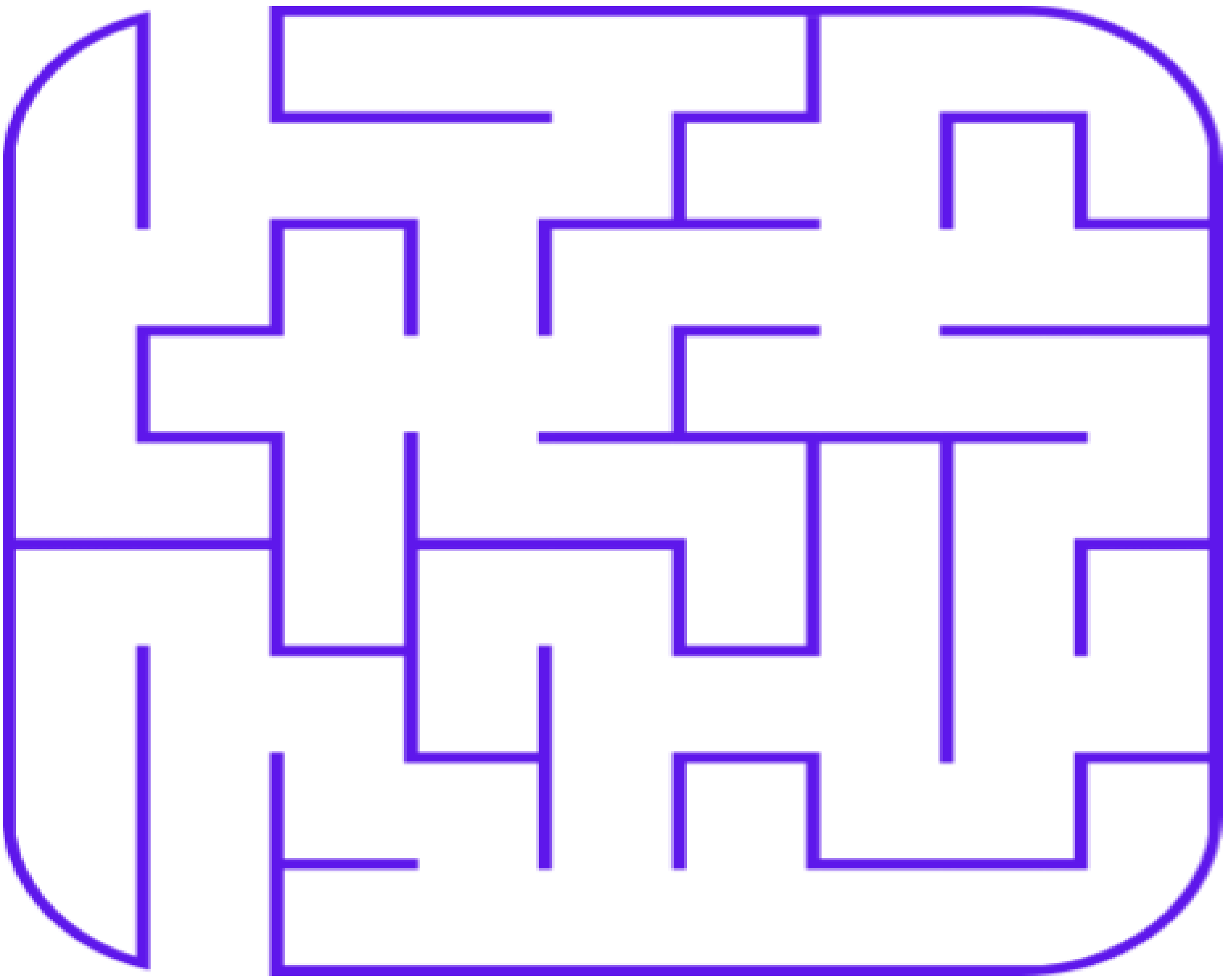
COSC2123
(UNDERGRADUATE)

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 visited, not including repeated visits due to looping or backtracking, as confirmed by Ed and Jeff in the EdForum. This coincides with the given DFS Solver implementation.

