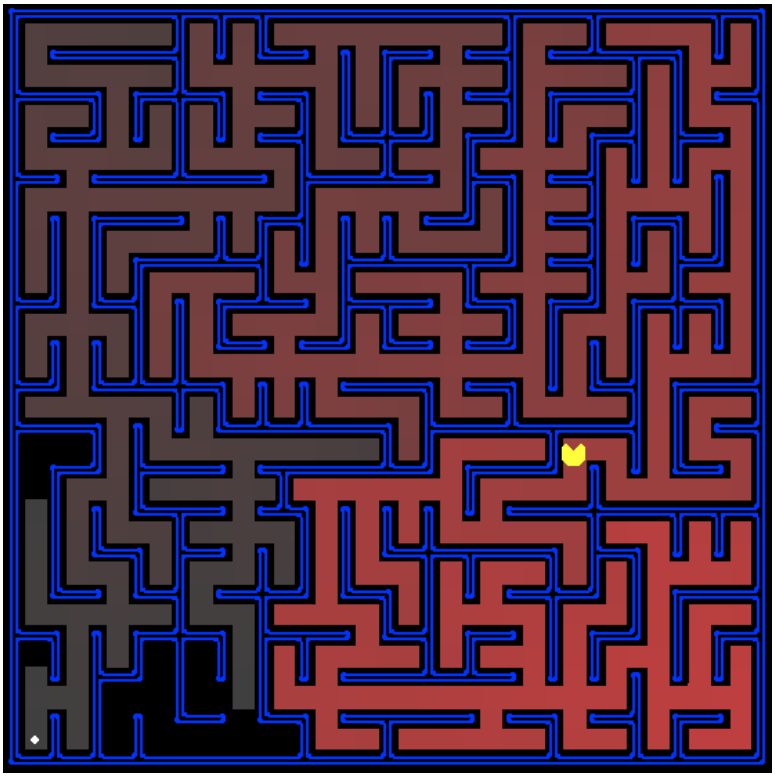
CSCI 360 Project #1: Hungry Pacman Searches for Food!

Released: January 28, 2022

Due: February 11, 2022



Contents

Introduction 2

Question 1: Depth-First Search (DFS) - 3 pts 3

Question 2: Breadth-First Search (BFS) - 3 pts 4

Question 3: Uniform-Cost Search (UCS) - 3 pts 4

Question 4: A* Search - 3 pts 5

Question 5: Finding All the Corners with BFS - 3 pts 5

Question 6: Finding All the Corners with A* - 3 pts 6

Extra Credit Exploration: Pacman Explores Superfoods <> 0-2 pts! **7**

Submission **8**

Introduction

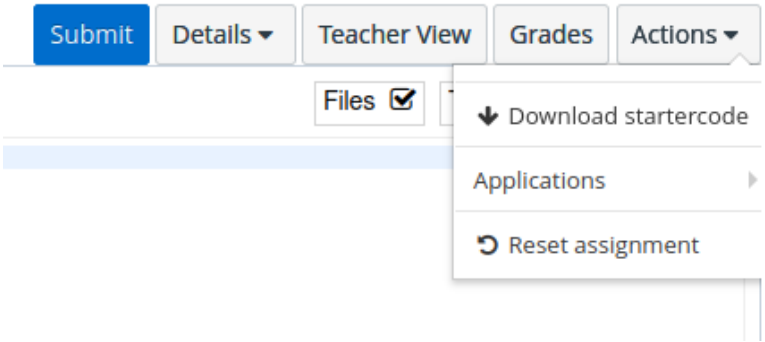
In this project, you will help a Pacman agent find food in his maze world by implementing four search algorithms — Breadth-First Search, Depth-First Search, Uniform Cost Search, and A* Search. Your algorithms will not have to deal with enemies (i.e. ghosts) yet BUT you will be able to play a complete 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 Project 0 to find resources for a brief Python introduction/refresher!

Getting Started: In Project 0, we shared how to enroll in and use Vocareum, the assignment management software we're using this semester. We also shared how to install conda and use it to create an environment. This pacman codebase, borrowed from UC Berkeley[1], requires *Python 3.6* or later.

