Gina Ferguson || 11/17/2025 || CS350 || HW 5: Maze

Problem 1

Part A:

Problem 1 [22 points]

Imagine a maze that consists of an *n* x *n* grid of cells. You can move from a cell to its neighbor provided there is no wall obstructing your path in that direction. Your goal is to find a path from the top right cell in the maze to the bottom left cell.

(a) [5 points] Explain how finding a path through this maze can be represented as a graph problem. Clearly explain what the nodes and edges in your graph formulation correspond to in terms of the structure of the original maze. You may include a diagram or a sketch to explain your reasoning.

- The maze is a graph problem if I model the n x n maze as an undirected graph.
  - NODES:
    - with the nodes (aka vertices), each cell in the maze is one node in the graph. For example, in a 3 x 3 maze, we can number the cells 1-9 in row-major order; in general, all n^2 cells become the n^2 nodes.
  - EDGES:
    - The edge of the maze can be drawn as between two nodes, if you are allowed to move between those cells in the maze (they are adjacent horizontally or vertically and there is no wall between them). If there is a wall, then that edge is simply missing from the graph.
  - START NODE:
    - The start node is the top-right cell, and the goal node is the bottom-left cell (aka goal = the end of the maze). The "Solution" to the maze is then just a path in the graph from the start node to the goal node.
- Once the maze is represented as this graph, finding a path through the maze becomes a standard graph-search problem.
  - Using Breadth-First Search (BFS), I also use a FIFO queue, starting at the top-right node, marking it visited, and then repeatedly "visit" all of its neighbors, then all of their unvisited neighbors.. And so on. BFS explores the graph level by level, so the first time you reach the bottom-left node we have found a valid path from start to goal.
  - There is also the option of using a heap-based priority queue (as in the max-heap / tree structure from lecture) to pick which node to expand next; in both cases, we are just repeatedly taking a current node from a data structure, looking at its neighbor nodes (adjacent cells with no wall), and marking them as visited.
- In the code, the maze is stored as an adjacency list (aka adjacency matrix) that records, for each cell/node, which neighboring cells it connects to. Random mazes are generated by starting from the full grid (all possible edges between adjacent

cells) and then flipping a coin (aka the 50% probability declared in the code) to decide whether that wall is present or not.
    - Running BFS on this adjacency list tells us whether the maze is “solved” (there exists a path from the top-right node to the bottom-left node), and the path itself corresponds exactly to the sequence of cells the user would walk through in the original maze.
- SUMMARY: Using Python, I created a dictionary of lists that serve as the adjacency list (aka matrix), where each index corresponds to a tree node mapped into an array form in row-major order. As each potential wall is decided by a coin flip, edges are either kept or removed, determining which nodes remain connected and therefore reachable in the maze. To check if the maze is solvable from the top-right to the bottom-left cell, I used Breadth-First Search (BFS) to see if there is a path between those two nodes in the adjacency list. In simple terms, BFS works like exploring the maze level-by-level, starting at the beginning and checking all possible moves outward until either the goal is found or all options are exhausted (using a visited status for each index/node)..I used ASCII symbols to draw the maze for the user and mostly so I could test it out clearly.

Part B:

(b) [10 points] In a programming language of your choice, implement an algorithm to find if a given maze has a solution (i.e., if it has a path that connects the top right cell to the bottom left cell). If it has a solution, the algorithm should return this path. If there is more than one possible solution, your algorithm should return any one of those possible paths. If there is no solution to the maze, return a special identifier to indicate that the maze is not solvable. Document your code clearly indicating the input, and sample output for a small maze (say 2x2 or 3x3).

- I chose Python instead of C++ finally. This was new for me, but I do enjoy the user-driven menu style in case the teacher or me want to test multiple amounts for variables etc. To determine whether a maze has a solution I did kind of explain above, but I went further and added the path output naming each cell in order from start to finish for the path. During the process, BFS is storing a parent pointer (the ‘pointer’ is really just the dictionary parents etc) for each visited node. The algorithm can then rebuild and return one valid path from start to finish whenever a solution DOES exist (after the “coin flip” wall-building).
    - **Example from my terminal:**

```
(.venv) gina@Ginas-MBP Python % python HW5-maze-attempt-1.py

Welcome to Gina's Maze Program!
1) Generate one random maze and check if it is solvable
2) Run 1000-maze experiment
3) Quit
Enter choice (1 - 3): 1
Enter maze size n (i.e. 3 for 3x3, 5 for 5x5): 3
```

```
Checking single maze 3x3 from 3 to 7...

ASCII Maze:
+---+---+---+
|  1    2 |  3 |
+   +---+---+
|  4 |  5 |  6 |
+---+---+---+
|  7 |  8 |  9 |
+---+---+---+
Solvable? False... (no path found

Welcome to Gina's Maze Program!
1) Generate one random maze and check if it is solvable
2) Run 1000-maze experiment
3) Quit
Enter choice (1 - 3): 1
Enter maze size n (i.e. 3 for 3x3, 5 for 5x5): 5

Checking single maze 5x5 from 5 to 21...

ASCII Maze:
+---+---+---+---+---+
|  1 |  2    3    4    5 |
+   +---+---+   +   +
|  6    7    8 |  9   10 |
+---+   +   +   +   +
| 11   12   13 | 14   15 |
+---+   +   +---+   +
| 16 | 17   18 | 19   20 |
+---+---+   +---+   +
| 21 | 22   23   24 | 25 |
+---+---+---+---+---+
Solvable? False... (no path found

Welcome to Gina's Maze Program!
1) Generate one random maze and check if it is solvable
2) Run 1000-maze experiment
3) Quit
Enter choice (1 - 3): 3
Thanks and goodbye!
```

