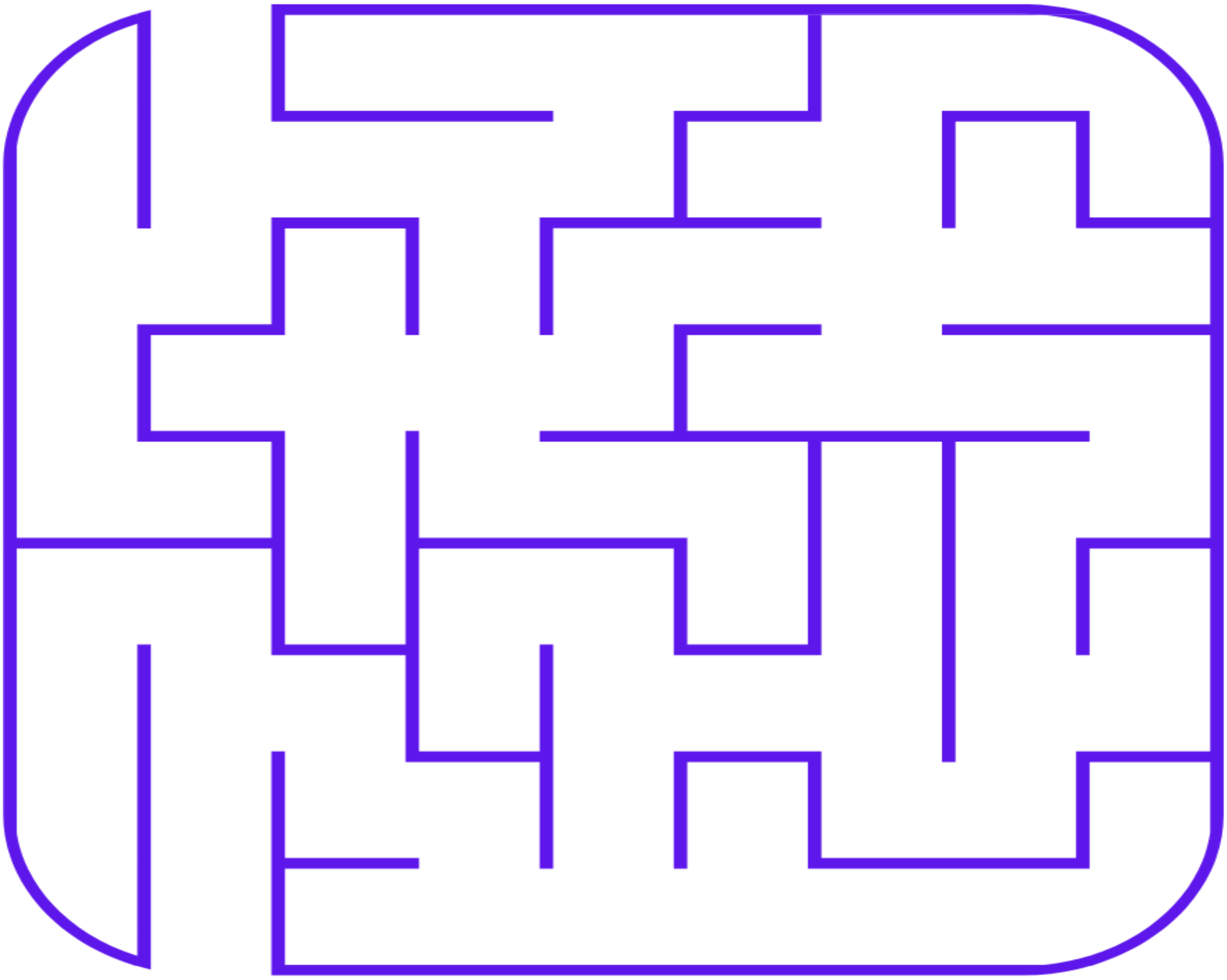
COSC2123

IMPLEMENTING & EVALUATING DATA STRUCTURES FOR MAZE GENERATION

OISIN SOL EMLYN AEONN

S3952320



FRIDAY 19TH APRIL 2024

This Page was intentionally left blank.

