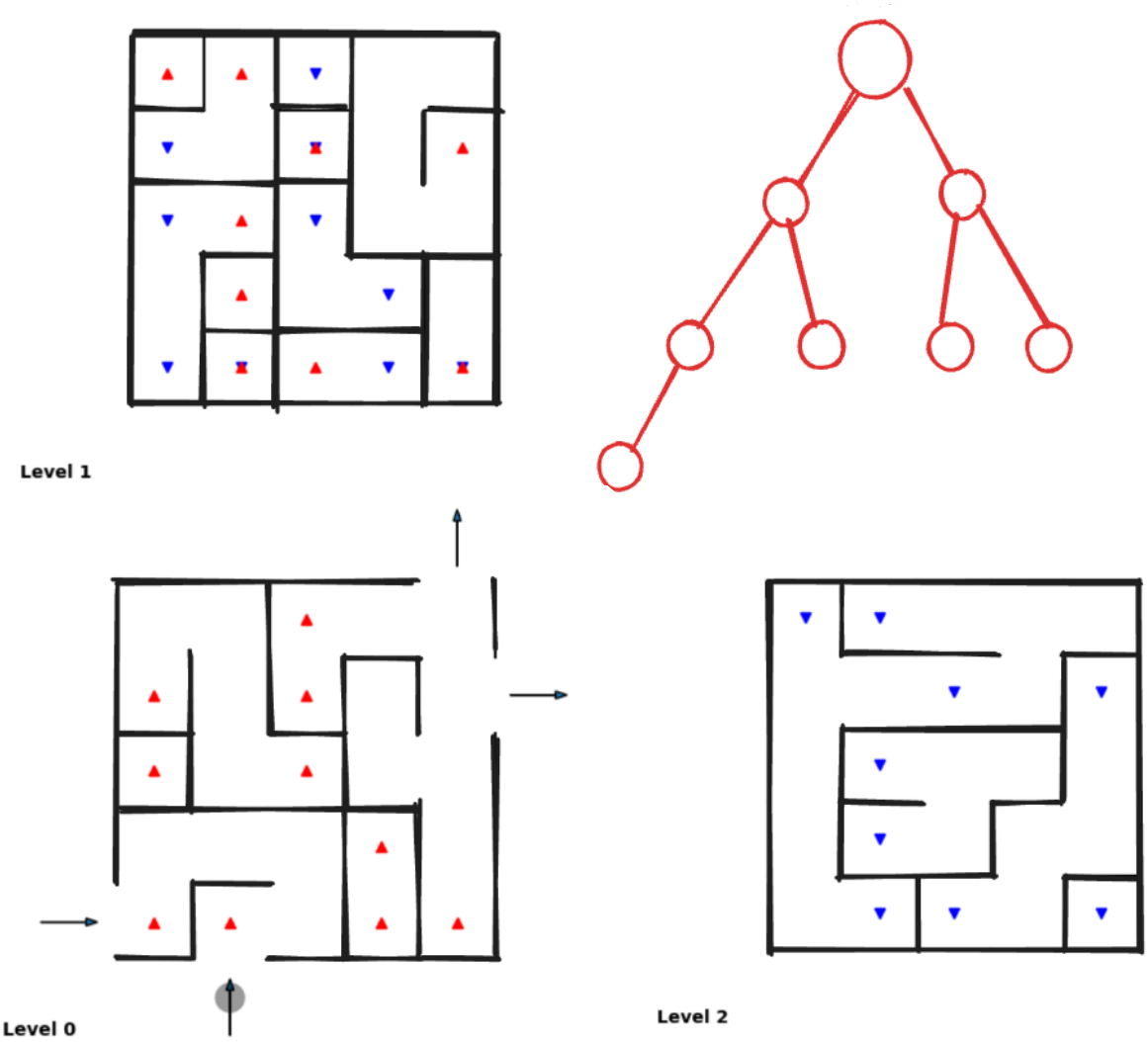
# 3D MAZE SOLVERS




*Mrwan Fahad A Alhandi*

*RMIT University | 21 June 2024*

# Algorithm Choice

For Task C, my solver modifies Dijkstra's Algorithm to solve a 3D maze by minimizing the sum of total cells explored and the distance from entrance to exit. Traditionally, Dijkstra's Algorithm finds the shortest path from a source node to all other nodes in a graph. By using a priority queue (implemented as a heap) to always explore the node with the smallest combined cost (distance plus cells explored), we ensure efficient pathfinding. The heap's property of maintaining the smallest element at the root allows efficient retrieval and removal of the smallest cell using `heappop`. Each time a node is processed, its unvisited neighbors that are not blocked by walls are added to the priority queue with their updated cost. This adaptation balances finding the shortest path with minimizing unnecessary cell exploration. A detailed pseudocode of the algorithm is provided in figure 1.

```
INPUT: A 3D maze
OUTPUT: Shortest path from an entrance to an exit
1. INITIALIZE:
   Set up initial variables and data structures
   min_cost, best_entrance, best_exit, best_path, entrances, exit_count
2. FOR each entrance:
   Initialize priority queue, visited set, and cost dictionaries
3. WHILE priority queue is not empty:
   Pop the smallest element
   IF current cell is visited, continue
   Mark current cell as visited
   IF current cell is an exit:
      Calculate total cost
      IF total cost is less than min cost, update best path
      IF all exits are found, break
   FOR each neighbor:
      IF neighbor is not visited and has no wall:
         Calculate new cost
            IF new cost is better, update and push to queue
   4. IF best path found, mark maze solved and append path
   5. RETURN best path
```

*Figure 1: Dijkstra's Algorithm Pseudocode*

Consider figure 2 maze, the algorithm explores all paths and chooses the shortest one. In the first iteration, the heap will be [(1, 0, 0, 0)], containing the only neighbor. In the second iteration, the heap will be [(2, 0, 0, 1)], where 1 is the cost of the total cell explored. Here, the algorithm must choose which path to take, either up or right. The heap will include both initially, so it will become [(3, 0, 0, 2), (3, 0, 1, 1)]. The order of the neighbors method list is [left,right,below,front,down,up].

