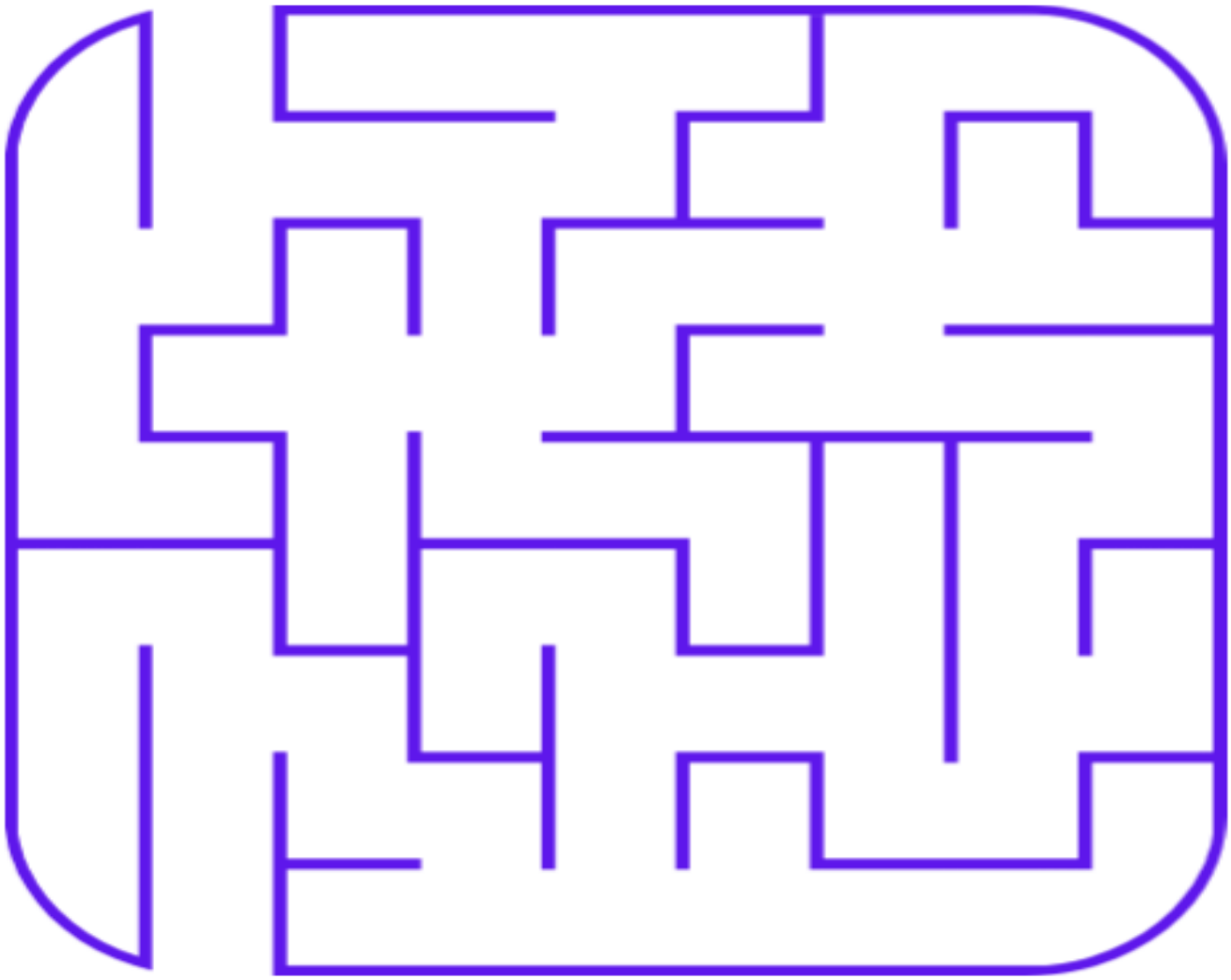
COSC2123

(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.

Hypothesis

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 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 unique cells explored by a solver and was strictly forbidden. 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, entrances, and exit locations. However, we are unable to run the solver before generating the maze, and there is an additional challenge posed by a mystery solver with an unknown strategy. This adds a layer of complexity, requiring our maze generation strategy to balance accommodating known solving strategies and accounting for the unpredictability of the mystery solver, emphasising the need for a robust approach effective for both known and unknown solvers.