Implementing & Evaluating Data Structures for Maze Generation

Published April 19, 2024

COSC2123 Algorithms & Analysis RMIT University Canvas

Oisin Sol Emlyn Aeonn

Student ID: 3952320

All Rights Reserved RMIT University

X ____________________

For Edward Small, Jeffrey Chan,

& RMIT University.

Table of Contents

Glossary

Abstract Data Type (ADT): Mathematical structure concerning operations, and behaviour without explicit implementation.

Algorithm: Sequence of (unambiguous) instructions for solving a problem. (Levitin 2011).

Analysis: Process of examining, and evaluating data to gain insights.

Data Structure: Organisation, and storage of data in computers, so it can be efficiently accessed, and modified.

Dictionary: Data structure utilising key-value pairs. Allows efficient lookup, insertion, and deletion.

Empirical: Observations made on real data to confirm hypotheses, and verify predictions.

Monte Carlo Method: Random sampling, and verification of data to ensure robustness, and withstand potential bias.

Theoretical: Thinking based on the principles of the subject, and hypotheses of what you predict will be real.

Vertex/Node: Unit in a graph representing a cell in the maze.

Running-time: How long it takes for an algorithm to complete its execution.

Uncertainty Quantification (UQ): Use of statistics, and probability to characterise, and quantify uncertainty so it can be mitigated.

Array: Data Structure that stores a fixed-size set of elements of the same type.

Maze: A network of paths designed to be tricky to navigate from entrance to exit. Can be represented using a Graph.

Edge: A connection between two vertices.

Adjacency List: A collection of unordered lists used to represent a graph, where each list describes the set of neighbours of any particular vertex.

Variance: Measure of how far a set of numbers are spread out.

Adjacency Matrix: A matrix used to represent a graph, where each element represents whether there is an edge between two vertices.

Wall: A barrier or obstacle that separates two vertices.

Symbols

∈ denotes "is an element of" in set theory.
≈ denotes an approximation between two near-equivalent values.
ℤ denotes the set of integers e.g. {0,1,2,3}.

Σ denotes the sum of.
Θ() denotes Big-O Notation for the exact complexity.
O∩Ω=Θ (Ed Small 2024)

Abstract

This project explores the theoretical, and empirical analysis of arrays, adjacency matrices, and lists in the context of maze generation. Specifically, implementing an adjacency matrix, and list from scratch and evaluating the expected verses practical running-time by generating various size mazes, and recording the time of the class, and individual functions.

The goal of this project is to learn about the different data structures, as well as theoretically and empirically analyse algorithms to solve problems more efficiently. By doing so we will complete Assessment 1 worth 20% in **COSC2123 Algorithms & Analysis**.

Research Methodology

This project will be implemented using Python 3.6.8 to run on the RMIT Teaching Servers. We will then set up the experiment by writing python scripts to generate numerous maze sizes, then record, and create charts for each data structure.

To evaluate the performance of the data structures for maze generation, the built-in timer was used as well as using external python libraries to measure the running-time, and number of calls per function. This allows for the collection of empirical data to compare the expected theoretical running-time with the actual practical running-time.

Significance Statement

This project is significant as it fulfils the requirements for Assessment 1 in the **COSC2123 Algorithms & Analysis** course. The project also provides an opportunity to learn, and develop skills in theoretically analysing algorithm complexity, gathering empirical data, and evaluating algorithms using both theoretical and practical approaches. These skills are crucial for designing efficient solutions to computational problems.

Introduction

In this report, we explore maze generation algorithms using different data structures, in particular arrays, adjacency lists, and adjacency matrices. We will theorise their theoretical time complexities and then test our hypotheses through gathering data on their running times, and empirically analysing the results. By evaluating the results and conducting a rigorous empirical analysis, we aim to understand the performance characteristics of each data structure in the domain of maze generation, and graph theory. My goal at the end of this report is to provide recommendations based on my findings, considering efficiency, simplicity, and scalability, to ensure you can make informed decisions when implementing maze generation algorithms.