The updated heap will be [(3, 0, 1, 1), (4, 0, 0, 3)]. Since the other path now has a shorter total distance, we go up, and the heap becomes [(4, 0, 2, 1), (4, 0, 0, 3)]. In the next iteration, the heap will become [(5, 0, 2, 2), (4, 0, 0, 3)]. The next iteration will check for (4, 0, 0, 3), and since it has no unvisited neighbors, the heap will become [(5, 0, 2, 2)]. Finally, we reach the exit (6, 0, 2, 3).

Once we reach an exit, we check if the total cost from this entrance to the exit is smaller than the shortest path from other entrances to their exits. If it is, we report the total cost and the best path taken. This algorithm are performed for every entrance we have.

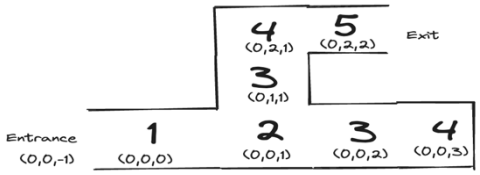



*Figure 2: Example Maze Demonstration*

# Algorithm Justification

## Theoretical Analysis

In terms of time complexity, the RecurBackMazeSolver operates with a depth-first search (DFS) approach, reflected by its O(V + E) complexity, where V and E represent the number of vertices (cells) and edges (walls) respectively. Each cell is explored thoroughly, ensuring complete maze traversal. Dijkstra's Algorithm, exhibits a higher complexity of O((V + E)log V)) due to its use of a priority queue (min-heap). This complexity arises from processing each vertex and edge while continually updating the priority queue to ensure the shortest path is selected. This makes it exceptionally reliable for finding the shortest path in mazes.

Wall Following maintains a simpler and typically linear complexity of O(n), where n refers to the number of edges or walls navigated. The Pledge Algorithm, like Wall Following but enhanced with a turn counter, adapts better to complex mazes. Its time complexity can vary but often approaches linearity under optimal conditions.

Among these, Dijkstra's Algorithm stands out for its robustness in ensuring the shortest path is found as will be seen in the empirical analysis, making it theoretically the slowest due to its comprehensive processing but the most reliable for precision in pathfinding.

## Empirical Analysis

To determine if Dijkstra's algorithm minimizes the cells explored and the distance to the exit most effectively, I ran five trials per solver on each of three different maze generators. All the tests were performed on the same 5x5 mazes using seeds. I averaged these results for each generator and compared the overall performance across all generators. As shown in Figure 3, Dijkstra's consistently yielded the lowest total and unique cell costs, demonstrating its superior efficiency in maze navigation. For detailed results, please refer to the appendix.

