**Time and Space Complexity in Graph Representations**

When discussing graph representations, the space and time complexity depend on how we store the graph (either as an **adjacency matrix** or an **adjacency list**). Below is an explanation of these representations and their respective time and space complexities:

**Graph Representations:**

1. **Adjacency Matrix**
2. **Adjacency List**

**1. Adjacency Matrix**

An **adjacency matrix** is a 2D array (or matrix) that represents a graph. The rows and columns represent vertices, and the entries in the matrix indicate whether there is an edge between two vertices.

* **Matrix Dimensions**: If a graph has n vertices, the adjacency matrix is an n x n matrix.
* **Matrix Entry**: If matrix[i][j] == 1, it means there is an edge between vertex i and vertex j. If the entry is 0, there is no edge between them.
* **For weighted graphs**: Instead of 1 and 0, the matrix stores the weight of the edge if one exists. If no edge exists, it may store ∞ (infinity) or some other marker.

**Space Complexity:**

* The space complexity is **O(n²)** because we have an n x n matrix, where n is the number of vertices in the graph.

**Time Complexity:**

* **Checking if an edge exists** between two vertices: O(1), since you can access the matrix entry directly.
* **Adding or removing an edge**: O(1) (modifying an entry in the matrix).
* **Traversing all neighbors of a vertex**: O(n), as you need to check the entire row corresponding to the vertex.

**Use Cases:**

* Best for **dense graphs**, where the number of edges is close to n² (as there are a lot of edges).
* Not efficient for sparse graphs, as it uses a lot of memory even when the number of edges is small.

**2. Adjacency List**

An **adjacency list** is a more memory-efficient way to represent a graph. For each vertex, you store a list (or linked list, or other data structure) of adjacent vertices (i.e., those connected by an edge).

* **For each vertex**: You store a list of vertices that it is connected to by an edge.
* **For weighted graphs**: Each entry in the list stores the adjacent vertex and the weight of the edge.

**Space Complexity:**

* The space complexity is **O(n + e)**, where n is the number of vertices and e is the number of edges. This is because you store one list per vertex and one entry per edge.

**Time Complexity:**

* **Checking if an edge exists**: O(n) in the worst case, because you may need to traverse the entire list of adjacent vertices for the given vertex.
* **Adding an edge**: O(1), as you can directly append to the adjacency list of the source vertex.
* **Removing an edge**: O(n), since you may have to traverse the adjacency list to find and remove the target vertex.
* **Traversing all neighbors of a vertex**: O(degree(v)), where v is the vertex, because you only traverse the vertices directly connected to v (its neighbors).

**Use Cases:**

* Best for **sparse graphs**, where the number of edges is small compared to the number of vertices.
* More memory-efficient for sparse graphs.

**Examples of Graphs**

Now let's consider examples of different types of graphs, and how they would be represented using the adjacency matrix and adjacency list.

**1. A Graph with No Weights or Directions (Undirected, Unweighted Graph)**

Example: A simple undirected graph with 3 vertices, where vertex 1 is connected to vertex 2 and vertex 2 is connected to vertex 3.

* **Adjacency Matrix**:

Copy code

0 1 0

1 0 1

0 1 0

* **Adjacency List**:
  + Vertex 1: [2]
  + Vertex 2: [1, 3]
  + Vertex 3: [2]

In this case, the graph is **undirected**, so the edges are bidirectional (i.e., an edge from vertex 1 to vertex 2 is the same as an edge from vertex 2 to vertex 1).

**2. A Graph with Weights but No Directions (Undirected, Weighted Graph)**

Example: A graph with 3 vertices, where vertex 1 is connected to vertex 2 with a weight of 5, and vertex 2 is connected to vertex 3 with a weight of 10.

* **Adjacency Matrix**:

Copy code

0 5 0

5 0 10

0 10 0

* **Adjacency List**:
  + Vertex 1: [(2, 5)]
  + Vertex 2: [(1, 5), (3, 10)]
  + Vertex 3: [(2, 10)]

The graph is **undirected**, so each edge appears twice (one for each direction).

**3. A Graph with Weights and Directions (Directed, Weighted Graph)**

Example: A graph with 3 vertices, where vertex 1 is connected to vertex 2 with a weight of 5, and vertex 2 is connected to vertex 3 with a weight of 10, and vertex 3 is connected to vertex 1 with a weight of 7.

* **Adjacency Matrix**:

Copy code

0 5 0

0 0 10

7 0 0

* **Adjacency List**:
  + Vertex 1: [(2, 5)]
  + Vertex 2: [(3, 10)]
  + Vertex 3: [(1, 7)]

Here, the graph is **directed**, so the edges are not bidirectional (i.e., an edge from vertex 1 to vertex 2 is not the same as an edge from vertex 2 to vertex 1).

**Comparison: Adjacency Matrix vs. Adjacency List**

| **Aspect** | **Adjacency Matrix** | **Adjacency List** |
| --- | --- | --- |
| **Space Complexity** | O(n²) | O(n + e) |
| **Time Complexity (Insert)** | O(1) | O(1) |
| **Time Complexity (Search for an edge)** | O(1) | O(n) in worst case |
| **Time Complexity (Iterate over neighbors)** | O(n) | O(degree(v)) |
| **Best Use Case** | Dense graphs | Sparse graphs |

**Summary of Different Graph Types**

1. **No Weights, No Directions**: The graph is **undirected** and **unweighted**. Both the adjacency matrix and list work well here, but if the graph is dense, an adjacency matrix is preferable.
2. **Weights but No Directions**: The graph is **undirected** and **weighted**. Here, an adjacency list is often more space-efficient for sparse graphs.
3. **Weights and Directions**: The graph is **directed** and **weighted**. For both dense and sparse graphs, the adjacency list is more space-efficient, but it might take longer to search for an edge.