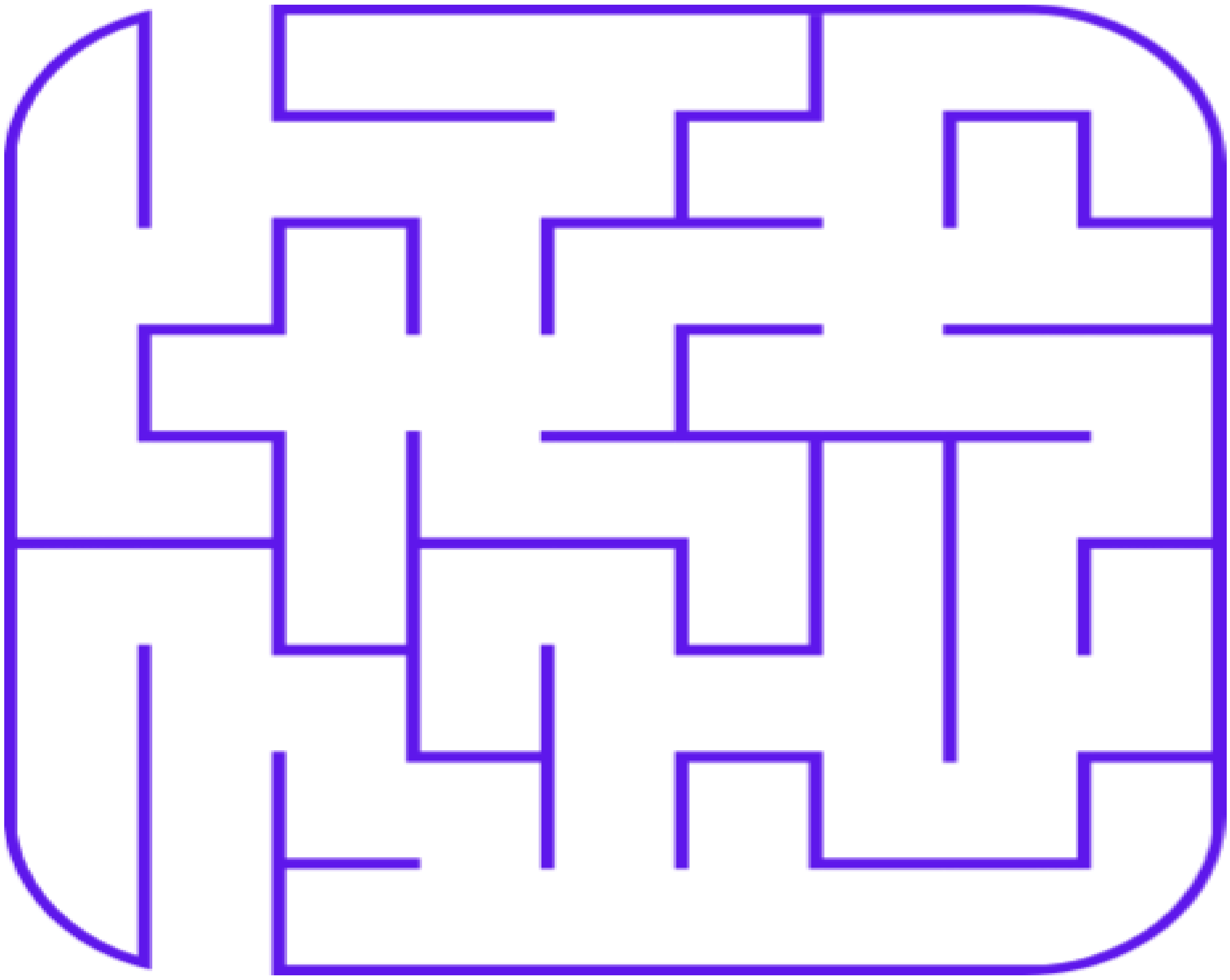
COSC2123

(UNDERGRADUATE)

EXPLORING MAZE GENERATION & SOLVING ALGORITHMS

OISIN SOL EMLYN AEONN

S3952320



FRIDAY 7 JUNE 2024

Table of Contents

Task C Report

Assumptions

Count refers to the number of ***unique cells explored***, not including repeated visits due to *looping* or *backtracking*, as confirmed by *Edward Small* and *Jeffrey Chan* in the ***EdForum;*** coinciding with the given *recur backtrack solver (using a DFS implementation).*

