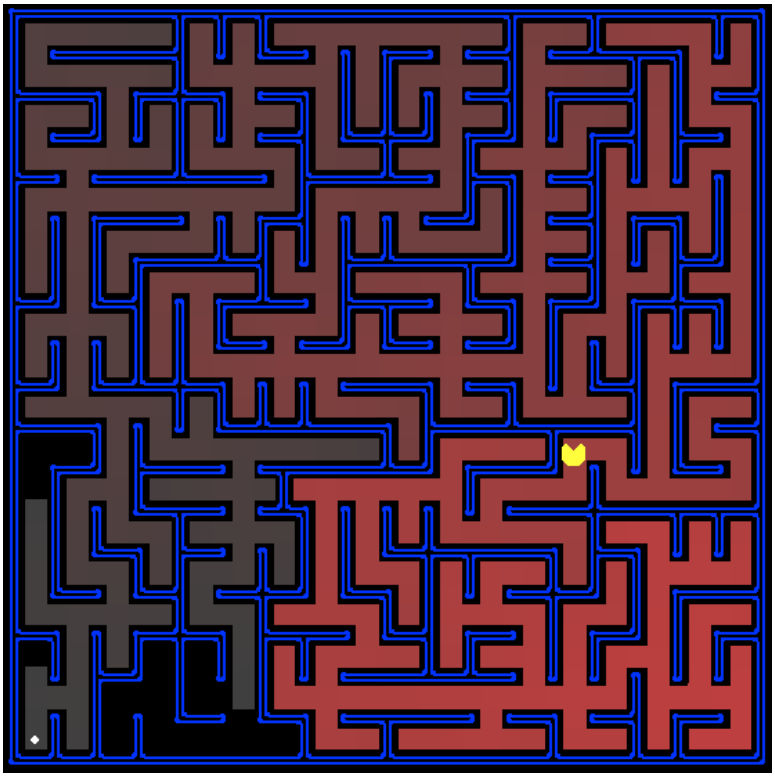
CSCI 360 Project #1: Hungry Pacman Searches for Food!

Released: February 5, 2021

Due: February 19, 2021



Contents

Introduction

In this project, you will help a Pacman agent find food in his maze world by implementing four search algorithms — Breadth-First Search, Depth-First Search, Uniform Cost Search, and A* Search. Your algorithms will not have to deal with enemies (i.e. ghosts) yet BUT you will be able to play a complete version of Pacman for fun!

As you may know, we will be using Python in this class, including this project. If you're new to this language, don't sweat it! This project will only test your ability to implement these search algorithms; the Python-specific syntax to learn will be minimal. Nonetheless, we strongly encourage you to start early and check out Berkeley's Python tutorial if you need a brief introduction/refresher for Python.

Getting Started: First things first, you'll need to set up the environment. The codebase requires *Python 3.6*. If you already have it, you're good to go! If not, there are a plethora of options. One is Anaconda — a Python package manager. Again, Berkeley's Python tutorial can help you with this setup. You are welcome to use a simple text editor or IDE; Atom and PyCharm are solid options respectively.

Regardless, we strongly encourage use of the command line to run the scripts, especially *autograder.py* to make sure you're getting full marks before submitting! Plus, becoming familiar with command line tools is an indispensable asset as a computer scientist. If you're running Windows and want a Mac/Linux-type terminal, check out Cmder!

Download *search.zip*, unzip it, and change to the root directory. Run the following command to play a game (use the arrow keys to move):

```
python pacman.py
```

Can you beat it? It's tough! If you're looking for even more challenge, try replacing the ghost's strategy with one that actively hunts you or actively avoids you when they're "scared":

```
python pacman.py -g DirectionalGhost
```

Note: If you're interested, you can see a complete list of command line options in *commands.txt*!

Evaluation: You'll just be editing two files — *search.py* and *searchAgent.py*. You'll be able to check for correctness as you complete the project with the *autograder.py*. Simply run:

```
python autograder.py
```

Each question is worth **three** points. Except for Question 6, which offers partial points based on nodes expanded, you will only receive full credit if your code passes all the test cases. Don't worry though! The test cases are reasonable and the *autograder* offers a detailed breakdown for each. There are no "reserved" test cases; the score the *autograder* produces is what you'll receive.

When you're ready to submit, you'll run *submit.py* which will generate a unique *submit.token* that you will then submit to Blackboard, along with *search.py* and *searchAgent.py*.

FOR EASE OF GRADING, PLEASE ZIP THESE THREE FILES!

Question 1: Depth-First Search (DFS)

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
```

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 slides. 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 solutions 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). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2: Breadth-First Search (BFS)

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
```

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

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
```

Question 3: Uniform-Cost Search (UCS)

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
```

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*).

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

Question 5: Finding All the Corners with BFS

Our new search problem is to find the shortest path through the maze that visits all four corners. Note that for some mazes like *tinyCorners*, the shortest path does not always go to the closest food first! **Hint**: the shortest path through *tinyCorners* takes 28 steps.

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

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:

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

Hint: 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.

Our implementation of *breadthFirstSearch* expands just under 2000 search nodes on *mediumCorners*. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 6: Finding All the Corners with A*

Note: 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*.

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

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.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

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. Depending on how few nodes your heuristic expands, you'll be graded:

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

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful!

Submission

Once you're happy with your score that the *autograder* gives you, run `python submit.py` to generate a unique *submit.token*. Again, **PLEASE ZIP** this token along with your two edited files — *search.py* and *searchAgents.py*. Call this *submission.zip* or anything else that makes sense. And then submit to Blackboard!


I AM ONCE AGAIN ASKING...
PLEASE ZIP YOUR SUBMISSION FILES!
imgflip.com