Theoretical Analysis

In this theoretical analysis, we will explore the best and worst case time complexities by evaluating the code implementations of the neighbours() and updateWall() functions, as well as the full data structure implementation. We will determine the Big-O notation for each case. The following tables present the theoretical best, and worst case time complexities for these algorithms and data structures.

NOTE: The colour scheme in this table is used for all subsequent Graphs. See Appendix 1 for complete tables on the theoretical analysis of all functions.

Array

Operations	Worst Case	Best Case
neighbours()	checkCoordinates() is run first which creates an empty list and, adds corresponding neighbour cells to the list. These operations are $\in \Theta(1)$, meaning that neighbours() is $\Theta(1)$ for the worst case.	checkCoordinates() is run first which creates an empty list and, adds corresponding neighbour cells to the list. These operations are $\in \Theta(1)$, meaning that neighbours() is $\Theta(1)$ for the best case.
arrayMaze	init() initialises a 2-D list of size $(2 \times n + 2) \times (2 \times k + 2)$ and fills it with boolean values. This requires iterating over each cell in the 2-D grid, resulting in a time complexity $\in \Theta(n \times k)$. All other methods in the ArrayMaze class perform operations in constant time $\Theta(1)$, and thus aren't relevant as they aren't the dominant term.	init() initialises a 2-D list of size $(2 \times n + 2) \times (2 \times k + 2)$ and fills it with boolean values. This requires iterating over each cell in the 2-D grid, resulting in a time complexity $\in \Theta(n \times k)$. All other methods in the ArrayMaze class perform operations in constant time $\Theta(1)$, and thus aren't relevant as they aren't the dominant term.

Figure 1.1: Time Complexity for Array-Based Maze Implementation

Adjacency List

Operations	Worst Case	Best Case
updateWall()	updateWall() scans for the edge linking the two vertices in the adjacency list. In the worst case, the edge is at the end or $\notin$ adjacency list. This requires iterating through all the neighbours of both vertices, which can be up to $\Theta(n)$ in a dense graph. Therefore, the worst case time complexity of updateWall() $\in \Theta(n)$.	updateWall() checks if both vertices $\in$ adjacency list. In the best case, the edge connecting the two vertices is found immediately in the adjacency lists. This implies the method only needs to iterate through a single neighbour. All other operations in the updateWall(), are performed in constant time. Therefore, the best case time complexity of updateWall() $\in \Theta(1)$.
neighbours()	Since in the maze implementation a maximum number of 4 neighbours can be present, these would be returned, which is still $\in \Theta(1)$.	The best case occurs when the coordinate $\notin$ adjList{}, thus it just returns an empty list $\in \Theta(1)$.
adjListGraph	addVertex() needs to check if the vertex $\in$ adjList{} before adding it which requires the traversal of the entire adjList{}. $\forall$ other methods in the adjListGraph Class $\in \Theta(1)$, thus their time complexities are not relevant because the addVertex() is the dominant term. Therefore, adjListGraph $\in \Theta(n)$ in the worst case.	addVertex() adds the vertex to adjList{} in constant time $\Theta(1)$ when the vertex $\notin$ adjList{}. $\forall$ other methods in the adjListGraph Class $\in \Theta(1)$, thus their time complexities are not relevant because the addVertex() is the dominant term. Therefore, adjListGraph $\in \Theta(1)$ in the best case.

