Efficiency and Scalability Analysis of Graph Representations in Maze Generation Algorithms

Mrwan Alhandi

Table of Contents

Theoretical Analysis

adjListGraph

Theoretical Analysis		
Operations	Best Case	Worst Case
updateWall()	**O(1)** - **Vertex Conversion:** O(1). - **Adjacency & Existence check:** O(1) - **Existence in the list:** If vertex2 is not in the list, adding it is O(1). If vertex2 is the first in the list, removing it is O(1).	**O(n)** - **Vertex Conversion:** O(1). - **Adjacency & Existence check:** O(1) - **Existence in the list:** If vertex2 is near the end of the list or not present, and vertex1 has many neighbors (i.e., the list is long), this check becomes. O(d(v)), where d(v) is the degree of the vertex. This means it could approach O(n) in a densely connected graph where a vertex could potentially be connected to nearly every other vertex.
neighbours()	**O(1)** - The neighbours() function consistently checks the four possible neighboring positions, which is a fixed number of operations and therefore is always O(1), irrespective of the graph's density.	**O(1)** - Same operations in all cases.

adjMatGraph

Theoretical Analysis		
Operations	Best Case	Worst Case
updateWall()	**O(1)** - **Vertex Location:** the vertex is at the beginning, the operation can return true immediately, resulting in constant time complexity. - **Edge Update:** The update is immediate once the vertex is located, contributing to the O(1) complexity in the best case.	**O(rowsNum*ColsNum) = O(n^2)** - **Vertex Location:** the vertex is at the end or not found, requiring a full scan of the matrix, which results in O(rowsNum*colsNum). - **Edge Update:** The update is a constant time operation after the vertex is located but this comes after a potentially full matrix traversal.
neighbours()	O(1) - The calculation of neighboring cells involves four fixed operations—four conditional checks and potential additions to the neighbor list—regardless of the cell's position in the matrix.	O(1) - This is identical to the best case as the number of operations remains consistent. Regardless of the cell's position, only four conditional checks are performed, and complexity does not escalate with matrix size since it involves checking only the immediate neighbors of a single cell.

Data Generation

To rigorously evaluate the performance of two graph representations, adjMatrixGraph and adjListGraph, we generate a suite of maze configurations. These configurations are systematically varied by maze size.

In our initial tests, we found that variations in maze proportions—tall, wide, or square—and different placements of entrances and exits do not significantly affect the algorithms' run times. This consistent performance across various configurations suggests that the maze's size has a greater impact on algorithm efficiency than its shape or entry points. Given these results, we have determined that it is not necessary to include these details in the report, as they do not contribute meaningfully to our understanding of the algorithms' effectiveness.

Maze Sizes

Three distinct sizes are selected to represent varying complexities:

- **Small:** Mazes with fewer than 100 cells, testing the algorithms' efficiency in a constrained environment.
- **Medium:** Mazes with 150 to 170 cells, offering a balance between complexity and solvability. This choice was because the matrix representation started to take long time to scale.
- **Large:** Mazes with 200 to 400 cells, challenging the scalability of the algorithms.

This approach ensures that the test of algorithms' performance as input size increases is both comprehensive and reliable, effectively demonstrating how scalability and efficiency vary across different configurations.

Experimental Setup

In our study, we assessed the execution times for the updateWall() and neighbours() functions within two graph implementations, adjMatGraph and adjListGraph. To accurately measure the duration of each operation, we utilized Python's time module. Execution time was recorded at the initiation of each function call and immediately after the function completed. The resulting times were accumulated in a list, and upon completion of all operations, the individual times were summed to determine the total execution time for each method.

To ensure a comprehensive analysis, nine configuration files were created to represent three different maze sizes: small, medium, and large, with three files for each category. To minimize anomalies and derive robust conclusions, we averaged the results from multiple runs for each maze size. This approach provided a representative performance metric for the small, medium, and large mazes under both graph representations, facilitating an accurate comparison of their efficiency and scalability.

