

Lab 2: Search Algorithms for Path-finding

There are four exercises in this lab:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Best-First Search
4. A* Search

In this assignment, you will continue to help your Pac-Man agent find paths through his maze world to reach a particular location. You will build general search algorithms and apply them to many different Pac-Man scenarios.

Files you'll edit:

File	Description
search.py	Where all your search algorithms will reside.
searchAgent.py	Where all your search-based agents will reside.

Files you should look at but NOT edit:

File	Description
util.py	Useful data structures for implementing search algorithms.
pacman.py	The main file that runs Pac-Man games. This file describes a PacMan GameState type, which you use in this lab.
game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

PART I: Finding a fixed food dot using Uninformed Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pac-Man'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`. This simply follows a fixed sequence of actions to demonstrate how the code works. Pac-Man should navigate the maze successfully.

Now it's time to write the generic search functions to help Pac-Man plan routes! Pseudocode for the breadth-first search, depth-first search, and A* search algorithms might be useful for the implementation is shown below.

```

function UninformedSearch(problem) returns a list of actions
    initialize the frontier using the initial state of the problem
    initialize exploredSet to empty
    #For explored, use Pacman position as the key with a value True
    #initialize a set of states already explored
    while frontier is not empty do
        choose a leaf node and remove it from the frontier
        if the node contains a goal state
            return list of actions from start state to goal state
        add the state to the exploredSet
        for each successor of the node state
            if the successor state is not in exploredSet
                add node of the successor onto the frontier
    return an empty list (i.e. no solution!)

```

Important note: All your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions must be legal moves (valid directions, no moving through walls).

Hint: Algorithms for DFS and BFS differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and then BFS 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.

Hint: Make sure to check out the `Stack`, `Queue`, and `PriorityQueue` types provided to you in `util.py`.

EXERCISE 1 (25 points):

Implement the depth-first search algorithm in the `depthFirstSearch` function in `search.py`. Although DFS and BFS ignore the costs, you'll need them for later search methods. Your code should quickly find a solution for:

```

python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
python pacman.py -l bigMaze -p SearchAgent -z .5 -a fn=dfs

```

The Pac-Man board will show an overlay of color for 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 Pac-Man actually go to all the explored squares on his way to the goal?

Hint: The solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the frontier in the order provided by `getSuccessors()`; you might get 244 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.

EXERCISE 2 (25 points):

Implement the breadth-first search algorithm in the `breadthFirstSearch` function in `search.py`. Use the same algorithm as shown in the above pseudocode. Test your code the same way you did for depth-first search.

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -z .5 -a fn=bfs
```

Does BFS find a least cost solution? **Hint:** If Pac-Man moves too slowly for you, try the option `--frameTime 0`.

PART II: Finding a fixed food dot using Informed Search

EXERCISE 3 (25 points):

Implement the Best-First Search algorithm in the `bestFirstSearch` function in `search.py`. Test your code the same way you did for other search algorithms.

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=befs
python pacman.py -l mediumMaze -p SearchAgent -a fn=befs
python pacman.py -l bigMaze -p SearchAgent -a fn=befs -z .5
```

Does best-first search find a least cost solution? How many nodes are expanded?

EXERCISE 4 (25 points):

Implement the A* Search algorithm in the `aStarSearch` function in `search.py`. Use the same algorithm as shown in your text (or class). `aStarSearch` function takes an optional heuristic function as an argument. The heuristic function itself takes two arguments (a state in the search problem, and the problem itself). `search.py` provides a `nullHeuristic` function that you can look at. Also, in the `searchAgents.py`, a Manhattan heuristic as well as Euclidian heuristic function is defined. Test your code the same way you did for other search algorithms.

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=astar
python pacman.py -l mediumMaze -p SearchAgent -a fn=astar
python pacman.py -l bigMaze -p SearchAgent -a fn=astar -z .5
```

To specify a heuristic function from `searchAgents.py`, use the following:

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

The function `manhattanHeuristic()` is already written in `searchAgents.py`. Alternately you could write your own in `search.py`.

Object Glossary

Here's a glossary of the key objects in the code base related to search problems, for your reference:

Object	Description
SearchProblem (search.py)	A SearchProblem is an abstract object that represents the state space, successor function, costs, and goal state of a problem. You will interact with any SearchProblem only through the methods defined at the top of search.py.
PositionSearchProblem (searchAgents.py)	A specific type of SearchProblem that you will be working with. It corresponds to searching for a single pellet in a maze.
Search Function	A search function is a function which takes an instance of SearchProblem as a parameter, runs some algorithm, and returns a sequence of actions that lead to a goal. Example of search functions are depthFirstSearch and breadthFirstSearch, which you have to write. You are provided tinyMazeSearch which is a very bad search function that only works correctly on tinyMaze.
SearchAgent	SearchAgent is a class which implements an Agent (an object that interacts with the world) and does its planning through a search function. The SearchAgent first uses the search function provided to make a plan of actions to take to reach the goal state, and then executes the actions one at a time.

What to submit

1. Fill out the table below:

Maze	Depth-First Search			Breadth-First Search		
	#nodes explored	Solution length	Is it optimal?	#nodes explored	Solution length	Is it optimal?
tiny						
medium						
big						

Maze	Best-First Search			A* Search		
	#nodes explored	Solution length	Is it optimal?	#nodes explored	Solution length	Is it optimal?
tiny						
medium						
big						

2. Based on the above, a short discussion/reflection of how the searches compare.
3. Source code: two files include search.py and searchAgents.py. These should include your code for the Depth-First Search, Breadth-First Search, Best-First Search, and A*.
4. Please create a folder called "yourname_studentID_Lab2" that includes all the required files (included the Python practice file) and generate a zip file called "yourname_ studentID_Lab2.zip".
5. Please submit your work (.zip) to Moodle.