ASSIGNMENT 1

IMPLEMENTING AND EVALUATING DATA STRUCTURES FOR MAZE GENERATION

Presented by :
Linh Thuy Do - s3927777

I. Theoretical Analysis

a. Adjacent list implementation (EdgeListGraph)

Operation	Best Case	Worst Case
updateWall()	Θ(1)	Θ(E)
Scenario	The edge to be updated is found in the first position of the list of edges. The loop terminates after one iteration.	The edge to be updated is located at the last position of the list of edges or does not exist, requiring a full traversal of the list. Here, E is the number of edges
Explanation (Running time)	**Iteration Over Edges:** In the best case, the loop runs for just one iteration, when the correct edge is the first in the list. => The best-case time complexity is: Θ(1)	**Iteration Over Edges:** The loop runs for all E edges => The worst-case time complexity is: Θ(E)
Example scenario	- Consider a graph where the first edge in the list connects the vertices *A* and *B*. - When *vert1* is *A* and *vert2* is *B*, the function finds and updates the wall status after just one iteration, resulting in a time complexity of Θ(1).	- Consider a graph where the last edge in the list connects the vertices *A* and *B*, or they are not connected at all. - When *vert1* is *A* and *vert2* is *B*, the function must iterate through all E edges to find the correct one or determine that it does not exist. The time complexity in this scenario is Θ(E).
neighbours()	Θ(1)	Θ(E)
Scenario	The vertex has no neighbors, so the loop completes instantly.	All edges need to be checked to collect neighbors, which occurs when the vertex has multiple neighbors or no neighbors, requiring a full traversal of the edge list.
Explanation (Running time)	**- Initialization of *neighbours* List:** This operation takes constant time O(1). **- Iteration Over Edges:** In the best case, the loop runs for a minimal number of iterations before finding the connected edge or determining no connections. This could happen as quickly as after checking the first edge, making the running time O(1). => The best-case time complexity is: Θ(1)	**- Initialization of *neighbours* List:** O(1) **- Iteration Over Edges**: The loop runs for all E edges. => The worst-case time complexity is: Θ(E)
Example scenario	- Consider a graph where the vertex *A* is isolated, meaning it has no connecting edges. - When *label* is *A*, the function will quickly determine there are no connected vertices without needing to iterate through all edges. The time complexity is minimal: Θ(1).	- Consider a graph where every vertex is connected by an edge to the vertex *A*. For example, a star-shaped graph with *A* at the center. - If *label* is *A*, the function must iterate through all E edges to find and add each connected vertex to the *neighbours* list. The time complexity in this scenario is Θ(E).

b. Adjacent matrix implementation (incidenceMatGraph)

Operation	Best Case	Worst Case
updateWall()	Θ(1)	Θ(E)
Scenario	The edge to be updated is found in the first position of the list of edges. The loop terminates after one iteration.	The edge to be updated is located at the last position of the list of edges or does not exist, requiring a full traversal of the list.

Explanation (Running time)	**Iteration Over Edges:** In the best case, the correct edge is found in the first iteration of the loop. => The best-case time complexity is: Θ(1)	**Iteration Over Edges:** The loop runs for all E edges. => The worst-case time complexity is: Θ(E)
Example scenario	- Consider a graph where the first edge in the list connects the vertices *A* and *B*. - When *vert1* is *A* and *vert2* is *B*, the function finds and updates the wall status after just one iteration, resulting in a time complexity of Θ(1).	- Consider a graph where the last edge in the list connects the vertices *A* and B, or they are not connected at all. - When *vert1* is *A* and *vert2* is *B*, the function must iterate through all E edges to find the correct one or determine that it does not exist. The time complexity in this scenario is Θ(E).
neighbours()	Θ(V)	Θ(V * E)
Scenario	The vertex has no neighbors, so no edges are checked, where V is the number of vertices and E is the number of edges.	All edges and all vertices are checked to find neighbors, where V is the number of vertices and E is the number of edges.
Explanation (Running time)	**- Initialization of *neighbours* List:** This operation takes constant time O(1). **- Check if *label* is in *self.vertices*:** This operation takes O(V) time, as it involves searching through the *vertices* list. **- Finding the Index *v_index*:** This takes O(V). Iteration over Edges (Outer Loop): In the best case, the loop runs for E iterations, where E is the number of edges. However, if the first edge is connected, the inner loop may terminate early, making this part of the operation take O(1) time. **- Inner Loop for Checking Connections:** This loop runs for V iterations in the worst case, but in the best case, it may take O(1). => The best-case time complexity is: Θ(V+1) = Θ(V)	**- Initialization of *neighbours* List:** O(1) **- Check if *label* is in *self.vertices*:** O(V) **- Finding the Index *v_index*:** O(V) **- Iteration over Edges (Outer Loop): The** loop runs for E iterations. **- Inner Loop for Checking Connections:** For each edge, the inner loop checks V vertices, and since it's the worst case, it may need to evaluate all V vertices per edge. => The worst-case time complexity is: Θ(V+V×E) = Θ(V*E)
Example scenario	- Consider a graph where the vertex *A* is isolated, meaning it has no connecting edges. - When *label* is *A*, the function quickly determines that there are no connected vertices, and the time complexity is minimal: Θ(V).	- Consider a fully connected graph (complete graph) where every vertex is connected to every other vertex. - If the *label* is one of these highly connected vertices, the function must examine each edge and every vertex connected to it, leading to the maximum time complexity of Θ(V*E).