Figure 1.2: Time Complexity for Adjacency List Graph Maze

Adjacency Matrix

Operations	Worst Case	Best Case
updateWall()	If either or both coordinates $\notin$ vertex_indices{}, then the function returns False $\in \Theta(1)$.	When both coordinates $\in$ vertex_indices{} as it simply updates the wall status in the adjacency matrix and returns True $\in \Theta(1)$.
neighbours()	neighbours() is $\Theta(n)$ as it needs to iterate through the corresponding row in the adjacency matrix to find all the neighbours $\in \Theta(n)$.	$O(1)$ occurs when the coordinate $\notin$ vertex_indices{}, thus it just returns an empty list $\in \Theta(1)$.
adjMatGraph	init() in the worst case needs to initialize an adjMatrix[] where rowNum = colNum $\in \Theta(n^2)$. addVertex() $\in \Theta(n^2)$ as it needs to expand the matrix by adding new rows and columns. $\forall$ other methods in the adjMatGraph Class $\in \Theta(1)$ or $\Theta(n)$. Therefore, the overall worst-case time complexity of the adjMatGraph Class $\in \Theta(n^2)$.	init() in the best case still needs to initialise an adjacency matrix of size $(n \times k) \in \Theta(n \times k)$. $\forall$ other methods in the adjMatGraph Class perform operations $\in \Theta(1)$ in the best case. Therefore, the best case of adjMatGraph $\in \Theta(n \times k)$.

Figure 1.3: Time Complexity for Adjacency Matrix Graph Maze

Data Generation

To generate data for my empirical analysis, I wrote a Python Script (see Appendix 4) designed to create .json configuration files of variously sized mazes for different data structures.

The maximum maze size was determined based on my initial testing of the limitations of my computer's 16 GB RAM Capacity, particularly for the Adjacency Matrix data structure implementation. This excessive RAM usage indicated the space-complexity of each data structure implementation.

To ensure a comprehensive evaluation, I generated 8 variously sized mazes per type, doubling the total number of cells each time (totalCells = columns × rows). This approach allowed me to test a wide array of sizes and uncover any trends or patterns in the performance of the data structures.

The Python Script generated three types of maze: square, vertical, and horizontal, so we could evaluate if the maze dimension effected performance rather than just totalCells. The maze sizes, and range I tested are defined below:

- ***Square Maze range:*** $\{n \times n \mid n \in \mathbb{Z}, 5 \leq n \leq 200\}$
- ***Vertical Maze range:*** $\{5 \times m \mid m \in \mathbb{Z}, 10 \leq m \leq 1600\}$
- ***Horizontal Maze range:*** $\{k \times 5 \mid k \in \mathbb{Z}, 10 \leq k \leq 1600\}$

See Appendix 5 for all configuration files used for square, vertical, and horizontal maze.

Experiment Setup

I designed my experiment in an effort to thoroughly evaluate the performance of the different data structures when generating mazes of different dimensions. The tests were conducted on my Gaming Laptop (see Appendix 3 for specs) with no applications running and without the visualisation using Matplotlib to minimise any potential interference.

Each test case (configuration file) was run ≤ 10 times (yes even a 200 × 200 matrix) to mitigate bias and ensure the reliability of my results. I checked the variance of the results and repeated the tests if the variance exceeded ± 10%. I employed some methodology from Uncertainty Quantification (UQ) and Monte Carlo methods to enhance the robustness of my findings, as well as quantify uncertainties, and identify any outliers see Appendix 2.

Furthermore, I did my best to maintain consistency, so rounded all time measurements to the nearest significant digit before recording them into Appendix 6.

In addition to the total generation timer built into the skeleton code, I used the ‘timeit’ and ‘atexit’ python libraries to implement on the data structure classes to measure the execution time and count the number of times each function is called in each data structure. Please see Appendix 9 for the implementation. This allowed me to compare the performance of the data structures more comprehensively.

The complete dataset, consisting of the results for each data structure and maze type, is stored in Appendix 6. This dataset is the foundation for our empirical analysis and enables us to uncover trends and make predictions. This will also allow us to confirm our hypotheses about the performance of the algorithms, as well as each data structure. We can then recommend the best one.

To further analyse the collected data, I conducted an Exploratory Data Analysis (EDA) using the ‘pandas’ and ‘plotly express’ python libraries in a Python Notebook (Appendix 2). The EDA aimed to reveal any patterns, correlations, or anomalies in the data, providing valuable insights into the behaviour of the data structures across different maze sizes and configurations. It also allowed me to create interactive visualisations for my empirical analysis.

After conducting the EDA, I discovered that a portion of my dataset had been incorrectly generated. This realisation prompted me to revisit my data generation process, rectify the issues in my configuration files, and re-run my experiments. After which I applied my EDA techniques once more to verify that the data had been corrected and to ensure the integrity of my dataset.