Hypothesis

Different maze generators and solvers will significantly vary in the total number of uniquely explored cells. *Minimum Spanning Tree (MST) Algorithms* like *Prim's* generate balanced, shallow mazes that may be easier to solve. *Wilson's Algorithm* produces highly complex, unstructured mazes, posing a challenge for solvers. *Depth-First Search (DFS) Algorithms* create long, winding paths that could be more difficult for non-exhaustive search solvers. Among solvers, I expect DFS to explore the most unique cells, especially in complex mazes. The Wall Following, and Pledge Algorithm may struggle with DFS-generated mazes due to long paths or Wilson's algorithm's complexity with Pledge's dependent on the entry direction.

As maze dimensions (levels, rows, columns) increase, I expect explored cells to go up significantly. Additionally, more entry and exit points should decrease the maze coverage time by providing more completion options.

Potential mystery solvers might include direction-based DFS, likely to explore the most unique cells, and Breadth-First Search (BFS), which performs well in unique cell exploration by exploring nodes level by level.

Theoretical Analysis
move some from above down here.
Approach

My approach involves first conducting an empirical analysis to confirm my theoretical hypothesis of various maze generators and solvers to identify the least efficient generator for each solver. I will then depending on my results have a go at creating a custom generator that causes all solvers to traverse a larger coverage of the maze as well as writing conditional statements that match each solver to its least efficient generator, I aim to maximise the number of unique cells explored. For the mystery solver, I will design a perfect maze with minimal dead-ends and long passageways, ensuring that the solver has to explore nearly the entire maze to find the solution. For a starting generator I will attempt to create an

My strategy will be to first do an empirical analysis to confirm my hypothesis, and theoretical analysis of all of my generators, and which solver they will be best at maximizing the number of unique cells explored. My strategy will be since we can know the solver when it is called we can simply write a conditional set of conditional statements that go if solver = "solverX" use generatorX which is the least efficient maze generator type for said solver.

To combat the Mystery Solver on the other hand I did some research into the possible algorithms this could use, as well as some possible generators in general, and particular ones we can use to stop some.

I considered other generation algorithms like hunt-and-kill, as well as Krushkal, but settled on a combination, as well as a general difficult one. I hypothesis that the mystery solver could be:

My approach is to create such a maze in which there is only a single solution (e.g. a perfect maze), but which has very little or no dead-ends. Thus, meaning that in

order for a solver to get to the end it should need to explore all or near-all of the maze. This is achieved through long passage ways like in the pictured maze below. Our new generator should:
design a maze that is perfect, with only a single path (walk) with minimal (or no) deadends,

Variables

maybe put this in experiment setup or data generation.

When trying to create a maze generation algorithm that maximizes the number of unique cells explored by a given solver there are lots of different factors that may influence this. Obviously there is the number of entrances, and exits present in the maze, as well as their distance, and the shortest path between a given pair, and of course the maze size (level, and rows, and columns). To combat this I created 2 empirical tests to test

Most importantly in our tests we must ensure fairness, and equity across all of our algorithms by using a random seed generator. Programming Languages / Computers when generating dynamic mazes or solving them use lots of random numbers like starting point for a generator, as well as direction. These all can impact / factor into the number of cells it takes to solve a maze. Say one algorithm decides to make a direct path from entry to exit on the right hand wall, and a right-handed wall following algorithm is used we will see that it solves straight away! To get around this we must use a random seed generator, which luckily is provided in the skeleton code. This allows us to test the same generated maze across multiple solvers. This makes our results much more comprable.

Data Generation

In answering this question throroughly we will first explore all previously implemented maze generated against each, and all implemented solvers.

size

entry and exit test

Experiment Setup

During empirical analysis I sought to test various of these conditions to ensure that we create a general solution that given a config file it will still create a very difficult to solve maze requiring the solver to go to the maximum amount of unique cells. This would include various maze sizes, various maze entry / exits. I used the seed to ensure when testing across various different generators, and solvers that the maze remained the same requiring us to never have to re-test a single solve as they should always be accurate unlike in assignment 1 where we were measuring time which is subject to far more conditions beyond control.

I sought to test Maze Coverage
and the total number of unique cells explored

Empirical Analysis

Results Discussion

Having uncovered some trends behind the relationship each solver had with generator the recurback generator or DFS proves to be the most effective at increasing the Average Number of Cells Explored, and Average Maze Coverage this is important going forward as it means

In this section we will discuss, and calculate the

Average Number of Cells Explored = average(sum of E/|tests|)

Average Maze Coverage = average(averageNumberofCellsExpored/average(totalMazeCellCount))

mention that average maze coverage is a better metric as it is not subject to the influence of the dimensions of the maze.