Introduction

In **T*ask C***, I aim to find the *shortest path* between a set of *entry-exit pairs* in a *perfect 3-D* maze while minimising the total number of *unique cells explored*. Given this challenge I need to find, and optimise a solver *s.t.*

min(cells explored() + distance(entrance, exit))

One major constraint I have is that the locations of the exits are unknown, being only provided with the number of exits present in the maze. This represents a challenge, as *heuristic path-finding algorithms* such as ***A***** or other traditional *greedy techniques* that rely on knowing where the destination or goals are to guide their searches efficiently are not allowed to be used.

My initial thoughts on how to minimise the optimisation function is that you are most likely best off with the first path to an exit you find unless you pass by another entrance. I don't think it is ever beneficial to further explore the maze in search of another exit as this would just needlessly increase the exploration total, while the potential to reduce the function's distance attribute is minimal. However, this does not necessarily mean that running the algorithm again if it is subject to randomness or using a different entrance won't reduce the number of unique cells you traversed in the maze.

There are also lots of factors that go into finding a solver that can solve a maze efficiently. For instance a *Wall Following Algorithm* may not work as effectively against some *Depth-First-Search* (*DFS)* generated mazes due to the long, and winding paths, but may work better against a *Minimum Spanning Tree (MST)* like *Prim's Algorithm as they have less depth*. To extend on the previously denoted optimisation the goal is to find an algorithm or function *s.t.* it minimises the optimisation function ∀ generators. In terms of Algorithms that I know to be relatively efficient we have the given *DFS* as well as my implemented *Wall Following* and *Pledge* which will be useful for empirically analysing if my new solver is more efficient.

Initial Thoughts

Thus, given these constraints I adopted more practical approaches, such as *Tremaux's Algorithm* or a *Biased DFS*. The rationale for adopting *Tremaux's Algorithm* is its ability to mark already explored paths, allowing for a more direct approach compared to *Wall Followers* or my *Pledge method*, which often follow longer and indirect routes. This makes *Tremaux's Algorithm* more suitable for efficiently navigating ,more complex mazes, with more depth.

On the other hand, none of the algorithms implemented thus far take into account their position and previous positions *(beyond not looping)*. Modifying a solver to make it b*iased such as the DFS implemented recur backtracking algorithm* could be highly effective, as I could make it favour exploring directions that have been less visited in the maze *(beyond direct neighbours)*. This approach resembles a *greedy algorithm* but remains within the rules and constraints. I believe this method may have a higher likelihood of finding an exit while exploring less unique cells in most cases.

To evaluate any solver it would be useful to additionally implement a *Shortest Path Algorithm* such as *Dijkstra's (see Appendix A)* or a *Multi-Sourced Breadth-First Search (BFS).* These will provide us with the actual shortest path between an entry-exit pair, allowing us to compare my solutions to the other solvers.

Theoretical Analysis

Algorithm	***Wall Follower / Pledge***	***Dijkstra***	***BFS***	***DFS***	***Biased DFS***	***Tremaux***
E (No. of Unique Cells Explored)	*Proportional to path length until exit.*	Every solve will explore all cells thus the best, average, and worst case ∈ *Θ(n), where n is the total number of cells in the maze.*	*Explores all cells level-by-level until an exit is found.*	*Typically, less than BFS; explores deeply along branches until exit.*	*Varies – worst case is if the exit is near an entrance.*	*Similar to DFS.*
D (Distance between Entry and Exit)	*Same as above.*	*Shortest path length.*	*Shortest path length.*	*Varies.*	*Varies.*	*Varies.*
Worst-Case Complexity	*∀ solvers there ∃ a maze s.t. the worst-case scenario is the last uniquely explored cell is the exit which ∈ Θ(n), where n is the total number of cells in the maze.*					

Approach

For both my implementations, I will start with a base case of finding paths to multiple exits from a single entrance. I plan to later implement a mechanism *s.t.* if my algorithm encounters another entrance while exploring and then finds an exit, this shorter path is added instead. This observation can be used as a final optimisation step to further minimise the overall cost.