II. Empirical Analysis

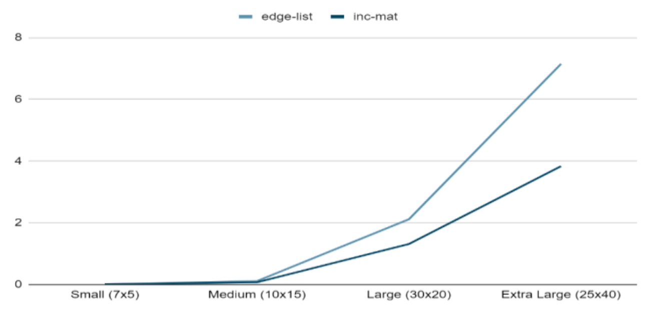

Figure 1: Difference in Group 1

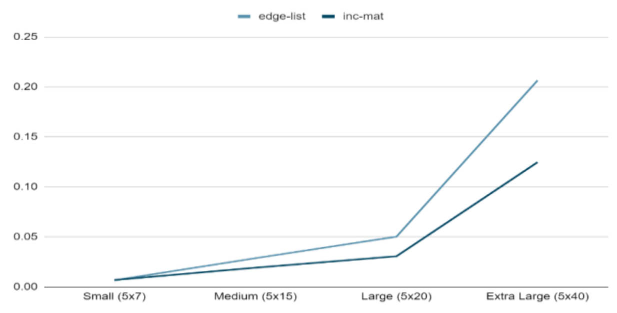

Figure 2: Difference in Group 2

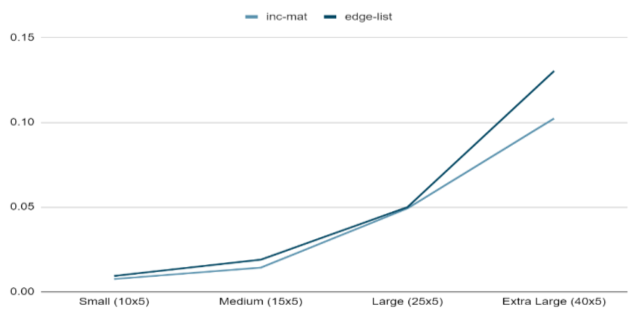

Figure 3: Difference in Group 3

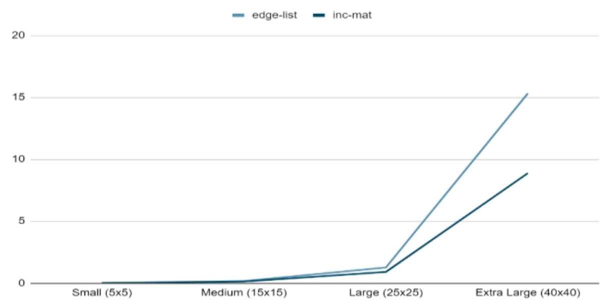

Figure 4: Difference in Group 4

a. Data & Experimental Setup

In this study, we aim to analyze the performance of two different graph data structures (edge-list and incidence matrix) when used to generate mazes of varying dimensions. The primary parameters in this experiment are **E** (Number of edges in the graph) and **V** (Number of vertices in the graph). To gain an understanding of the performance of each data structure, the experiments are conducted across four different maze-size categories to test how different dimensions impact the running time of maze generation.