For configuration of the best maze (one in which further optimizes solvers to explore the maximum number of unique cells we hope for:
spans multiple floors, and has only a single entry / exit. And large dimensions.

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.

Data Generation

For my empirical analysis, I generated 3-D mazes of at least three levels across three different types. Each maze type includes configurations with a single entrance and exit, multiple randomly placed entrances and exits, and varying sizes. This approach ensures a comprehensive evaluation of maze generation performance across diverse scenarios.

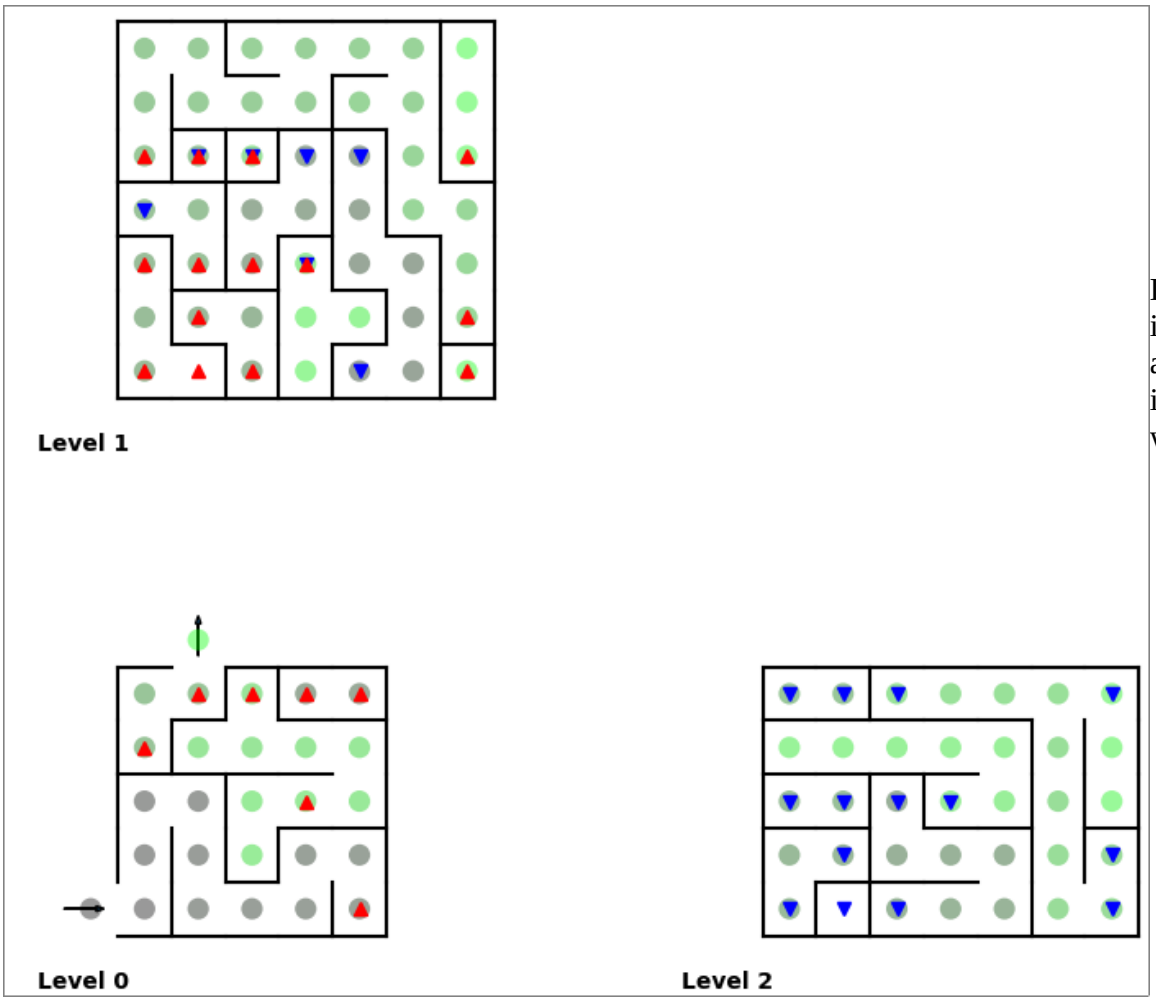


Diagram of my intent in creating a maze. So this is a maze in which

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