Through this experimental setup, I aimed to gather empirical evidence that would allow me to make informed conclusions about the strengths and weaknesses of each data structure in the context of maze representation and navigation.

Empirical Analysis

NOTE:

The following Figures use a Logarithmic Y-Axis, and same colour scheme.

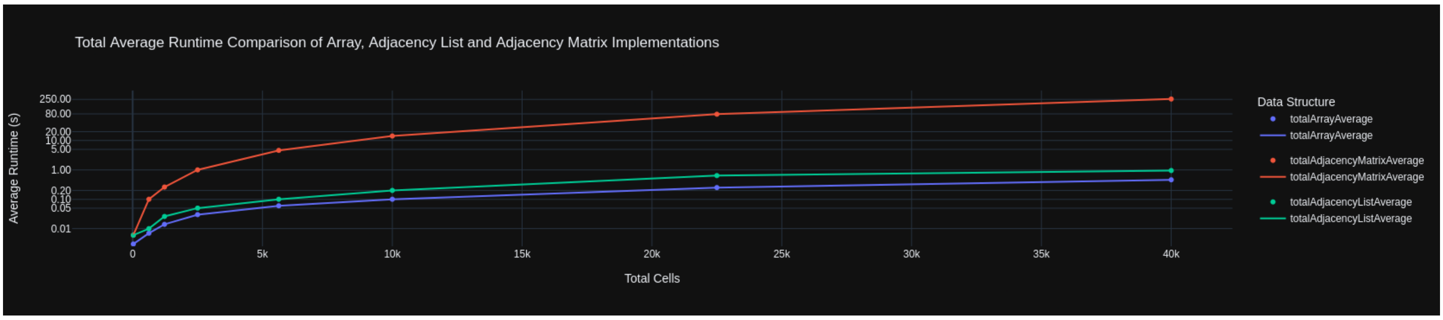


Figure 2.1: Square Maze Average Runtime

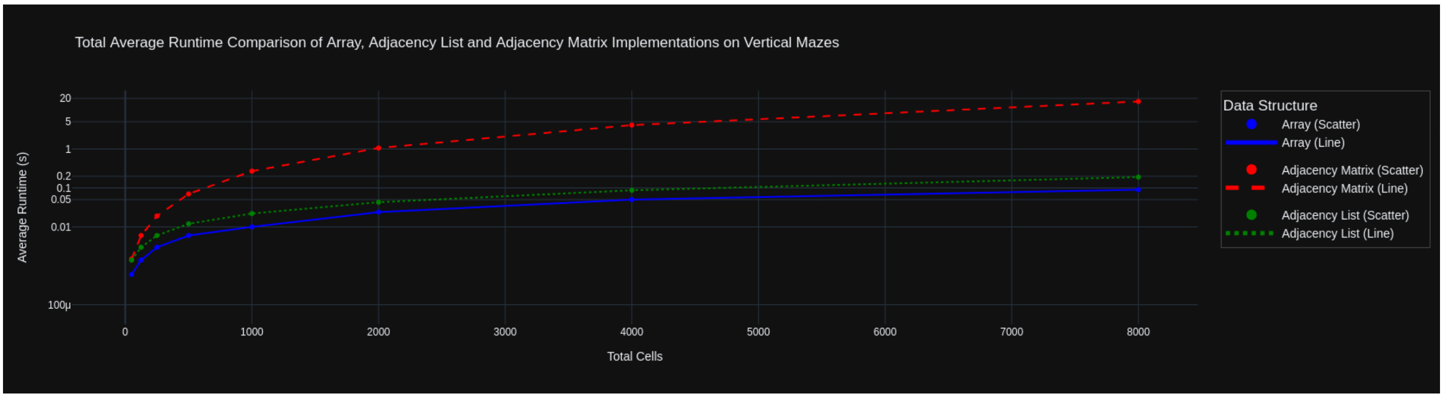


Figure 2.2: Vertical Maze Average Runtime

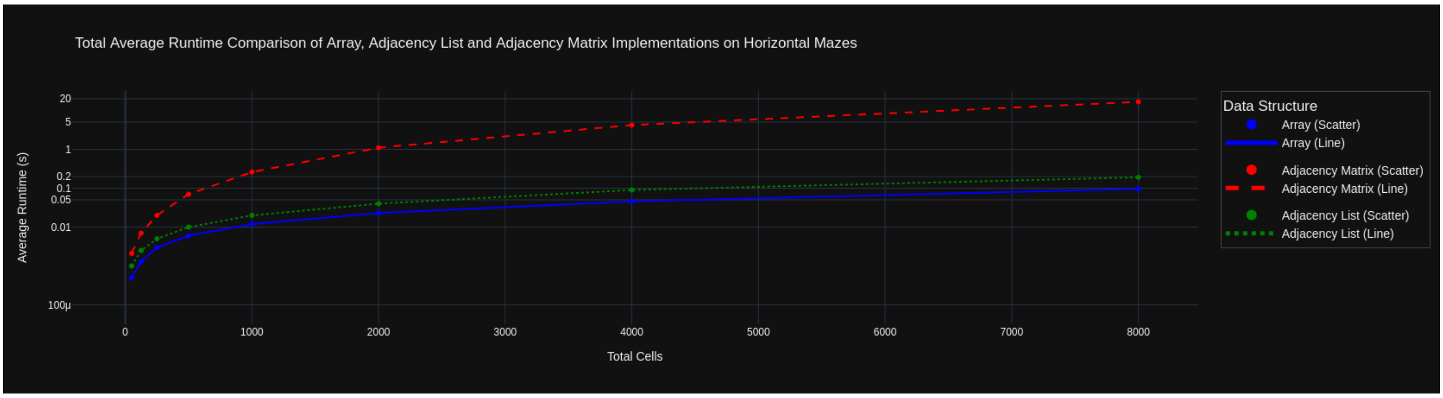


Figure 2.3: Horizontal Maze Average Runtime

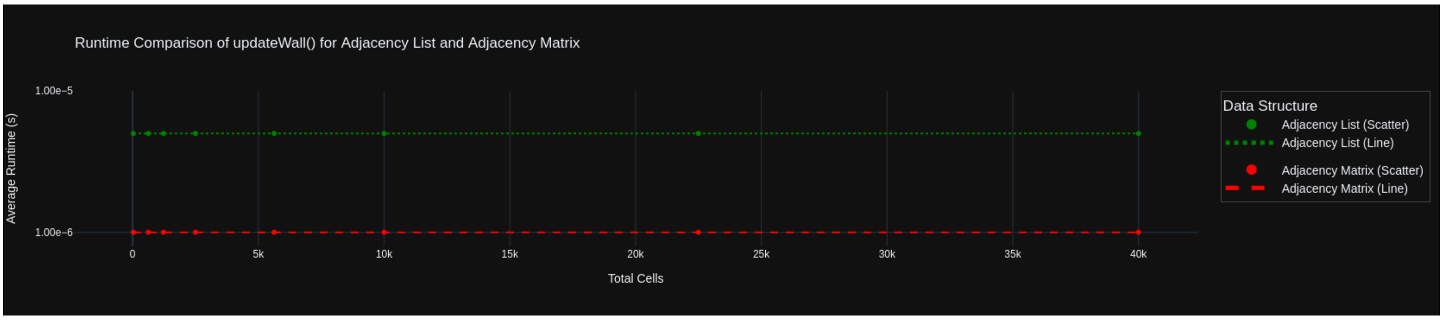


Figure 2.4: updateWall() Average Runtime

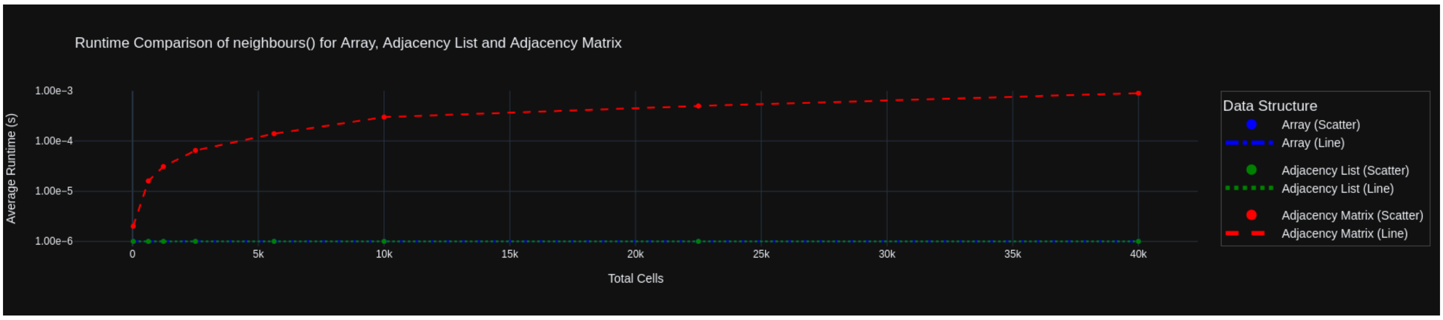


Figure 2.5: neighbours() Average Runtime

Results Discussion

Figure 2.1 illustrates the average time taken by different data structure implementations to generate square mazes of varying dimensions. The graph clearly shows that the Adjacency Matrix implementation is the slowest, requiring approximately 250 seconds to generate a 200×200 maze (40,000 cells). This coincides with our theoretical analysis' worst case of $\Theta(n^2)$. In contrast, the Array implementation has proven to be the most efficient, taking approximately 0.5 seconds to generate a maze of the same size. Interestingly, my Adjacency List implementation falls close to the Array, with a generation time of around 0.96 seconds for the