Part C:

(c) [2 points] What is the asymptotic complexity of your algorithm in part (b)? (Your answer should depend on the side length *(n)* of the maze.)

- Vertices (V) = the n x n grid/matrix, so V = n^2 cells (nodes)
  - At most, E <= 4V edges
- For graphs like this the formula is O(V+E), and so a 1000 maze test means O(1000*(V+E))... aka O(n^2) because 1000 is a constant.
- Long story short:
  - QUADRATIC run time 🙂... O(n^2)

Part D:

(d) [5 points] Generate 1000 random mazes of size *5x5* by placing a wall between two cells using a 50-50 coin flip. Run your algorithm from part (b) on these 1000 mazes keeping a record of whether each maze had a solution or not. What fraction of these 1000 mazes were solvable? (A maze is solvable if there exists a path from the top right cell to the bottom left cell.)

- I got a little bit ahead of myself but using the 1000 maze test, and the python algorithm, generating 1000 5x5 mazes was about 16-17% solvable. The walls were built using 50/50 probability, and so then the path-finding rules lead to a 16-17% probably on average solvability.

Problem 2:

Problem 2 [5 points] Prove that $n^n \in \Omega(n!)$.

- In my discrete structures courses, I recall induction being a way to get more stable proofs. n^n being a super-exponential, and in the Big Omega set of factorial n, this is saying that the lower bound of n! Includes the rate n^n
- Step 1: I'm going to make this an inequality since there is a lower bound mentioned
  - For all n≥1, $n^n \geq n!$ (the assumption)
  - When n! Is spelled out...
    - $n! = 1*2*3 \ldots n \leq n*n*n \ldots n = n^n$
  - If c = 1 and $n_0 = 1$, then the Ω condition holds...
    - $n^n \geq 1 * n!$, for all $n \geq 1$
- Step 2: Induction, Base case n = 1
  - $1! = 1 \leq 1^1 = 1$
  - Assume $k! \leq k^k$, then.... $(k+1)! = (k+1)k! \leq (k+1) * k^k$
  - Since k+1 > k... $k^k \leq (k+1)^k$
  - Therefore.. $(k+1)! \leq (k+1)^{k+1}$

n^n does live within the lower bound of Ω(n!) ✅

SOURCES:
Video:

To take a look at how python is used with dictionary and Breadth-First Search I watched this 4 minute video:
https://www.youtube.com/watch?v=HZ5YTanv5QE

Set up instructions:

Local:

- cd /Users/Desktop/Python
    - Create Python Virtual Environment (first time only)
        - Run in terminal (in specified Python folder): python3 -m venv .venv
        - Run in terminal (in specified Python folder): source .venv/bin/activate
            - You'll notice "(.venv)" as your folder location in your terminal for the next command onward…
        - Run in the same terminal (both commands at the same time is ok):
            - pip install --upgrade pip
              pip install black flake8 isort pytest ipython
        - Create first Python project by running this in terminal:
            - vim HW5-maze.py
        - In the VIM editor, paste the HW5-Maze.py code text (or …)
        - ESC to leave the Insert mode in VIM, and type ":wqa" to write, quit, all
        - Run the program in the terminal:
            - Python HW5-maze.py
    - Running the program:
        - Source .venv/bin/activate in the Python folder
        - Install NumPy in Python folder
            - Run in the terminal after starting .venv:
                - Pip install numpy

VIM User, not new to using python via terminal:

- In your Python folder, place my file HW5-maze-attempt-1.py inside.
- Start the Python environment by running this in the terminal:
    - Source .venv/bin/activate
- Then run this in the terminal to run the program:
    - Python HW5-maze-attempt-1.py
- You should then see a user menu in the terminal that indicates the program has started for you. Try testing the 3x3, 5x5 and 1000 maze count by selecting the appropriate option in the user-driven menu.