To work on and submit this assignment,

1. complete Vocareum and environment setup, as per Project 0
2. on the Vocareum project 1 workspace, download the codebase by clicking "Download startercode" under "Actions" (screenshot below)



3. fill in the missing code segments within *search.py* and *searchAgent.py* as per the instructions in this handout
4. upload these two files back to Vocareum and submit

Once you've downloaded the codebase, navigate 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

[1] http://ai.berkeley.edu/project_overview.html

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
```

Hint: You may find that your for some of these questions, your code solves pacman just fine but the autograder reports that you failed a few graph test cases. If this is because of node expansion, take a closer look at when you test if you've visited a state. What code should get executed if not?

Question 1: Depth-First Search (DFS) - 3 pts

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.

Hint: You may find that your for some of these questions, your code solves pacman just fine but the autograder reports that you failed a few graph test cases. If this is because of node expansion, take a closer look at when you test if you've visited a state. What code should get executed if not?

Question 2: Breadth-First Search (BFS) - 3 pts

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) - 3 pts

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 - 3 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
```

Question 5: Finding All the Corners with BFS - 3 pts

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* - 3 pts

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!

Vocareum leaderboard: For some extra fun, when you submit to Vocareum, the number of nodes your pacman agent expands in this question will be published to a classroom leaderboard!

Extra Credit Exploration: Pacman Explores Superfoods <> 0-2 pts!

Congrats on making it this far! If you feel called to dive a little deeper, then continue reading for an extra-credit exploration and friendly competition. Otherwise, go ahead and submit :)

For most of his ghost-fighting life, Pacman has eaten a rather bland unbalanced diet of pellets and occasional capsules. Then the other day, on suggestion from his doctor, Pacman went to Whole Foods and saw some quinoa and chia seeds for the first time in his life. This inspired him to hop on the "superfood" train and explore a more diverse diet!

Your mission then, should you choose to accept it, is to help Pacman find diverse food sources for a more balanced diet. More specifically, there are now **three** food types and his doctor suggests he should eat **at least two** of each type on a regular basis.

Setup: we added a new file *superFoodSearch.py* that has startercode similar to what is found in Questions 5 and 6 for implementing the class *SuperFoodSearchProblem* and function *superFoodHeuristic*. We also made changes to other files within the codebase. To integrate these changes, please follow this flow:

1. Download the startercode from Vocareum, as before
2. Replace the *search.py* and *searchAgents.py* with the versions you've worked on thus far
3. Append the following code snippets to **the end** of your *searchAgents.py*:

```
from superFoodSearch import SuperFoodSearchProblem, superFoodHeuristic
class AStarSuperFoodAgent(SearchAgent):
"A SearchAgent for SuperFoodSearchProblem using A* and your superFoodHeuristic"
def __init__(self):
    self.searchFunction = lambda prob: search.aStarSearch(prob, superFoodHeuristic)
    self.searchType = SuperFoodSearchProblem
```

SuperFood with BFS Search (+1 pt): Make sure to complete Question 2 before working on this part. Implement the class *SuperFoodSearchProblem* within *superFoodSearch.py*. This search problem finds paths to eat at least two of each type of food. Similar to question 5, implement a suitable state space, goal state check, and successor function. You can test your code on the following layouts:

```
python pacman.py -l tinySuperFood -p SearchAgent -a fn=bfs,prob=SuperFoodSearchProblem
python pacman.py -l mediumSuperFood -p SearchAgent -a fn=bfs,prob=SuperFoodSearchProblem
```

Note: Running the mediumSuperFood test could take up to 10 seconds or more depending on your machine and implementation. But if it is taking more than a minute, something might be wrong!

You will receive **one point** if everything is implemented correctly, according to the autograder. If you want to just test this BFS component of the bonus question, then run:

```
python3 autograder.py -q q7
```

SuperFood with A* Search (0-1 pts): Make sure to complete Question 4 before working on this part. Implement the function *superFoodHeuristic* within *superFoodSearch.py*. You can test your heuristic with the same layouts as above:

```
python pacman.py -l tinySuperFood -p AStarSuperFoodAgent
python pacman.py -l mediumSuperFood -p AStarSuperFoodAgent
```

With the autograder, you can also confirm that your heuristic is admissible and consistent:

```
python3 autograder.py -q q8
```

You'll notice that this part is scored out of zero points. After the due date, we'll look at all bonus question submissions and let $x :=$ minimal # of expanded states for *mediumSuperFood*. Let $y := 10900$, which is the baseline # of expanded states derived from one of our implementations of BFS for the same layout. Then, your score for this part will be normalized to $[0, 1]$, based on where your # of expanded states is located on the interval $[x, y]$.

Submit as per the instructions below. As in Question 6, your # of expanded states will be published to the leaderboard for friendly competition!

Submission

Once you're happy with your score that the *autograder* gives you, upload *search.py*, *searchAgent.py*, and *superFoodSearch.py* (if you chose to do the bonus question) to Vocareum and submit. This will run the same autograder and will publish your # of expanded nodes from Question 6 (as well as the bonus question, if applicable) to the class leaderboard. You are welcome to submit as many times as you'd like before the due date!