same size maze. Further analysis reveals that the trend for the Adjacency List's runtime can be approximated by the function f(x) ≈ 2 × avgRuntime(adjlist). This indicates that it takes roughly twice as long as the Array implementation to generate mazes of the same dimensions.

Figures 2.2 and 2.3 demonstrate that the performance trends for generating vertical (5×10→5×1600) and horizontal (10×5→1600×5) mazes are nearly identical. This indicates that maze orientation is not a significant factor in generation time. These case examples further support this finding, showing that for mazes with a comparable number of total cells, the Array, and Adjacency List implementations exhibit similar performance regardless of maze shape. Comparing these results to Figure 2.1 (square mazes), we observe consistent trends across all maze types, with the Array being the most efficient and the Adjacency Matrix being by far the slowest. We can safely conclude that this means the total number of cells in the maze, rather than its orientation or shape, is the primary factor influencing generation time.

In Figure 2.4, we observe that the updateWall() operation in both the Adjacency List and Adjacency Matrix implementations has a constant time complexity $\Theta(1)$. The Adjacency Matrix's updateWall() takes approximately 1.00e-6 seconds (1.00×10^{-6}), while the Adjacency List's updateWall() takes around 5.00e-6 seconds (5.00×10^{-6}). When referring to Appendix 2, which further explores the number of calls per function, we can see that updateWall() is called for every edge in the graph. From our knowledge of how Graphs represent Mazes we could've predicted this, and know that this approximately equals the total number of cells. Despite this difference in running time of updateWall() for Adjacency List vs. Matrix we can assume this is not a dominant term as it is $\Theta(1)$, and Adjacency Matrix being much slower overall.

Figure 2.5 examines the performance of the neighbours() function for the Array, Adjacency List, and Adjacency Matrix implementations. The neighbours() function is called approximately 2 × totalCells count of the maze. The Array and Adjacency List implementation both have a constant time complexity for the neighbours() function, taking around 1.00e-6 seconds (1.00×10^{-6}) per call. On the other hand, the Adjacency Matrix's neighbours() function time complexity increases linearly with the total number of cells in the maze, following cell count as it doubles. This linear growth in time complexity for the Adjacency Matrix's neighbours() function contributes to its overall slower performance compared to the Array and Adjacency List implementations, but not compared to quadratic complexity of the addVertex(), and init() in the Adjacency Matrix Implementation. For further discussion of this please see Appendix 1.

