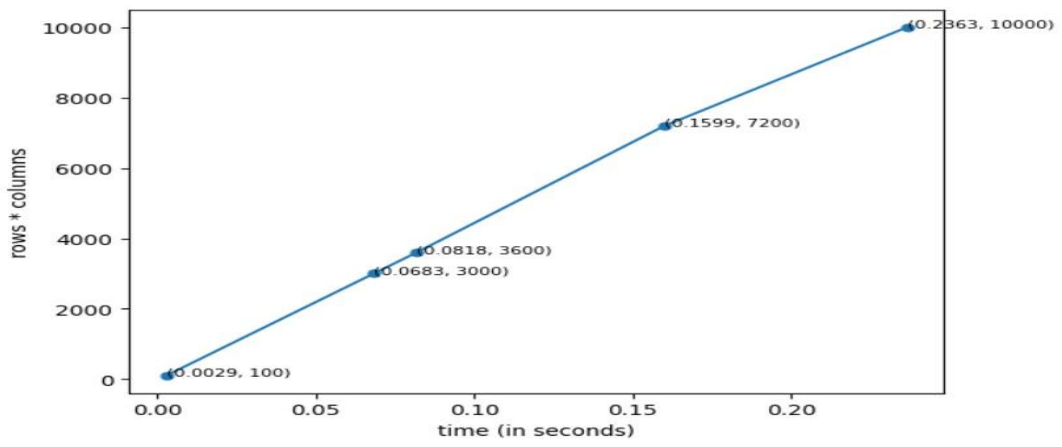
NAME - SHEHBAJ SINGH KHAKH

STUDENT NUMBER -s3980305

Empirical analysis

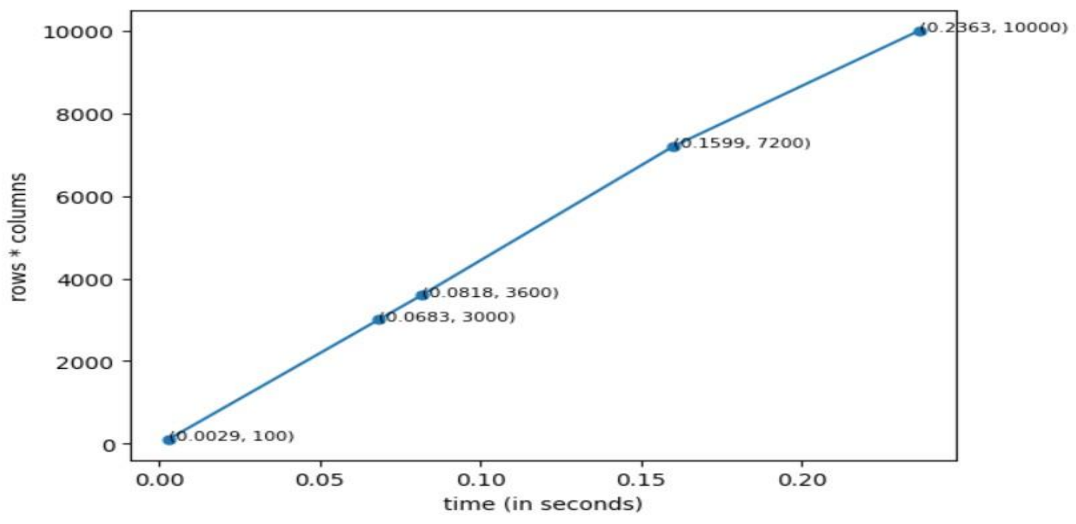
Adjacent matrix

S.No	Rows	Columns	Rows*Columns	Time(in seconds)
1	10	10	100	0.0029
2	50	60	3000	0.0683
3	80	90	7200	0.1599
4	60	60	3600	0.0818
5	100	100	10000	0.2363