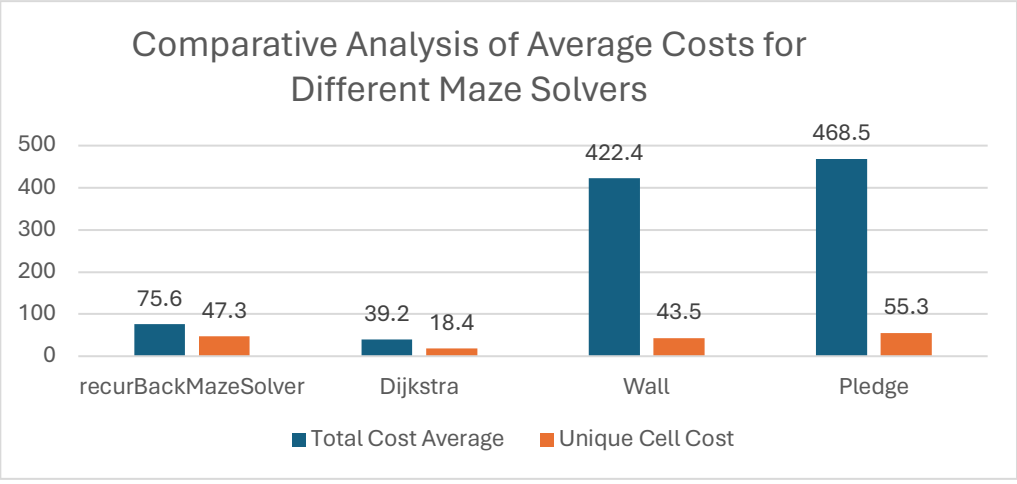



*Figure 3: Comparative Analysis of Maze Solvers*

## Appendix

| Total Cells Explored\|Unique Cells Explored | recurBackMazeSolver | | Dijkstra | | Wall | | Pledge | |
|---|---|---|---|---|---|---|---|---|
| recurBackGenerator | 74.4 | 50.2 | 53.2 | 25.8 | 60.4 | 39.8 | 165.6 | 67.6 |
| primGenerator | 59.4 | 36.6 | 30 | 14.2 | 922.6 | 57.2 | 934.4 | 54.6 |
| wilsonGenerator | 93 | 55 | 34.4 | 15.2 | 284.2 | 33.4 | 305.6 | 43.8 |
| Average | 75.6 | 47.3 | 39.2 | 18.4 | 422.4 | 43.5 | 468.5 | 55.3 |

Seeds:

```
# random.seed(40)
        # random.seed(41)
        # random.seed(42)
        # random.seed(43)
        # random.seed(44)
```

recurBackGenerator | recurBackMazeSolver – Total Cost: 56 + 72 + 77 + 122 + 45
recurBackGenerator | recurBackMazeSolver – Unique Cells: 36 + 60 + 47 + 68 + 40

primGenerator | recurBackMazeSolver – Total Cost: 88 + 15 + 101 + 53 + 40
primGenerator | recurBackMazeSolver – Unique Cost: 54 + 12 + 59 + 32 + 26

wilsonGenerator | recurBackMazeSolver – Total Cost: 125 + 75 + 117 + 55 + 93
wilsonGenerator | recurBackMazeSolver – Unique Cost: 70 + 45 + 68 + 37 + 55

---

recurBackGenerator | Dijkstra – Total Cost: 32 + 96 + 42 + 28 + 68
recurBackGenerator | Dijkstra – Unique Cost: 16 + 48 + 17 + 14 + 34

primGenerator | Dijkstra – Total Cost: 28 + 30 + 34 + 28 + 30
primGenerator | Dijkstra – Unique Cost: 14 + 14 + 14 + 14 + 15

wilsonGenerator | Dijkstra – Total Cost: 34 + 28 + 42 + 36 + 32
wilsonGenerator | Dijkstra – Unique Cost: 15 + 14 + 18 + 16 + 13

---

recurBackGenerator | Wall – Total Cost: 51 + 69 + 38 + 111 + 33
recurBackGenerator | Wall – Unique Cost: 19 + 58 + 27 + 62 + 33

primGenerator | Wall – Total Cost: 1014 + 1145 + 546 + 1888 + 20
primGenerator | Wall – Unique Cost: 77 + 63 + 56 + 76 + 14

wilsonGenerator | Wall – Total Cost: 84 + 574 + 719 + 24 + 20
wilsonGenerator | Wall – Unique Cost: 42 + 48 + 38 + 21 + 18