Real Time Complexity

Data Structures	Time Complexity
Array	$\Theta(n)$
Adjacency List	$\Theta(n)$
Adjacency Matrix	$\Theta(n^2)$

Functions	Time Complexity
Adjacency List, and Matrix updateWall()	$\Theta(1)$
Array, and Adjacency List neighbours()	$\Theta(1)$
Adjacency Matrix neighbours():	$\Theta(n)$

Equation Approximation

To approximate the equation for each data structure, we first calculate a ratio using this equation: ***time ÷ totalCells***

By examining this ratio, we can then determine the trend between time and the total number of cells.

arrayMaze ≈ 0.000011 × totalCells

The ratios for the Array implementation remain relatively constant when comparing time, and the total number of cells. This indicates that the time complexity grows linearly with the total number of cells, confirming our theoretical $\Theta(n)$ hypothesis.

$$\textit{adjListGraph} \approx 0.0000046 \times \textit{totalCells}$$

Similar to the Array implementation, the ratios for the Adjacency List remain relatively constant, indicating that the time complexity grows linearly with the total number of cells, again confirming our theoretical Θ(n) hypothesis.

$$\textit{adjMatGraph} \approx 0.00000001 \times (\textit{totalCells})^2$$

In contrast, the ratios for the Adjacency Matrix implementation did not remain constant when comparing time, and the total number of cells. It only started remaining relatively constant when calculated using time ÷ $(\text{totalCells})^2$, which confirms our $\Theta(n^2)$ hypothesis.

These equations provided us a clear understanding of how the time complexity of each data structure grows with respect to the total number of cells in the maze. The Array and Adjacency List implementations exhibit linear growth $\in \Theta(n)$, while the Adjacency Matrix implementation exhibits quadratic growth $\in \Theta(n^2)$. This confirms our theoretical hypotheses made earlier.

Conclusion

In this analysis, we have explored the performance of different data structure implementations for maze generation. We focussed on Arrays, Adjacency Lists, and the Adjacency Matrix. By examining the time complexity and approximating the equations for each implementation, we have gained insights into their efficiency and scalability.

Our analysis has shown that the Array implementation consistently outperforms the other data structures, with the Adjacency List being approximately twice as slow as the Array, and the Adjacency Matrix exhibiting significant inefficiencies in its current implementation.

These findings align with the principle of Occam's Razor, which suggests that the simplest solution is often the best. The Array implementation offers both simplicity and efficiency, making it the preferred choice for maze generation.

Based on the theoretical, and empirical findings of my experiments, I recommend using the Array implementation for maze generation. This is especially apparent when generating larger mazes. The Array, and Adjacency List offer the lowest time complexity, with an approximation of Θ(n), where n is the total number of cells in the maze. This makes these data structures the most efficient choice for mazes with a large total number of cells. However, the Adjacency Matrix implementation, with a time complexity of $\Theta(n^2)$, should be avoided for generating large dimension mazes due to its quadratic growth in time complexity.

Given more time, I would have liked to have explored the impact of entry and exit points on the running time of maze generation algorithms. Additionally, investigating efficient maze-solving techniques, such as Depth-First Search (DFS) or Breadth-First Search (BFS), would be fun. As well as exploring the concept of perfect mazes, which are mazes with a single unique path from the entrance to the exit.

I would have also liked to have explored space-complexity more as I saw my RAM Usage skyrocket when testing Adjacency Matrices suggesting it did not have any benefit, however I did not explicitly measure this.

Other data structures that could be explored for maze generation include quad trees, bitmaps, and disjoint sets. These structures have the potential to provide alternative approaches to maze generation and may offer unique benefits in terms of efficiency or functionality.

I would have also liked to delve deeper into Uncertainty Quantification (UQ) and Monte Carlo methodology to further validate our findings and assess the robustness of our conclusions.

In summary, our findings indicate that the Array implementation is the fastest and most efficient choice for maze generation, followed by the Adjacency List, while the Adjacency Matrix implementation requires Optimisation.

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