Empirical Analysis

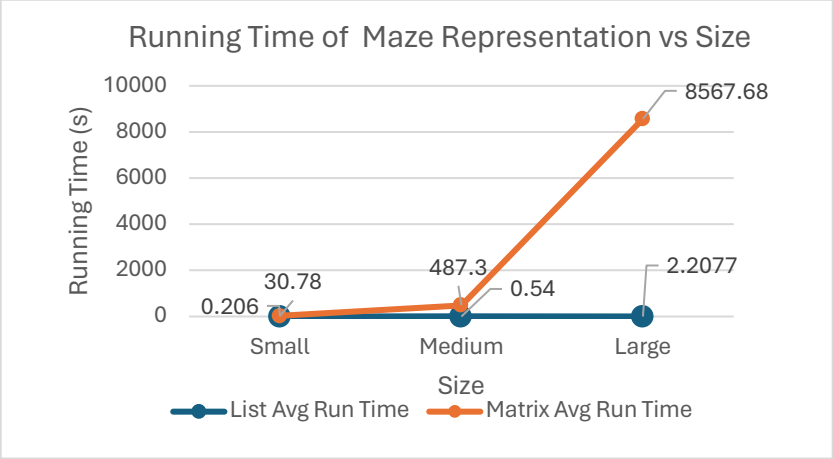

Figure 01: Running Time of Maze Representation vs Size

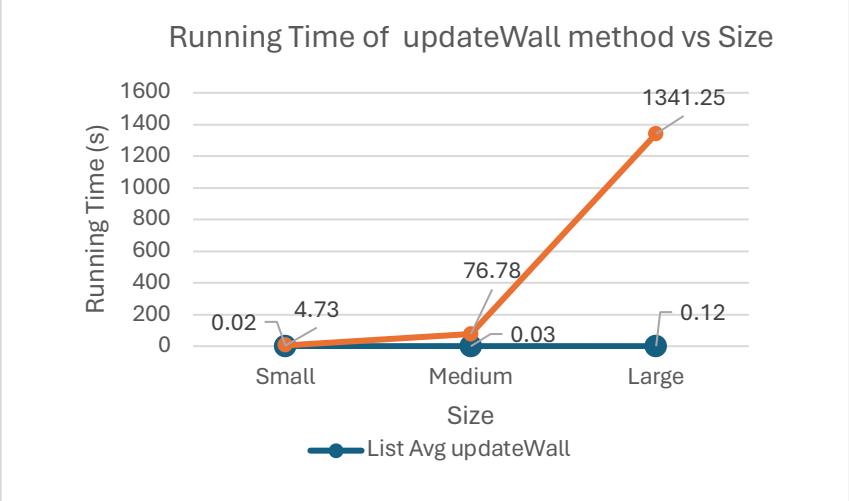

Figure 02: Running Time of updateWall vs Size

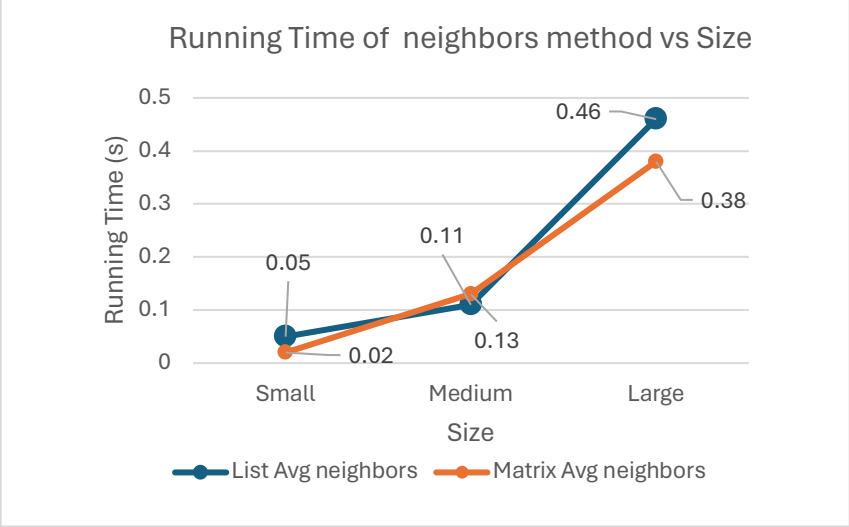

Figure 03: Running Time of neighbors vs Size

Empirical vs Theoretical

Environment

The tests were conducted on a MacBook Pro with an M2 Pro processor. I opted to run the tests on a local machine primarily for convenience; this setup enabled the use of advanced code editors, which offer enhanced functionality over simpler text editors like Nano. This allowed for more efficient file editing and effective management of tests, including the ability to make modifications swiftly as required. Additionally, I conducted sample testing on the teaching server and observed similar time difference scales, which confirmed the consistency of the results across various environments.