Group	Maze Sizes	Purpose
Group 1: Random Rows and Columns	**Small**: 7x5 **Medium**: 10x15 **Large**: 30x20 **Extra Large**: 25x40	Test the effect of both row and column variations on the running time.
Group 2: Constant Rows, Increasing Columns	**Small**: 5x7 **Medium**: 5x15 **Large**: 5x20 **Extra Large**: 5x40	Evaluate how increasing columns while keeping rows constant affects the running time.
Group 3: Increasing Rows, Constant Columns	**Small**: 10x5 **Medium**: 15x5 **Large**: 25x5 **Extra Large**: 40x5	Examine the impact of increasing rows with constant columns on performance.
Group 4: Square Mazes (Rows = Columns)	**Small**: 5x5 **Medium**: 15x15 **Large**: 25x25 **Extra Large**: 40x40	Understand how performance scales with input size n^2 when rows equal columns.

- The time to generate the maze and perform graph operations (like adding edges or updating walls) is measured using a *time* module. The timings are recorded using *time.perf_counter()* before and after the operations and the difference is calculated to capture the precise running time in seconds. This approach ensures that the timing is accurate and consistent, especially when dealing with very small operations that are completed in milliseconds.

b. Insights Derived

With the choice of maze sizes for testing, I covered a broad spectrum of scenarios that reveal the performance characteristics of both the Edge list and Incidence Matrix data structures. These insights will offer a detailed understanding of how each structure behaves under various conditions:

1. Scalability and Efficiency Evaluation: The selected maze sizes covered different graph densities, from sparse to dense. As the graph size increased, the Edge-List showed significant slowdowns, especially in dense graphs, while the Incidence Matrix maintained better efficiency.

2. Impact of Rows vs. Columns: By varying dimensions (e.g., 7x5 vs. 5x7), I observed how each structure handled different aspect ratios. The Edge-List struggled with longer corridors, while the Incidence Matrix managed connections more efficiently, though it too slowed down in dense graphs.

3. Handling Sparse and Dense Graphs

- **Small and Medium Sizes**: In sparse graphs, both structures performed well, but the Edge-List's linear search began to show inefficiencies as the graph size grew.

- **Large and Extra Large Sizes**: In denser graphs, the Edge-List's performance declined due to the increased number of edges, while the Incidence Matrix handled these conditions more effectively.

4. Timing Consistency and Reliability: By generating each maze size multiple times, I ensure that my timing data is reliable and consistent. Averaging the results helps mitigate any anomalies, providing a clearer picture of true performance.

5. Theoretical Insight into Observed Patterns

- **Edge-List Graph**: The time complexity of operations such as *updateWall()* and *neighbours()* varies depending on the position of edges and the number of neighbors. For example, in dense graphs (like the 40x40 maze), the worst-case time complexity for *neighbours()* is evident, leading to higher runtime as seen in my timing data.
- **Incidence Matrix Graph**: The incidence matrix generally performs better in dense graphs due to its structured approach to storing edge connections. However, as the number of vertices and edges increases, the worst-case complexity can lead to longer runtimes, especially in scenarios like the 40x40 maze.

6. Why Edge-List Runs Slower than Incidence Matrix

The Edge List runs slower because it requires linear searches through potentially large lists of edges, which becomes increasingly inefficient as the number of edges grows. In contrast, the Incidence Matrix allows for direct access to connections between vertices, making it more efficient, particularly in dense graphs where the number of edges is high. This structured approach enables the Incidence Matrix to handle operations faster, especially in larger and more complex graphs.

III. Evaluation of Data Structures

1. Performance in Small Mazes:

- **Edge-List Graph**: The Edge-List graph shows minimal timing differences in small mazes (e.g., 5x5, 7x5, 10x5). The operations are swift because the number of edges is small, and the overhead of traversing the edge list is negligible. For instance, operations like *updateWall()* often find the target edge quickly, contributing to the low times observed.
- **Incidence Matrix Graph**: Similarly, the Incidence Matrix graph performs well in small mazes, but with slightly higher overhead. This is because the matrix must still check the connections between vertices, even if the graph is sparse. The need to index into the incidence matrix for each vertex slightly increases the time, but the difference is not significant at this scale.

=> For small graphs, the differences in performance between Edge-List and Incidence Matrix are minimal because the problem size is small enough that the inherent complexities of each data structure do not have a large impact. The slight overhead in the Incidence Matrix comes from the matrix operations, whereas the Edge List benefits from quick iteration through a small list.