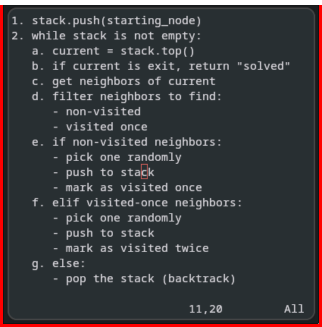
To confirm my findings, we will need to weigh and balance the principle of *explore vs. exploit.* I hypothesise that the best strategy will involve the algorithm exploring various parts of the maze and stopping immediately upon finding an exit.

Algorithms Description & Pseudocode

As previously introduced, I am selecting two different algorithms to analyse, implement and evaluate in an attempt to minimise exploration whilst still finding an efficient path. These were *Tremaux's* as well as a *Biased DFS Algorithm*. Below I will describe, and give a more in-depth description, as well as the *pseudocode* on how to implement these algorithms.

Algorithm	*Tremaux's Algorithm*	*Biased DFS*
Description	*Tremaux's Algorithm solves mazes through marking paths as visited to avoid loops and redundant visits. It tracks visited cells, and when all neighbours have been visited it backtracks to the last place where it has seen a unvisited cell. The algorithm stops upon finding an exit.*	*A standard DFS solver with a modified prioritisation system to bias exploration towards less-explored neighbours. It calculates weights based on visit counts and uses probabilities to select the next cell, making it more informed than a traditional DFS solver.*

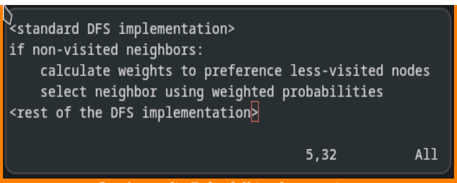
Pseudocode



```
1. stack.push(starting_node)
2. while stack is not empty:
   a. current = stack.top()
   b. if current is exit, return "solved"
   c. get neighbors of current
   d. filter neighbors to find:
      - non-visited
      - visited once
   e. if non-visited neighbors:
      - pick one randomly
      - push to stack
      - mark as visited once
   f. elif visited-once neighbors:
      - pick one randomly
      - push to stack
      - mark as visited twice
   g. else:
      - pop the stack (backtrack)

                              11,20         All
```

See Appendix C for full implementation.



```
<standard DFS implementation>
if non-visited neighbors:
    calculate weights to preference less-visited nodes
    select neighbor using weighted probabilities
<rest of the DFS implementation>

                                   5,32          All
```

See Appendix D for full implementation.

From my experience with implementing from the Wall Following and Pledge Algorithms, I've learnt that these algorithms often visit many cells multiple times when they reach a dead-end *(through backtracking or looping)*. To ensure a consistent evaluation, it's crucial that we do not double count cells. Instead, we should only count unique cells explored as we have previously defined.

Empirical Analysis

Data Generation & Experiment Setup

To generate data for my empirical analysis, I wrote a *Python Script (see Appendix I)* designed to generate and test *.json configuration* files for 4 variously sized mazes *(with static entrances & exits)*, each doubling in the total number of cells *(found by levels × columns × rows)* present in the maze. I will also test *(with less focus)* 4 statically sized mazes with an increasing number of random entries and exits *(placed at a minimum distance apart, and a preference for entrances being on the lower levels, and exits being at the top of the maze)*. This approach allows me to test a wide array of maze attributes to uncover the trends in the performance of each generator verses each solver while mitigating any bias. The maze attributes I tested are defined below:

- ***Square Maze range:*** *{ levels × rows × columns | $3 \leq l \leq 6, 5 \leq r \leq 10, 5 \leq c \leq 10$ } ∈ ℤ*
- ***Entry and Exit Maze range:*** *{ $3 \times 10 \times 10$ | $2 \leq$ entry / exits ≤ 5 } ∈ ℤ*

See Appendix G for all configuration files used in Task C for data generation.

To further mitigate bias and ensure quality data, I will employ the use of the random seed attribute. This will ensure that random numbers remain constant for generation and solving, affecting starting points and directions. Testing and averaging our tests multiple times is vital for consistent and comparable results as one generation may make it easy for one algorithm leading to biased results. Thus, I will test each combination three times *(with seeds 42, 44, and 46)*, running additional tests if I detect outliers to ensure accurate averages. Furthermore, the minimum number of entrances in any maze will be 2, and I will test each maze for all entrances present in the maze. **NOTE:** All cell counts will be rounded to the nearest cell when divided by the number of tests, while percentages and averages used solely for graphing plots will not be rounded.