Analyzing List vs Matrix Representation

Analyzing the code's runtime revealed that loops significantly impact performance. List methods with O(n), devoid of nested loops, contrast sharply with the Matrix Representation, where nested loops are common. Such inefficiency is evident as matrix runtimes balloon with larger inputs, suggesting a potential O(n^4) complexity, exemplified by a 16-fold runtime increase when input size doubles from 100x100 (23.44s)to 200x200 (395.61s). This multiplicative factor of 16, when doubling the input size, is indicative of O(n^4) complexity: for an algorithm with this complexity, when the input size doubles (a factor of 2), the runtime is expected to increase by the factor raised to the fourth power (2^4 =16), hence aligning with the empirically observed results. This pattern underscores the criticality—and inherent beauty—of loop optimization in algorithm design.

Analyzing updateWall Method

Theoretically, the `updateWall` for matrix representation is expected to exhibit O(n^2) complexity due to the presence of a single nested loop, where the operations involved are limited to assignments and index retrieval. However, empirical evidence suggests a complexity of O(n^4) for the matrix representation. Upon closer inspection of the `GraphMaze` class's `initCells` method, we discover the root of this discrepancy: a nested loop iterates over each row and column, invoking the `addEdge` method—upon which `updateWall` depends. Since `addEdge` itself contains a nested loop to locate positions within the matrix, the compounded iterations across these methods amplify the overall computational complexity, leading to the observed O(n^4) performance. For List Representation, the run time is O(n) which confirms the theory analysis.

Analyzing neighbors Method

The `neighbors` method contains no loops and is called once in `GraphMaze`, resulting in constant O(1) complexity for both representations. Empirical data show only minor changes with increasing size, aligning with observations that doubling the input size doubles the time, indicating linear O(n) behavior.

Conclusion

In conclusion, our study provides a comprehensive analysis of the performance and scalability of two graph representations, adjMatGraph and adjListGraph, in the context of maze solving algorithms. Theoretical analysis revealed contrasting complexities for various operations, with the matrix representation often exhibiting higher complexities due to nested loops.

The list representation outperforms the matrix representation due to the inefficiencies inherent in the matrix structure. The need for two attributes—one for indexing and another for maze adjustment— with the nested loops slows down the algorithm significantly. These findings underscore the critical importance of thoughtful data structure design and code implementation choices.

These findings underscore the importance of algorithm design, data structure choices, and implementation, emphasizing the need for efficient loop utilization to enhance performance. Additionally, our study highlights the critical role of empirical validation in algorithm analysis, showcasing the nuanced complexities that may arise in real-world scenarios and the errors we may made by guessing the theoretical time complexity.

Appendix

In seconds	Maze Size	List Total Run time	updateWall List	neighbours List	Matrix Total Run time	updateWall Matrix	neighbours Matrix
File 1 (Small Maze)	50 x 50	0.046	0.00	0.01	1.58	0.24	0.00
File 2 (Small Maze)	70 x 70	0.08	0.01	0.02	5.76	0.88	0.01
File 3 (Small Maze)	100 x 100	0.08	0.01	0.02	23.44	3.61	0.01
File 1 (Medium Maze)	150 x 150	0.15	0.01	0.03	122.73	19.08	0.03
File 2 (Medium Maze)	160 x 160	0.18	0.01	0.03	161.02	25.70	0.05
File 3 (Medium Maze)	170 x 170	0.21	0.01	0.05	203.55	32.00	0.05
File 1 (Large Maze)	200 x 200	0.29	0.02	0.05	395.61	62.16	0.05
File 2 (Large Maze)	300 x 300	0.70	0.04	0.14	2043.02	321.64	0.13
File 3 (Large Maze)	400 x 400	1.22	0.06	0.27	6129.05	975.43	0.20
File 4 (For List Only)	1000 x 1000	7.84	0.38	1.54			
File 5 (For List Only)	2000 x 2000	34.65	1.62	6.69			

Table 01: Run Time Tests Records