2. Performance in Medium Mazes:

- **Edge-List Graph**: As the size of the maze increases, the time for operations grows noticeably. For example, in a 15x15 maze, the Edge-List graph shows increased operation time, especially for the *neighbours()* function, which needs to check more edges as the graph grows denser. The quadratic growth in the number of edges begins to impact performance, resulting in higher timings.
- **Incidence Matrix Graph**: The Incidence Matrix, on the other hand, starts to demonstrate its strengths at this size. The matrix structure allows for quicker determination of connections between vertices, especially when dealing with a moderate number of edges. This advantage is reflected in the timings, where the Incidence Matrix often outperforms the Edge List, particularly in configurations with larger numbers of rows or columns.

=> The difference in performance here is primarily due to how each data structure handles an increase in complexity. The Edge-List's linear search through edges becomes more costly as the number of edges grows, while the Incidence Matrix's structured approach allows for more efficient queries. The theoretical time complexities support these observations: as the edge count grows, the Edge-List's $\Theta(E)$ complexity for operations becomes a limiting factor, whereas the matrix's performance remains more stable.

3. Performance in Large Mazes:

- **Edge-List Graph**: In large mazes (e.g., 25x25, 30x20), the Edge-List graph's operation times increase substantially. The quadratic growth in edges becomes evident as the *neighbours()* function must check a significantly larger number of edges. This leads to slower performance, particularly in dense graphs where each vertex is connected to many others.
- **Incidence Matrix Graph**: The Incidence Matrix continues to perform better than the Edge-List in these larger mazes. Its ability to quickly assess connections between vertices without needing to traverse a list of edges provides a clear advantage. The matrix's ability to handle dense graphs more efficiently is reflected in lower timings, even as the problem size grows.

=> The difference in performance at this level is due to the Edge-List's need to iterate through an increasingly large list of edges. This contrasts with the Incidence Matrix, where the structure of the matrix allows for more direct access to vertex connections. The matrix's ability to handle dense graphs is crucial here, as it avoids the costly operations associated with iterating over a long list of edges, reflected in the lower observed timings.

4. Performance in Extra Large Mazes:

- **Edge-List Graph**: In the largest maze sizes (e.g., 40x40, 25x40), the Edge-List graph struggles significantly. The time required for operations like *updateWall()* and *neighbours()* grows dramatically, particularly in dense graphs. The linear search through edges, combined with the quadratic growth in the number of edges, leads to very high operation times.
- **Incidence Matrix Graph**: The Incidence Matrix maintains its advantage in these large configurations, though the operation times also increase. However, the matrix is better suited to handle a large number of connections in these dense graphs, leading to less severe timing increases compared to the Edge List.

=> The extreme performance differences observed here are a direct result of the Edge-List's inefficiencies in handling dense, large-scale graphs. The Θ(E) complexity becomes particularly costly when E grows very large, as seen in these configurations. Conversely, the Incidence Matrix's ability to efficiently manage and query connections between vertices, even in large and dense graphs, provides a significant performance advantage, as supported by its Θ(V*E) complexity in the worst case, but often better in practical scenarios.

IV. Summary & Recommendations

In summary, the comparative analysis of the edge-list and Incidence Matrix data structures for maze generation reveals distinct performance characteristics based on maze size and configuration. The edge-list approach exhibited superior efficiency in scenarios where the number of edges relative to vertices is lower, particularly when the maze dimensions were smaller or when the increase in rows and columns was unbalanced. This is consistent with the theoretical expectation that the edge-list structure, with its linear time complexity for edge additions, performs well in sparse graphs.

Conversely, the Incidence Matrix demonstrated better performance as the maze size increased, particularly in dense configurations where the number of edges grows significantly. The consistent performance of the Incidence Matrix across large and extra-large mazes is attributed to its efficient edge presence checks and updates, making it more suitable for dense graphs where many connections exist between vertices.

Based on the analysis, for smaller mazes or configurations with a lower edge-to-vertex ratio, I recommend using the edge-list data structure due to its simplicity and lower overhead. However, for larger mazes, particularly those with balanced or dense configurations (e.g., 25x25 or larger), the Incidence Matrix is the more suitable choice as it scales more effectively with increased complexity, offering better performance in managing numerous edges. This recommendation aligns with the observed runtime behavior and supports the use of the appropriate data structure to optimize computational efficiency based on maze dimensions.