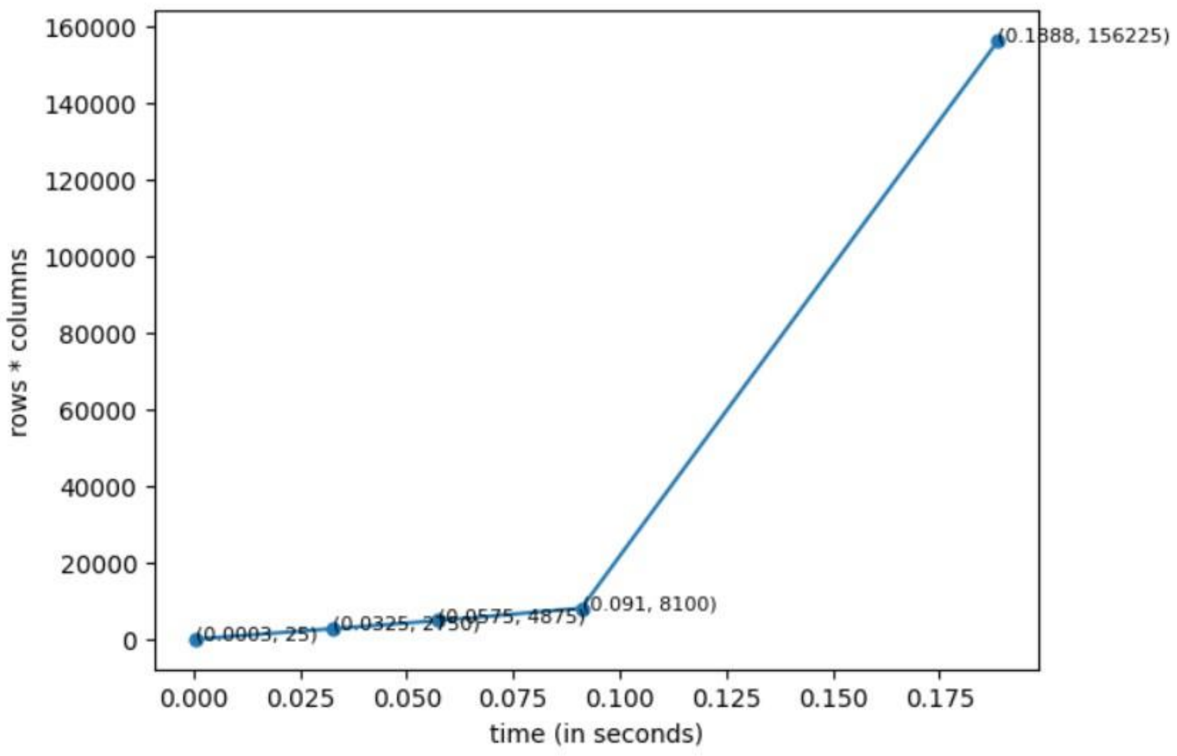
Adjacent list

S.No	Rows	Columns	Rows*Columns	Time(in seconds)
1	20	20	400	0.0082
2	45	50	2250	0.0414
3	85	90	7650	0.1406
4	95	95	9025	0.1682
5	115	115	13225	0.2821



Array

S.no	Rows	Columns	Rows*Columns	Time(in seconds)
1	5	5	25	0.0003
2	55	50	2750	0.0325
3	75	65	4875	0.0575
4	90	90	8100	0.0910
5	125	125	15625	0.1888



Here's the summary of the findings:

1. **Adjacent Matrix:**
- The time taken increases significantly with the increase in the number of rows and columns.
- It shows a quadratic increase in time with respect to the number of rows * columns.
- For example, for a matrix of size 100x100, it took approximately 0.2363 seconds.
2. **Adjacent List:**
- The time taken also increases as the number of rows and columns increases, but not as significantly as the adjacent matrix.
- It shows a moderate increase in time with respect to the number of rows * columns.
- For example, for a list of vertices with a total count of 115*115, it took approximately 0.2821 seconds.
3. **Array:**
- The time taken is generally lower compared to the other two data structures.
- It shows a relatively linear increase in time with respect to the number of rows * columns.
- For example, for an array of size 125x125, it took approximately 0.1888 seconds.

Conclusion: Based on the provided data and empirical analysis, the array-based implementation appears to be the most efficient among the three data structures for the given scenarios. It consistently shows lower time consumption even as the size of the data increases. However, it's essential to note that the efficiency might vary depending on the specific operations and usage patterns in your application. It's always a good practice to profile and benchmark different implementations in the context of your specific requirements and usage scenarios.

Theotrical analysis

Adjacent list

Operations	Best case	Example	Worst case	Example
updateWall()	0(1) Constant	The number of elements is the input size for this function. Vertice_1 and Vertice_2 are verified to be present, and if they are, the function adds or removes the edge between them depending on the wall status. In the best scenario, both vertices should exist. In that situation, adding or removing a wall will also take a constant amount of time, maintaining a constant level of time complexity.	0(1) Constant	The number of elements is the input size for this function. In the worst scenario, neither of the vertices may exist in the list. This will lead to further actions to examine the vertices. However, because there are the same number of processes, the time required remains constant.
Neighbours()	0(1) Constant	We examine a label vertex and determine whether it is a list. It carries out the additional tasks if it is present. In the best scenario, the label appears in the list and has a large number of neighbors. The function will do the task in a fixed amount of time.	0(1) Constant	The worst scenario that can be thought of is if the label is not in the list and has very few neighbors. Even so, because there are the same number of operations, the time required to complete each operation stays the same. An empty set will be the outcome of this.