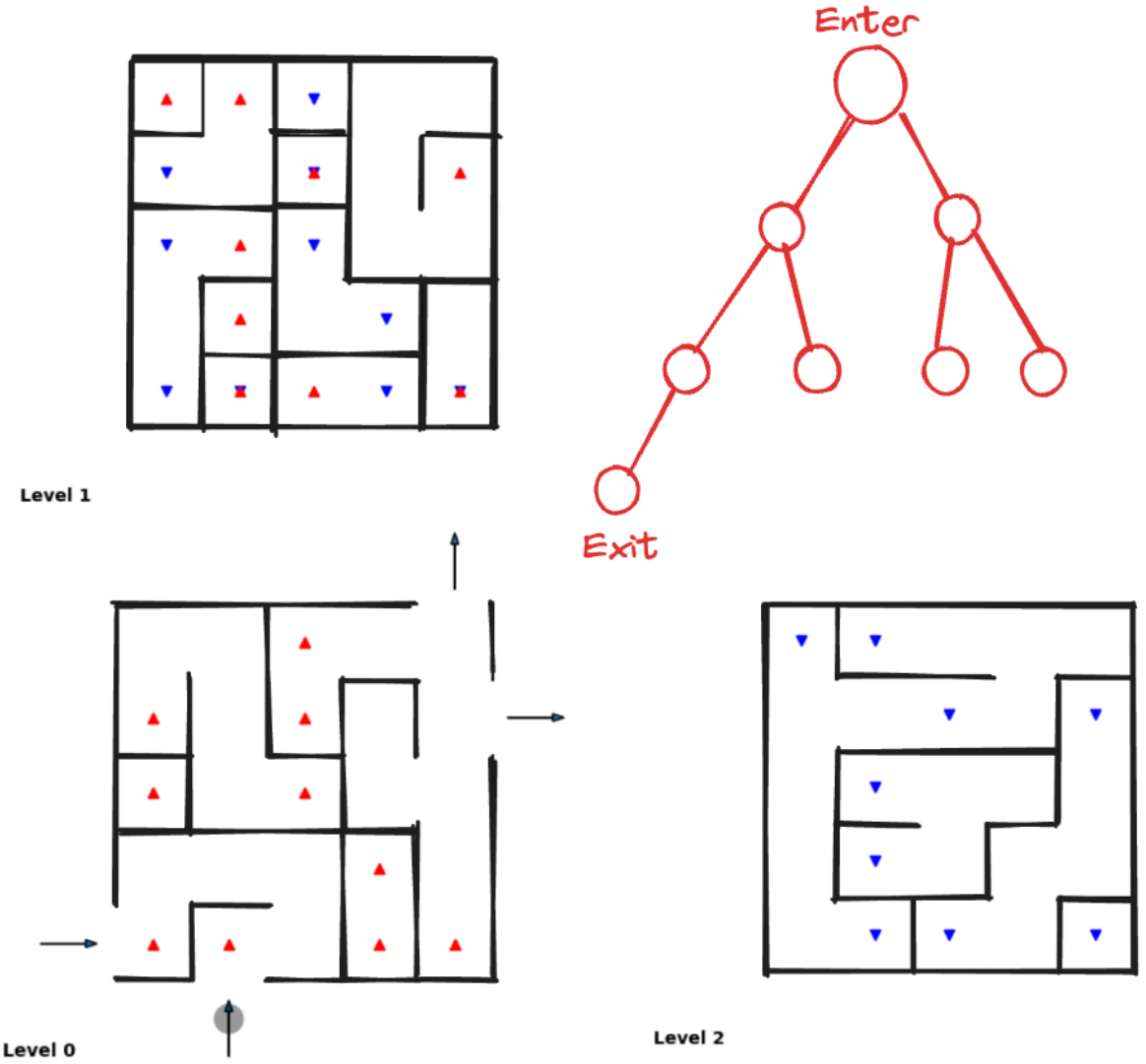
---

recurBackGenerator | Pledge – Total Cost: 431 + 48 + 116 + 203 + 30
recurBackGenerator | Pledge – Unique Cost: 76 + 48 + 67 + 134 + 13

primGenerator | Pledge – Total Cost: 95 + 3500 + 253 + 484 + 340
primGenerator | Pledge – Unique Cost: 35 + 70 + 62 + 49 + 57

wilsonGenerator | Pledge – Total Cost: 88 + 591 + 92 + 201 + 56
wilsonGenerator | Pledge – Unique Cost: 44 + 48 + 43 + 47 + 37

# 3D MAZE GENERATOR




*Mrwan Fahad A Alhandi*

*RMIT University | 21 June 2024*

## DFS & Wall Following Reasoning

To make a generator that maximize the cells explored for the solvers, we must consider their worst-case scenarios.

1. **Depth-First Search (DFS):**

The worst case for DFS involves extensive backtracking, forcing the algorithm to explore many wrong paths before finding the solution. This can be achieved by creating numerous dead ends and long winding paths that lead nowhere. For instance, in Figure 1, should the algorithm initially opt for path 3 instead of path 4, it would necessitate exploring all adjacent nodes, and it must backtrack. Making a generator that focus on making as much dead ends as possible will maximize the total cells explored. Moreover, as illustrated in Figure 2, the longer the path, the greater the amount of backtracking needed for the algorithm to discover a new, unvisited cell or path.