Results & Discussion

I evaluated the maze-solving algorithms using the previously denoted criterion, focussing on their performance across different mazes generated by different algorithms.

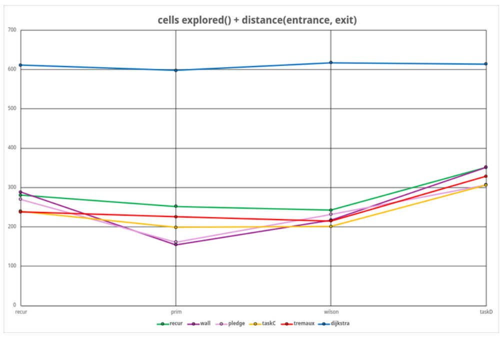


Figure 1: Unique Cells Explored+Distance(Entry, Exit)

Prim's Generated Mazes

Prim's Algorithm creates shallow mazes, which are well-suited for solvers like the *Wall Follower* and the *Pledge algorithm*. These solvers excel in *Prim's* mazes because the shallow structure mitigates their limitations. In contrast, using a *DFS-based solver* on these mazes is less efficient due to the lack of depth and complexity.

Complex Generated Mazes

My *Biased DFS* and *Tremaux's Algorithm* outperformed others on mazes generated by *DFS* and *Wilson's Algorithms*. These algorithms, with their complex, winding paths, match well with the strengths of biased DFS and Tremaux's methods. The ***Task D Generator*** *(spoiler)*, inspired by *DFS*, creates challenging mazes with long paths and minimal dead-ends, making them difficult for *Wall Follower* and *Pledge solvers*.

Conclusion

My ***Empirical Analysis*** demonstrated that *Biased DFS* and *Tremaux's Algorithms* explored fewer unique cells and found shorter paths in *DFS* and *Wilson generated mazes*, indicating their efficiency. However, these solvers underperformed in *Prim's generated mazes*, suggesting the need for *further optimisation*. Given the constraints I felt quite limited in what I was allowed to implement. For instance, I really wanted to implement a *Jump Point Search (JPS) Algorithm* that checks all sides of each level to find an exit. From there having found the goal, we could run a *heuristic greedy algorithm* to efficiently traverse the maze in the direction of the found exit. However, I deemed this approach to be *illegal* as it would mean solving the maze in a way that isn't feasible for someone without the ability to jump across the map. In addition to exploring and implementing the ***A**** *Algorithm*, even though it was not allowed *(see Appendix B)*, I also considered a *Limited-DFS Algorithm*, and *Biased Random Walk*, which could add an element of randomness to the exploration while still favouring certain directions. Additionally, finding an alternative entrance on a path to an exit and adding the path from this entrance to the exit instead of always using the starting entrance could further reduce traversal costs, but was difficult to get working in cases where the entrance was further away due to the mazes' generated path *(mainly due to perfect mazes).*

We also uncovered how the number of exits in a maze can make certain algorithms more efficient like a *Wall Follower* as it tends to explore close to the edge, avoiding deep exploration *(if moving in the right direction in a perfect maze)*. Another technique worth exploring involves moving as far away from the entrance as possible, hoping the exit is far from the entrance.

The development process presented several challenges, particularly with implementing *Tremaux's Algorithm* from scratch. Issues included ensuring the solver's smooth entry and exit from the maze. Further refinements and testing are necessary to address these challenges and optimise performance in complex maze structures.

In conclusion, *Biased DFS* and *Tremaux's Algorithm* are effective for *DFS* and *Wilson's mazes* but less so for *Prim's mazes*. *Wall Follower* and *Pledge Algorithms* are better suited for *shallow structures*. Further exploration, particularly in implementing mechanisms to detect and use shorter paths, is warranted to *optimise performance* in more *complex maze structures*.

