states explored and the order in which they were explored (brighter red means earlier exploration).

Hint: If you use a `Stack` (found in `util.py`) 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).

Hint: It is possible to write a generic graph search function that takes in the fringe data structure as an argument. After all, DFS, BFS, UCS, and A* Search are all just variations of graph search with the only difference being how the fringe is managed. You are not required to write your code in this manner (that is, you are allowed write out the entire search algorithm for each search function separately), but this does shorten your code significantly. As an extra challenge, note that it's also possible to implement all of these search algorithms using a Priority Queue for the fringe (again, not required, simply food for thought).

Question 2: 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.

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Question 3: Uniform Cost Search

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`.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. Be sure 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):

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python3 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).

Question 4: 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.

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`).

```
python3 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?

Question 5: Finding All Corners Search Problem

Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! (The shortest path through `tinyCorners` takes 28 steps.)

Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

```
python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Question 6: Corners Problem Heuristic

Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Coming up with heuristics can be very tricky. It is possible to get 1 or 2 (or 3, in fact) points on this question with fairly

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost *c*, then taking that action can only cause a drop in heuristic of at most *c*.

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Your score for this question will depend on how many nodes A* Search expands using your heuristic:

Number of nodes expand	**Grade**
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

Submission

When you are finished, submit your completed `search.py` and `searchAgents.py` files to Canvas.

Project 1: Search

Criteria	Ratings		Pts
Coding style Code is cleanly written with comments providing non-redundant supplementary explanation	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 1 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 2 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 3 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 4 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 5 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 6 Completed as described in project description	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
			Total Points: 21.0