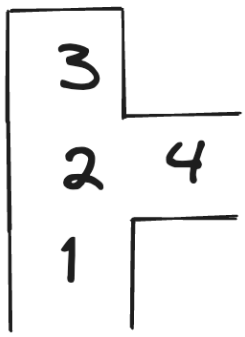



*Figure 4: Dead End Example*

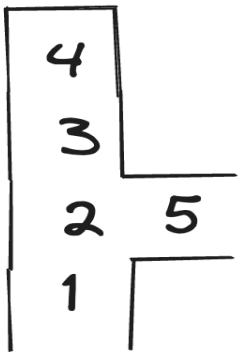



*Figure 5: Dead End + Long Path*

2. **Wall Following:**

The worst-case scenario for wall-following algorithms occurs when they make numerous turns before advancing through each cell, increasing the number of cells visited. This often results from enclosed cells, as shown in Figures 3 and 4, where the inclusion of such cells leads to more extensive exploration.

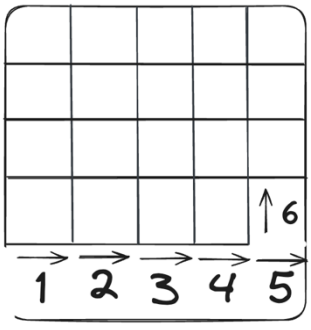



*Figure 7: No Enclosed Cell*

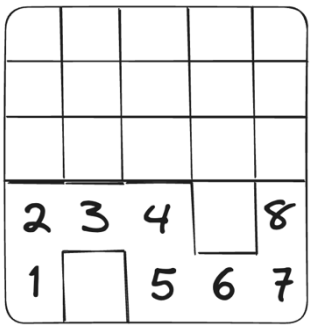



*Figure 6: Enclosed Cell*

# Strategy

1. **Generate the Initial Maze with DFS:**

Begin by generating a maze using the depth-first search (DFS) algorithm. The current DFS implementation in recurBackSolver selects random neighbors, which can sometimes lead to the removal of unnecessary walls. The primary goal of this step is simply to ensure that the maze has a solvable path, rather than to increase its complexity. To achieve this more efficiently, instead of choosing neighbors randomly, select the unvisited neighbor that is closest to the exit, either by row or column. This adjustment focuses on creating a clear, solvable path without unnecessarily complicating the maze's structure.

2. **Enhance Complexity:**

After the initial maze generation, enhance the complexity by strategically placing additional walls. This modification aims to create longer paths, which challenges the DFS algorithm during backtracking, thereby increasing the total number of cells explored. Additionally, to complicate matters for wall-following or Pledge algorithms, incorporate enclosed cells to compel more thorough exploration.

For extending paths, implement the Pledge algorithm by encouraging the generator to continue in its initially chosen direction as much as possible. When encountering a wall, remove it to maintain direction, prolonging the path further. If the generator switches to wall-following mode, refrain from removing any walls until it reverts to the Pledge approach. This strategy ensures a maze that not only challenges solvers by increasing the path length and complexity but also forces more extensive exploration through the incorporation of strategic barriers.

For Enclosed cells, we can also use wall following algorithm to each time turn the arrow and put a wall in the direction the arrow is facing.

3. **Maintain Solvability:**

Whenever we remove or add a wall when enhancing the complexity, we must ensure that the maze remains solvable after each modification by verifying the solvability using a reliable algorithm, such as Dijkstra’s or DFS. If a modification makes the maze unsolvable, revert the change.

The pseudocode for the algorithm is in the appendix for reference.

# Appendix

INPUT: Maze with initial paths

OUTPUT: Maze with enhanced complexity for solvers

1. INITIALIZE:
   - Set up the maze array with initial paths using DFS
   - Initialize variables for potential modifications

2. ENHANCE COMPLEXITY:
   For each cell in the maze:
   - Check if it's a dead-end
   - If true, potentially add walls to form enclosed cells
   - If using Pledge algorithm:
     Continue in initial direction as much as possible
     When hitting a wall, remove the wall if it leads to an unexplored area
   - If switched to wall-following mode:
     Do not remove walls until back to Pledge approach

3. VERIFY AND ADJUST:
   For each modification:
   - Ensure that there is still a solvable path using a path-checking algorithm
   - If a modification causes unsolvability, revert it

4. RETURN:
   - Return the enhanced maze