Task D Report

Objective

In ***Task D***, we aim to design and implement a *3-D* maze generator s.t. for the set of all Mazes *M*, we maximise the sum of unique cells explored *E* by *the set of known solvers S and the unknown mystery solver u*. Mathematically, this problem can be denoted as:

$$f(S) = \max_{m \in M} (E(m)) \quad \text{where} \quad E(m) = u(m) + \sum_{s \in S} s(m)$$

Constraints

Initially, I considered generating non-perfect mazes with isolated parts (*islands*), which could render the maze unsolvable for a *Wall Following Algorithm* due to potential loops. I had already explored this type of maze generation to test my *Pledge Algorithm* against scenarios where it might be more efficient or impossible to solve without this additional logic. However, this approach was deemed unsuitable as it would not achieve the goal of maximising the optimisation function for the number of unique cells explored by a solver as well as it was explicitly prohibited. Therefore, the maze generation algorithm must ensure that the maze is always perfect, meaning it should form a tree without loops or islands, although it may contain *cul-de-sacs* (dead-ends). During the maze generation process, we have prior knowledge of the solver type, as well as the entry, and exit locations. The challenge posed by the mystery solver with an unknown strategy adds a layer of complexity, requiring the maze generation strategy to balance accommodating known solving strategies and accounting for the unpredictability of an unknown solver.

Theoretical Analysis

I hypothesise that different combinations of maze generators and solvers will significantly vary in the total number of uniquely explored cells. As maze dimensions *(levels, rows, columns)* increase, the total number of these cells will also increase. Additionally, more entry and exit points should decrease the number of uniquely explored cells by providing more pathways to a valid solution.

Aspect	***Prim's Algorithm***	***DFS (Depth-First Search)***	***Wilson's Algorithm***
Maze Structure	*Balanced, shallow, grid-like with shortest average paths*	*Long, winding paths, tree-like with the longest average paths*	*Highly complex, unstructured (random) with moderate path lengths*
Computational Complexity *(Given V=L×R×C and E≈3V)*	*Initialising a priority queue and processing each edge gives a worst-case complexity* $\in \Theta(V_{\log} V)$.	*Initialising and marking cells, and traversing each vertex and edge gives a worst-case complexity* $\in \Theta(V)$.	*Initializing the grid and marking cells, and random walks potentially involving V steps per cell gives a worst-case complexity* $\in \Theta(V^2)$.
Distribution of Paths	*Evenly distributed with many junctions.*	*Few junctions, long corridors*	*Random with intermediate junctions*
Ease of Solving	*Easier to solve for most solvers*	*Harder to solve, especially for non-exhaustive solvers like Wall Following or Pledge*	*Intermediate difficulty, exhaustive solvers like DFS will be able to solve easily*

Approach

I plan to conduct an empirical analysis to confirm my theoretical hypothesis about the inefficiencies of various maze generators when used against various solvers. We will first identify which generators are the least efficient for each solver. This analysis will ensure a thorough development of the ***Task D Generator***, which will amalgamate different algorithms through conditional statements to match each solver to its least efficient generator, ensuring extensive traversal.

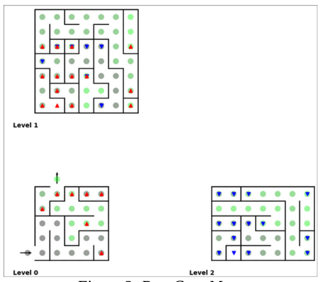


Figure 2: Best Case Maze

I also conducted extensive research into solving algorithms *(that could be the mystery solver)* and alternative maze generation methods, such as Hunt-and-Kill and Kruskal's algorithm. However, the primary focus I decided was on creating a *DFS-inspired generator*, hoping to further minimise dead-ends and emphasise long, winding paths. This approach will be particularly challenging for solving algorithms like the *Wall Follower* and *Pledge* as we've previously seen in ***Task C***. On the other hand by reducing the number of dead-ends, we also can hopefully increase maze coverage by exhaustive algorithms like *DFS*. This will effectively challenge the mystery solver *(whatever it might be)*, ensuring it traverses as much of the maze as possible to reach the solution. See Figure 2 for what we aim to design.

Additionally, I speculate that the potential mystery solver could be a direction-based *DFS* or a *Breadth-First Search (BFS)* algorithm, which will require optimising the generation technique against.

