Project 1: Search

**Table of Contents**

* Introduction
* Welcome
* Q1: Depth First Search
* Q2: Breadth First Search
* Q3: Uniform Cost Search
* Q4: A\* Search
* Q5: Corners Problem: Representation
* Q6: Corners Problem: Heuristic
* Submission



All those colored walls,  
Mazes give Pacman the blues,  
So teach him to search.

**Introduction**

In this project, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

As in Project 0, this project includes an autograder for you to grade your answers on your machine. This can be run with the command:

python autograder.py

See the autograder tutorial in Project 0 for more information about using the autograder.

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. You can download all the code and supporting files as a zip archive (1.search.zip) on canvas.

|  |
| --- |
| **Files you'll edit:** |
| search.py | Where all of your search algorithms will reside. |
| searchAgents.py | Where all of your search-based agents 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. |
| **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 |
| 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 | Project 1 specific autograding test classes |

**Files to Edit and Submit:** You will fill in portions of search.py and searchAgents.py during the assignment. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

**Welcome to Pacman**

After downloading the code (1.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.

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.

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.

**Question 1 (3 points): 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.

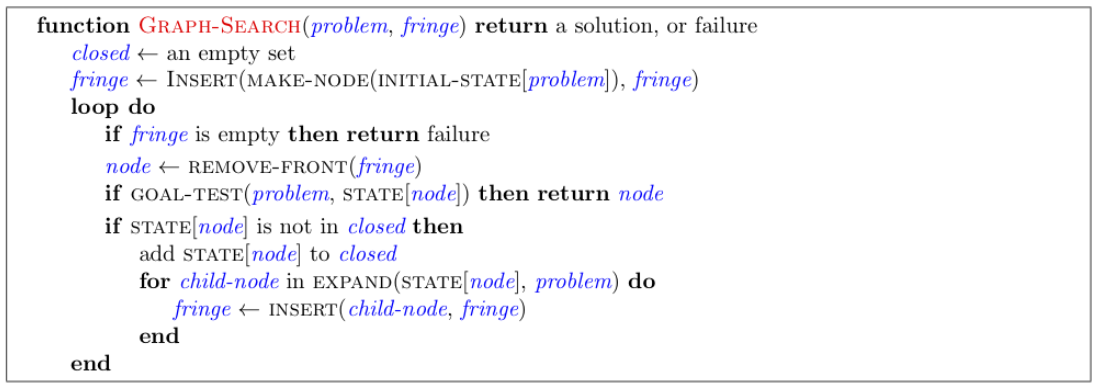
First, test that the SearchAgent is working correctly by running:

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.

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 below. 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.

For the INSERT below in the loop, we discussed at least two possible ways you can handle it. The first one is to add it to the fringe even when there is already a node for the same state on the fringe (a state can appear multiple times). The second way is to keep only one in the fringe (a state appears only once). Compare which way works better.



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

**Question 2 (3 points): 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

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

*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

**Question 3 (3 points): 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.

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).

**Question 4 (3 points): 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

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 (3 points): Finding All the Corners**

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.

In *corner mazes*, there are four dots, one in each corner. 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! *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

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).

*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 (3 points): Corners Problem: Heuristic**

*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, admissible heuristic for the CornersProblem in cornersHeuristic.

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

*Note:* AStarCornersAgent is a shortcut for

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

***Admissibility:*** 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).

We briefly mentioned in class 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. So, design your admissible heuristics accordingly. In rare cases, you might accidentally introduce admissible but inconsistent heuristics. The sign is if UCS and A\* ever return paths of different lengths since UCS is using a heuristic that is both admissible and consistent! If you observe this, modify your heuristic.

***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 admissible 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, autograder may complain. So, make sure to check whether UCS and A\* return the same solution.