Introduction

In the task of finding the shortest path between a set of entrance and an exit pairs in a perfect 3-D maze while minimising the total number of unique cells explored, we are presented with the challenge of finding a solver s.t. *min(cells explored() + distance(entrance, exit)).* The locations of the exits are unknown, and we are only provided with the number of exits present in the maze. This represents a challenge, as traditional path-finding algorithms such as A* rely on knowing the destination in order to guide their search efficiently. Thus, we will adopt more practical approaches like Multi-Sourced Breadth-First Search (BFS) and an adapted version of Dijkstra's Algorithm. The rationale for adopting these strategies is that they can efficiently explore the maze to discover all exits, and subsequently find the shortest path between each entrance-exit pair while minimising the overall cost.

Theoretical Analysis

NOT MINE: To clarify, when you find a solution (a path from an exit to an entrance) using BFS you are guaranteed to have found the shortest path to the closest exit. However, the number of cells you must visit to do this may be catestrophically large! Remember, we want to minimise:

min{E+D(Ce,Cx)}

where E is the number of cells we explore, and D(Ce,Cx) is the distance between the entrance we start at and the exit we find. So, BFS will find the minimum D(Ce,Cx), but at the cost of E being huge. I think for a maze of no walls, you'd get something like:

E=i=1∑N(N+1−i)=2N2+N

(or similar - I'd need to double check this) where N is the number of cells in the maze. And so, for our 4x4 maze, we'd have to explore 136 cells (so E=136) and we would get the shortest path of 8 cells to the exit (D(Ce,Cx)=8) giving an answer of 144.

If we used the wall following algorithm, we get E=8 and D(Ce,Cx)=8, giving us an answer of 16. So this is the trade-off. If we think about this critically, the absolute worst a a discovered path could be is that it goes through every cell. So:

max{D(Ce,Cx)}=N

The absolute worst that a search could be is using BFS where we have a maze of no walls and thus must continually visit every cell and backtrack, which is given by E. Since our search has the potentially to be a much worse than the path, it should probably be prioritised? Have you ever heard of Trémaux's algorithm? Might be worth a look!

Hypothesis

from my Theoretical Analysis in Part C we can see the Explored Cells, and Distance from the first, and optimal set of entry and exits.

We can in combination with our generators theoretical analysis now hypothesise which combination of generators and solvers will be best to maximise our function.

Given the formula and the constraint of not knowing the exit locations, I hypothesise that a two-phase approach may yield promising results. In the first phase, we employ an exploration strategy to discover all the exits in the maze efficiently. Once all exits have been identified, we can transition to the second phase, where we will exploit to find the shortest path. By combining the results from both phases, we can determine the optimal entrance-exit pair that minimises the overall cost, which is the sum of the number of unique cells explored and the distance between the entrance and the exit.

I explored using Dijkstra, and a Multi-sourced BFS algorithm to confirm my findings. My initial thoughts were that it would be very expensive to try to find another solution to the maze regardless of the algorithm used thus we should always use the first path found as if you were to try again you would be starting having already explored, adding a constant to the cells explored. It was confirmed by Ed and Jeff that the cells explored was unique cells thus meaning backtracking cells or algorithms that explore mazes like pouring water in the maze should be quite effective.

Strategy

I will start with a base case of finding paths to multiple exits from a single entrance. This will involve exploring the maze using a Multi-Sourced BFS or an adapted version of Dijkstra's Algorithm until all exits are discovered. Upon finding an exit, the algorithm can either continue exploring to locate additional exits or transition to the second phase, where it finds the shortest path from that exit to each entrance using Dijkstra's Algorithm or A* (with the exit as the goal).

If the algorithm encounters another entrance while exploring or finding the shortest path to an exit, it can be assumed that there may be a shorter path if that entrance can connect to the current path. This observation can be used as a final optimisation step to further minimise the overall cost.

After exploring all possible paths, the algorithm will evaluate the cost (cells explored + distance from entrance to exit) for each entrance-exit pair and select the pair with the minimum cost as the optimal solution.