Empirical Analysis

Data Generation & Experiment Setup

The same data generation, experiment setup, and methodology detailed in ***Part C*** will be utilised, with the addition of analysing an extra generator using altered *.json configuration files* *(see Appendix H).*

Results Discussion

In my test results, I sought to measure two key metrics useful for evaluating each of the generator-solver combinations:

$$\text{Average Number of Unique Cells Explored} = \frac{1}{|T|}\sum_{t \in T} E(t) \quad \text{where} \quad E(t) \quad \text{is the number of unique cells explored in test} \quad t, \quad \text{and} \quad |T| \quad \text{is the total number of tests.}$$

$$\text{Average Maze Coverage} = \frac{\text{Average Number of Unique Cells Explored}}{\text{average(Total Maze Cell Count)}} \quad \text{where Total Maze Cell Count} = \text{levels} \times \text{rows} \times \text{columns}$$

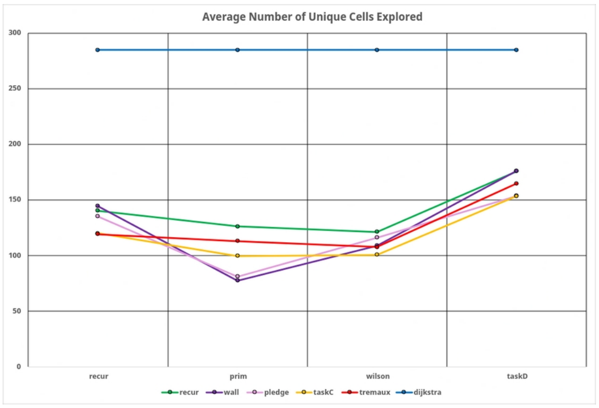


Figure 3: Total Unique Cells Explored

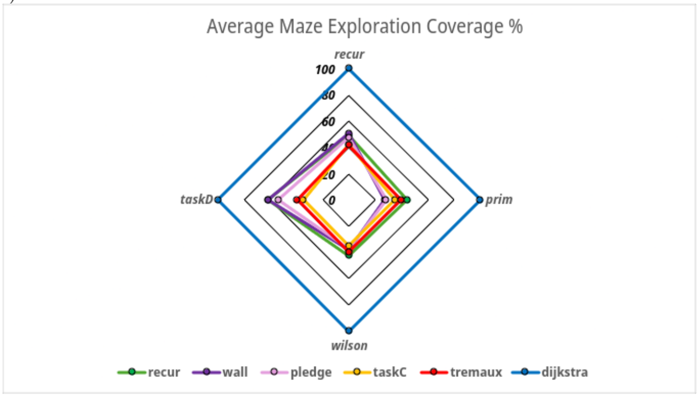


Figure 4: Average Maze Exploration Coverage Percentage

This *empirical analysis* examined the average number of unique cells explored, and derived the maze coverage percentage by various maze-solving algorithms across different maze generation methods. Each solver was tested with multiple, randomly placed entrances and exits to ensure unbiased results.

Looking at Figures 4 & 5 we can see the *DFS* solver showed moderate efficiency, performing best with *Wilson's mazes* (42.54%) winning first place in no category and struggling significantly with the ***Task D Generator*** (61.68%).

On the other hand the *Wall Following algorithm* excelled with *Prim's mazes* (27.13%), highlighting its efficiency in simpler structures, but it faced challenges with ***Task D*** (61.76%).
The *Pledge Algorithm* displayed similar patterns, being most efficient with *Prim's mazes* (28.36%) and least with ***Task D*** (53.91%).

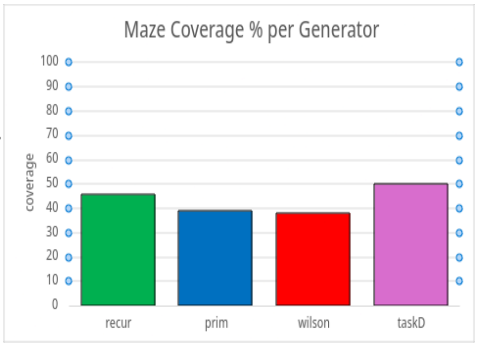


Figure 5: Maze Coverage % per Generator

Task C *Algorithm* consistently maintained *low exploration rates* across all maze generators (34.90%), validating its effectiveness. The *Tremaux Algorithm* also showed consistent performance, with lower exploration percentages (39.64%), making it relatively efficient.

As expected, *Dijkstra's Algorithm* explored the entire maze in every test (100%), ensuring the optimal path but at a high exploration cost. The standout result was the ***Task D Generator*** (as we had hoped), which increased average maze coverage by over 10% for Wilson & Prims, and almost 5% for DFS, suggesting it creates more challenging mazes for solvers.