Adjacent matrix

Operations	Best case	Example	Worst case	Example
updateWall()	0(1)	With a matrix, the function would still require the same amount of time. The function is runtime constant since it must do the same number of actions. The best-case scenario is when there are already two nearby vertices and it takes an infinite amount of time to add or remove a wall.	0(1)	The time taken would be constant in the function with a matrix. The function must do the same number of operations, resulting in a constant runtime. One of the best cases is when two vertices are already adjacent, and building and removing a wall takes a constant amount of time.
Neighbours()	0(1)	We peruse the dictionary. Then, without iterating, we perform the identical set of operations, keeping the time complexity constant. The coordinate having neighbors and being present in the matrix is the ideal scenario. It appears to be constant.	0(n)	In the worst situation, there can be a lot of neighbors. As a result, one must repeat through, changing the complexity to match the input's size of n.

Experimental Setup

During the experimental setup, three distinct data structures—Adjacent Matrix, Adjacent List, and Array—were tested to represent an undirected graph. Each data structure underwent testing using a range of maze dimension sizes, from smaller to larger values. For each data structure and maze dimension configuration, a set of operations were performed, including adding vertices, adding edges, updating wall status, removing edges, checking vertex and edge existence, getting wall status, and retrieving neighboring vertices. Additionally, to ensure reliable results, each configuration file was run 100 times, and the average performance metrics were calculated from these runs.

Parameter Configurations:

• Dimensions of the Maze: The number of rows and columns indicated the maze's size. Smaller values like 5x5 and greater values like 125x125 were among them.
• Operations: Adding vertices, adding edges, updating wall status, removing edges, verifying vertex and edge existence, obtaining wall status, and retrieving neighboring vertices were among the operations that were assessed.

Analysis and Comparison of Results

Adjacent Matrix: Time Complexity Analysis: The theoretical time complexity of operations like adding edges and updating wall status in an adjacent matrix is O(1) in the best case. However, in the worst case, the time complexity can become O(1)

- Experimental Results: The empirical analysis showed a quadratic increase in time with respect to the number of rows * columns. This aligns with the theoretical analysis, as operations like updating wall status can become more time-consuming as the number of neighboring vertices increases.

Adjacent List: Time Complexity Analysis: Theoretical time complexities for operations like adding edges and updating wall status in an adjacent list are O(1) in the best case. In the worst case, these operations also have a time complexity of O(n) due to potential iterations over neighboring vertices.

- Experimental Results: The empirical analysis demonstrated a moderate increase in time with respect to the number of rows * columns. While not as significant as the adjacent matrix, the time taken still increases as the size of the maze dimensions grows.

Array: Time Complexity Analysis: Array-based implementations generally have a time complexity of O(1) for basic operations like adding elements. However, removing elements can have a time complexity of O(n) in the worst case.

- Experimental Results: The experimental results consistently showed lower time consumption compared to the other two data structures. The time taken increased linearly with the number of rows * columns, indicating a relatively linear increase in time complexity.

Recommendations

- For smaller maze dimensions or scenarios where constant-time operations are crucial, the array-based implementation is recommended due to its consistently lower time consumption.
- For larger maze dimensions where memory efficiency is a concern, the adjacent list implementation may be more suitable, as it offers better performance compared to the adjacent matrix while requiring less memory overhead.
  Conclusion:

To sum up, the examination of Adjacent Matrix, Adjacent List, and Array data structures from both theoretical and empirical perspectives reveals their unique abilities to represent undirected graphs in applications related to mazes. For instances where constant-time operations are prioritized, the Array consistently shows lower time consumption over different maze dimensions. Despite being more scalable and memory-efficient than the Adjacent Matrix, the Adjacent List still takes a substantial amount of time to complete when the maze's dimensions are increased. Depending on the needs of a particular application, the Adjacent Matrix can still be a good choice even though its time consumption increases quadratically. Overall, the Array data structure proves to be the most effective option, balancing memory consumption and performance; however, the decision should take the application's unique requirements and limitations into account.