**Task C Report**

**James Plecher**

**S3898959**

**Strategy:**

Using a Breadth-First Search algorithm, it starts at the first entrance of the maze and stops exploring the maze once it has found both exits. The nodes are stored as dual information, being the current node, and the previous node. Once both exits are found, it starts at the first exit and follows back all the nodes, using the previous node information to the entrance, counting the amount of nodes travelled. This repeats for all exits. Then, once all shortest paths for entrance one has been found, it repeats for the rest of entrances in the maze. Once complete, it prints the amount of nodes explored plus the shortest distance path (min(cells explored() + distance(entrance, exit))).

**Reasoning**:  
Breadth-First Search (BFS) is a very methodical, consistent and guaranteed way of getting the exits in a maze. The main draw towards choosing BFS is it’s innate ability to back tracking, allowing for the shortest path to be easily determined for all maze to exit combinations. It also allows for a stop solving condition once it finds all exits, negating the need to explore the entire maze. Also theoretically, with BFS, you don’t need to find more than one exit per entrance, as it is guaranteed to be the closest exit to that entrance, which would also reduce the explored nodes, but to be safe I implemented it to find all entrances.

My reasons initial reasons for not choosing wall follower, pledge, recursive backtracking or DFS is that they are all very random by nature, can overlap and re-explore nodes that have already been explored, technically adding to the non-unique exploration count and for all but DFS, not having intuitive ways of calculating the shortest path, even after exploring to the exits. And yet DFS doesn’t guarantee the shortest path, just the shortest path it finds.

My ideal implementation of BFS would have been more efficient than my implemented version, where it would explore the maze once from the first entrance, then for the following entrances, only explore up until where had already been explored instead of to the exits again, but I spent a lot of time and effort and did not figure out a solution to the seemingly simple problem.

**Empirical Evidence:** \*For the sake of space, I did not provide much proof. To get an idea of the numbers I’m claiming, run the tests and see the results if you want\*

A main test I completed is a larger 3x15x15 (Square) maze, with the entrances being on the opposite diagonal of the maze. This produces terrible conditions for BFS. It has to traverse to the other side of the maze, whilst exploring all adjacent nodes, most of which are irrelevant. This is one of the cases where it would presumably be close to the worst case scenario (675 explored nodes + shortest travel path). Yet it still did relatively well, averaging a rough 70-80% full exploration rate. This can be compared to closer exits, wielding results of less than 10% of total exploration.

As I commenced more testing on square mazes (3x5x5, 3x10x10, 3x50x50), I realised the size of the maze wasn’t the concerning factor when it came to BFS, but more so the placement of the exits. If an entrance is extremely close, BFS will find it within very few explorations. The issue came from having to find exits that were very far away from the entrances, leading to exponentially larger exploration needed.

A maze with a few blue triangles

Description automatically generated with medium confidence

Non-square mazes are slightly different however. Depending on the size of each layer, the results can drastically change. Such as if a higher level is much larger than the level with the entrances and exits, most of that level will never be explored, which is a benefit BFS has over any other algorithm. To the right is level two of a non square maze. Its clearly demonstrates the potential advantage of BFS, as other algorithms would possibly traverse much of that area.

Overall I decided whilst it might not be the most optimal solution for every maze (in terms of min(cells explored() + distance(entrance, exit)), it is definitely the most versatile. It will always find the shortest route, most of the time stays below 80% exploration percentage (even for my unoptimized implementation). Algorithms such as DFS usually have an average maze exploration of roughly 65%, which is better than 80%, but its trade-off is it doesn’t provide a guaranteed shortest path and can be quite random and inconsistent in its results. Since I’m unsure of what the algorithm is to be used for, and which maze conditions it will have to solve for, such as the size, perfectness, squareness, entrance and exit positions, I think the versatile option that provides consistent exploration results and a shortest path every time is a highly recommendable choice.