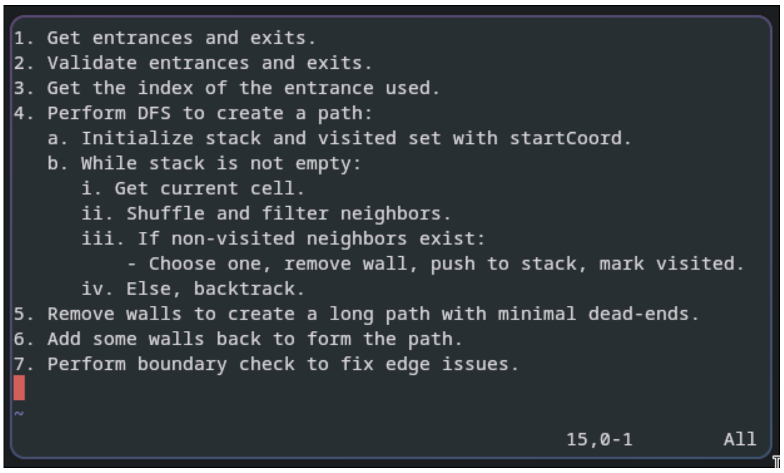
Overall, the findings underscore the importance of choosing appropriate maze generation techniques to match the desired challenge level for different solvers. *DFS* solvers generally explored more cells, while *Wall Following* and *Pledge Algorithms* were most efficient with *Prim's mazes*. I used these insights to guide further optimisation against maze-solving algorithms and the iterative development of the ***Task D Generator***.

Final Generator

Now that we've got some results for how each generator goes at slowing down each solver I can create my ***Task D Generator*** to further maximise solver exploration. At this point in time it was also unveiled the *Mystery Solver* would be a *random direction-based DFS Algorithm* which I tried to implement to further maximise my generator against as well as just generally *DFS*. I selected to create an iteration of *DFS* again as this generator type and approach already had the highest solver exploration. Proving to be inefficient to solve, particularly against other *DFS-based algorithms* which will be good for the *Mystery Solver*, as well as *Wall Followers*. The results confirmed that *DFS* implementations are the most inefficient for most solvers, reinforcing my hypothesis. Additionally, the generator uses some *greedy techniques*, considering the goals to create an inefficient path, further enhancing its effectiveness.

Pseudocode

From my results we can see that we do not need to actually use an amalgamation of algorithms as the least efficient to solve is across the board my ***Task D Generator***. I've provided my *pseudocode* as pictured in *Figure 6* for my general approach to creating such an algorithm, but please see Appendix E for the full implementation.



```
1. Get entrances and exits.
2. Validate entrances and exits.
3. Get the index of the entrance used.
4. Perform DFS to create a path:
   a. Initialize stack and visited set with startCoord.
   b. While stack is not empty:
      i. Get current cell.
      ii. Shuffle and filter neighbors.
      iii. If non-visited neighbors exist:
          - Choose one, remove wall, push to stack, mark visited.
      iv. Else, backtrack.
5. Remove walls to create a long path with minimal dead-ends.
6. Add some walls back to form the path.
7. Perform boundary check to fix edge issues.
~
                                                15,0-1        All
```

Figure 6: Pseudocode for Task D Generator (Modified DFS for single path)

Conclusion

The ***Task D Generator*** was developed with the objective of maximising solver exploration across a variety of maze-solving algorithms, including a *mystery solver*. Through an empirical analysis, and extensive development we identified the most and least efficient generators for each solver. The results demonstrated that the *DFS-based generator*, particularly modified version ***Task D Generator***, consistently increased the average maze coverage by over 10% for *Wilson & Prim's*, and almost 5% for *DFS*, creating more challenging mazes for solvers. This reinforced my hypothesis that *DFS* implementations are the most inefficient for most solvers, particularly against other *DFS-based algorithms* and *Wall Followers*. My ***Task D Generator*** employs a combination of randomised *DFS* traversal and *greedy techniques* to create long paths with minimal dead-ends, ensuring extensive traversal and higher inefficiency for solvers.

