Project 3
Data Structures and Algorithms

Report

Team Name: Group 189, Grease Police
Team Members: James Shen, Hunter Brinkman
GitHub URL: https://github.com/jachetheboss/DSA-Final-Project/upload/main
Video Link: file attached in submission

Project Title: Going the Distance (Maze Pun)

Problem: Investigate the difference between A* and BFS implementations for maze-solving.

Motivation: There are many ways to solve mazes and we want to see how we can apply mazes learned in this class in a real-life puzzle such as mazes. Using the raw breadth-first-search algorithm learned during the trees and graphs unit, and implementing a C++ version of the A* pseudocode algorithm found on (https://en.wikipedia.org/wiki/A*_search_algorithm), we thought this project would be a great introduction to the deep treasure trove of advanced pathfinding algorithms. A* is just the beginning.

Features: We have a console-based step-by-step interactive stdout display of the individual steps of the A* and BFS algorithms. This feature is invaluable for learning the difference between various pathfinding algorithms, as visualization makes code easier to follow.

Data: The input is a cartesian plane in matrix of cells format, with 0 meaning the cell is passable, 1 meaning there is a wall, making the cell impassable. The matrix of cells can be represented by a 2D vector in C++. The grid sizes are 20x20, as 20x20 is a convenient size to fit into stdout.

Tools: We will be coding using C++. The IDE of choice is Notepad++ and Visual Studio.

Visuals: Visuals will be created using stdout.

Strategy: We had a few ideas regarding maze-solving:

i. Breadth-first search (BFS)
ii. Depth-first search (DFS)
iii. Depth-limited Depth-first-search (DLDFS)
iv. Topological dead-end filling
v. A*

We narrowed our choices down to BFS and A*, as BFS is the simplest, and A* is the most advanced of the choices.

Division of Responsibility and Roles:

James will be the primary coder. Hunter will be the tertiary coder.

Project 3
Data Structures and Algorithms

Analysis

We made a few changes after the initial submission of the proposal. We initially aimed to use the BRIDGES API for graphics, but figured that stdout would be more interactive, and simpler to use for the user.

We added a new feature, called wallFactor, to customize the makeup of the initial grid (to increase or decrease the number of walls in the randomly-generated grids).

Time Complexity Investigations:

i. Function reverse(path): O(n) worst case time, with n = path length.

ii. Function reconstruct_path(goal): O(n) worst case time, with n = path length. We retrospectively find the previous cell from which we came from, and build the path in reverse order.

iii. Function h(): O(1). This is the heuristic helper function for A*, providing the minimum possible distance from current node to the goal. This function performs a constant number of abs(), subtracts, and additions, each of which is O(1).

iv. Function getKey(x, y): O(1). This converts the {x,y} coordinates into a single long long which is easily hashable as a key in an unordered_set.

v. Function BFS(grid): O(m*n), where m is #rows, and n is #cols. Worst case BFS finds/does not find a path in a long, winding grid, searching through every single cell in the grid.

vi. Function A*(grid): O(|E|*log(|V|)). The logarithmic time complexity comes from the minheap used in the algorithm. A* is essentially Dijkstra's with an added heuristic function.

vii. Function generateGrid(): O(m^2 * n^2). To generate the grid, a random permutation of each of the m * n coordinates in the grid is generated, and the first K cells (K = wallFactor * m * n) are chosen as the coordinates of the walls. The random permutation is generated by successively deleting coordinates from the list of coordinates, each delete takes O(m*n) time, and we delete m*n times.

Reflection

As a group, the overall experience of the project was a resoundingly positive one. The maze topic was highly engaging. The challenges were in the implementation of the A* algorithm. Even though we used the pseudocode as a guideline, understanding the code and debugging our C++ implementation was tricky (the pseudocode was slightly vague and abstracted).

Project 3
Data Structures and Algorithms

If we redid the project, we would investigate other variables, such as grid size, the effect of wallFactor on the probability that a valid path exists from start (1, 1) to finish (m-2, n-2), and more pathfinding algorithms.

Each of us learned how to program interactively in a stdout environment. Moreover, we learned how to work collaboratively, in a timely manner, via incremental changes. Specifically, my partner resolved the BFS implementation bug where the same cell is pushed twice to the next iteration of the BFS queue in the same iteration. Each cell should only appear in the queue a SINGLE TIME.