Pseudocode

Rationale / Reasoning

Variables

The main factors influencing the given formula outside of the solver are the maze dimensions (levels, columns, rows), number of entrances & exits as well as a small factor in the generating algorithm.

Theoretical Analysis

Justify – talk about explore vs exploit. I hypothesis that the best strategy for this will be on in which the algorithm explores various parts of the maze, and upon finding an exit stops. Count number of unique cells explored, number of cells of the shortest path, and graph it. Mention bigO notation of algorithms chose.

Data Generation

Experiment Setup

Empirical Analysis

I explored several algorithms; conducting an empirical analysis to confirm my hypothesis.

Results Discussion

Task D

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 *our set of known solvers S and our 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 our goal of maximising the optimisation function for the number of unique cells explored by a solver as well as it was explicitly prohibited. Therefore, our 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 our 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 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 ∈ $\Theta(V_{log}V)$.	Initialising and marking cells, and traversing each vertex and edge gives a worst-case complexity ∈ $\Theta(V)$.	Initializing the grid and marking cells, and random walks potentially involving V steps per cell gives a worst-case complexity ∈ $\Theta(V^2)$.
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 *TaskD Generator*, which will amalgamate different algorithms through conditional statements to match each solver to its least efficient generator, ensuring extensive traversal.

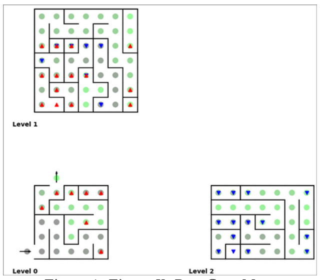


Figure 1: Figure X: Best Case Maze

I also conducted extensive research into solving algorithms and alternative maze generation methods, such as Hunt-and-Kill and Kruskal's algorithm. However, the primary focus is on creating a *DFS-inspired generator*, which minimises dead-ends and emphasises long, winding paths. This approach will be particularly challenging for solving algorithms like the Wall Follower and Pledge. On the other hand by reducing the number of dead-ends, we can hopefully increase maze coverage by exhaustive algorithms like *DFS*. This will effectively challenge the mystery solver, ensuring it traverses as much of the maze as possible to reach the solution.

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

To generate data for my empirical analysis, I wrote a *Python Script* (*see Appendix X*) 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 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 ≤ l ≤ 6, 5 ≤ r ≤ 10, 5 ≤ c ≤ 10 } ∈ ℤ*
- ***Entry and Exit Maze range:*** *{ 3 × 10 × 10 ∣ 2 ≤ entry / exits ≤ 5 } ∈ ℤ*

See Appendix X for all configuration files used 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 we will test each maze for all entrances present in the maze.

Results Discussion

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

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

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

Final Generator

Pseudocode

Our final generator (

In the generator query the solver name used in the .json file, if found different generator is better at creating difficult mazes.

Conditional

Make a generator that creates lots of corridors. Could make a generator that moves away from the exit. Mention that non-perfect mazes are not laberinths (e.g. contain islands). This would make it not only difficult but impossible for the wall following solver.

I observed that often if one entrance performed well the other one on average would perform poorly, as well as if they performed around the estimated average the other one was likely to perform similarly. Percentage of solver coverage was interesting for evaluating.

Benchmark all the generators vs each solver as well as the new generator.

Diagram of my intent in creating a maze. So this is a maze in which

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

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

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 & Ed 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

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

Appendix

For further analysis into the running time for the Array, Adjacency List, and Adjacency Matrix implementations please refer to the following Appendices:

Appendix 1: theoreticalAnalysis.pdf

Appendix 2: edaNotebook.ipynb

Appendix 3: computerSpecs.png

Appendix 4: configGenerator.py

Appendix 5: dataGen/Configs/*

Appendix 6: EmpiricalData.xlsx

Appendix 7: adjListGraph.py

Appendix 8: adjMatGraph.py

Appendix 9: /dataGen/timedClasses/*

This Project was completed using GitHub. See: https://github.com/des1-gner/Assign1-s3952320