The success of the Task D Generator highlights the importance of strategically designing maze generators to balance the challenges for known solvers and the unpredictability of an unknown solver. The insights gained from this analysis will guide future optimisation of maze-solving algorithms and further iterative development of maze generation techniques. By ensuring that the generator maximises exploration and presents significant challenges, we achieve the goal of creating complex and engaging mazes that push the boundaries of current solving strategies.

Next time, I plan to run more tests, as using only *three seeds across 8 mazes, 4 generators, and 5 solvers* is insufficient to identify true trends. Additionally, I aim to explore more types of generators, including various *MST algorithms*, and incorporate additional solvers utilising *greedy* or *heuristic approaches*. I am also interested in exploring unique approaches such as J*ump Search* or Bi*-Directional Search (both the entrance and exit trying to find a connection)*.

Please look into the Appendices for further analysis and detailed insights.

References APA 7th Edition

Al Sweigart (2022) https://inventwithpython.com/recursion/chapter11.html accessed April 2024

Ayu Crystal (2022) https://stackoverflow.com/questions/70755207/how-create-an-adjacency-matrix-of-a-maze-graph accessed April 2024

BJFoutz (2021) https://github.com/bjfoutz00/3D-Maze-Solver accessed June 2024

GroverIsOP (2022) https://www.reddit.com/r/javahelp/comments/yd9xju/random_maze_generator_withh_adjacent_lists_help/ accessed April 2024

Hybesis Hurna (2020) https://medium.com/swlh/handling-graphs-with-adjacency-lists-7e1befa93285 accessed April 2024

Jamis Buck (2010) https://gist.github.com/jamis/761534 accessed April 2024

Jeff Ericson (2016) https://courses.engr.illinois.edu/cs374/fa2018/notes/05-graphs.pdf accessed April 2024

Larson (2013) https://users.csc.calpoly.edu/~zwood/teaching/csc471/final13/lplarson/ accessed June 2024

Levitin Introduction to the Design and Analysis of Algorithms (3rd ed. 2011) accessed April 2024. https://raw.githubusercontent.com/oguzaktas/textbooks/master/Anany%20Levitin%20-%20Introduction%20to%20the%20Design%20and%20Analysis%20of%20Algorithms%20(3rd%20Edition)%20(2012)%20(Pearson).pdf

Lukasz Bienias et al. (2016) https://www.researchgate.net/figure/Mapping-the-maze-into-an-array_fig9_315969093 accessed April 2024

Professor Azadeh Alavi & Dr Pubudu Sanjeewani (2024) RMIT University COSC2673

Professor Jeffrey Chan & Edward Small (PhD) (2024) RMIT University COSC2123

Professor Marc Demange & Dr David Ellison (2023) RMIT University MATH1150

Suzana Markovic (2019) https://unitech-selectedpapers.tugab.bg/images/papers/2019/s5/s5_p72.pdf accessed April 2024

Tikhon Jelvis (2024) https://jelv.is/blog/Generating-Mazes-with-Inductive-Graphs/ accessed April 2024

Tim Oyebode (2022) https://medium.com/@teemarveel/procedural-maze-generation-using-binary-tree-algorithm-5f17cb52a221 accessed April 2024
Walt (2020) https://medium.com/analytics-vidhya/how-to-solve-a-3d-maze-9f0c8f2cf24 accessed June 2024

Whymarrh (2012) https://stackoverflow.com/questions/4551575/data-structure-to-represent-a-maze accessed April 2024

Other: https://puzzling.stackexchange.com/questions/105414/strategy-for-solving-a-3d-maze, https://discourse.threejs.org/t/maze-algorithm-for-solving-difficult-2d-and-3d-mazes/41264, https://www.youtube.com/watch?v=QtfAazQJgJI

Appendix

For further analysis into my ***Parts A-D*** please see the following Appendices:

Appendix A: solving/dijkstraMazeSolver.py

Appendix B: solving/aStarMazeSolver.py

Appendix C: solving/tremauxMazeSolver.py

Appendix D: solving/taskCMazeSolver.py

Appendix E: generation/taskDMazeGenerator.py

Appendix F: EmpiricalData.xlsx

Appendix G: testConfigs/taskC/*

Appendix H: testConfigs/taskD/*

Appendix I: /testConfigs/testGeneration.py

This Project was completed using **GitHub** *(private repo);* email for an invitation.