2. Graph Representations

A graph is an abstract concept. To store one in a computer's memory, we must translate this idea of vertices and edges into a concrete data structure. The choice of representation is critical, as it dictates the performance and space efficiency of our graph algorithms.

A good representation must efficiently answer two fundamental questions:

1. "**Is vertex `u` connected to vertex `v`?**"
2. "**What are all the neighbors of vertex `u`?**"

The two most common methods to solve this are the Adjacency Matrix and the Adjacency List.

2.1. Adjacency Matrix

An **Adjacency Matrix** is a 2D array, typically of size `V x V`, where `V` is the number of vertices. Each cell `adj[u][v]` in the matrix stores information about the edge between vertex `u` and vertex `v`.

2.1.1. Concept

Imagine a flight chart. The rows represent the "source" vertex and the columns represent the "destination" vertex.

- For an **unweighted graph**, we store a `1` (or `true`) at `adj[u][v]` if an edge exists, and `0` (or `false`) if it does not.
- For a **weighted graph**, instead of storing `0` or `1`, we store the weight of the edge.
- For an **undirected graph**, the matrix will be symmetric about the diagonal (i.e., `adj[u][v] == adj[v][u]`).
- For a **directed graph**, the matrix is not necessarily symmetric. `matrix[u][v]` can be `1` while `matrix[v][u]` is `0`

2.1.2. Example

Consider this simple undirected graph with 4 vertices:

```
(0) --- (1)
| \       |
|  \      |
```

```
|   \     |
|    \    |
|     \   |
|      \  |
|       \ |
(3) --- (2)
```

Edges: (0,1), (0,2), (0,3), (1,2), (2,3)

The **Adjacency Matrix** for this graph would be:

```
  0 1 2 3
0[0,1,1,1]
1[1,0,1,0]
2[1,1,0,1]
3[1,0,1,0]
```

2.1.3 Implementation (C++)

In C++, you would implement this using a `vector` of `vector`s.

```
// Assume V is the number of vertices
int V = 4;

// 1. Initialize a V x V matrix with all zeros
std::vector<std::vector<int>> adjMatrix(
    V,
    std::vector<int>(V, 0)
);

// 2. Add an edge (e.g., between 0 and 2)
void addEdge(int u, int v) {
    adjMatrix[u][v] = 1;
    adjMatrix[v][u] = 1; // For undirected
}

// 3. To check for an edge (e.g., between 1 and 3):
bool hasEdge = (adjMatrix[1][3] == 1); // false

// 4. To find all neighbors of a vertex (e.g., vertex 0):
```

```
for (int j = 0; j < V; ++j) {
    if (adjMatrix[0][j] == 1) {
        // j is a neighbor of 0
        // This will find j=1, j=2, and j=3
    }
}
```

2.1.4. Pros and Cons

Pros:

- **Fast Edge Lookup:** Checking if an edge `(u, v)` exists is `O(1)`. You just access the `adjMatrix[u][v]` cell.
- **Simple to Implement:** The logic is straightforward, as it's just a 2D array lookup.

Cons:

- **Space Inefficient:** This is the biggest drawback. The matrix always requires `O(V^2)` space, regardless of how many edges the graph has.
- **Slow Neighbor Iteration:** Finding all neighbors of a vertex requires iterating through its entire row, which is an `O(V)` operation, even if the vertex only has one neighbor.

2.2. Adjacency List

An **Adjacency List** is an array (or `std::vector`) of lists. The main array has `V` entries, where each entry `adj[u]` corresponds to vertex `u`. The entry `adj[u]` then holds a list of all other vertices `v` that are neighbors of `u`.

2.2.1. Concept

Imagine a phone's contacts list. The main array (of size `V`) acts like your address book index. Each entry in this index (e.g., `adj[u]`) doesn't hold a single value, but rather points to a list of all that vertex's direct neighbors.

- For an **unweighted graph**, the list for `adj[u]` simply stores the indices (like `v1`, `v2`) of all its neighbors.
- For a **weighted graph**, the list for `adj[u]` stores pairs, where each pair contains the neighbor's index and the edge's weight (e.g., `{v1, w1}`, `{v2, w2}`).
- For an **undirected graph**, when you add an edge `(u, v)`, you must add `v` to `adj[u]`'s list and add `u` to `adj[v]`'s list.
- For a **directed graph**, When you add an edge `(u, v)`, you only add `v` to `adj[u]`'s list.

2.2.2. Example

Using the same 4-vertex graph:

```
(0) --- (1)
| \       |
|  \      |
|   \     |
|    \    |
|     \   |
|      \  |
|       \ |
(3) --- (2)
```

The **Adjacency List** for this graph would be:

```
0: -> [1, 2, 3]
1: -> [0, 2]
2: -> [0, 1, 3]
3: -> [0, 2]
```

2.2.3. Implementation (C++)

In C++, you would implement this using a `vector` of `list`s (or a `vector` of `vector`s).

```
// Assume V is the number of vertices
int V = 4;

// 1. Initialize a vector of size V. Each element
//    is an empty list.
std::vector<std::list<int>> adjList(V);

// 2. Add an edge (e.g., between 0 and 2)
void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u); // For undirected
}

// 3. To check for an edge (e.g., between 1 and 3):
```

```
// This is slower than the matrix.
bool hasEdge = false;
for (auto& neighbor : adjList[1]) {
    if (neighbor == 3) {
        hasEdge = true;
        break;
    }
} // hasEdge is false

// 4. To find all neighbors of a vertex (e.g., vertex 0):
// This is extremely fast.
for (auto& neighbor : adjList[0]) {
    // neighbor is a neighbor of 0
    // This will find neighbor=1, neighbor=2, and neighbor=3
}
```

2.2.4. Handling Weighted Graphs

To store weights, the list cannot just be of `int`s. It must store pairs (or a custom `struct`).

- `std::vector<std::list<std::pair<int, int>>> adjList(V);`
- The `pair<int, int>` would store `{neighbor, weight}`.
- `addEdge` would look like: `adjList[u].push_back({v, weight});`

2.3.5. Pros and Cons

Pros:

- **Space Efficient:** This is the primary advantage. The total space required is `O(V + E)`.
- **Fast Neighbor Iteration:** Finding all neighbors of a vertex u is `O(deg(u))` (where `deg(u)` is the degree, or number of neighbors). This is optimally fast.

Cons:

- **Slow Edge Lookup:** Checking if a specific edge `(u, v)` exists is `O(deg(u))` in the worst case, because we must iterate through the list `adj[u]` to see if `v` is in it.

2.3. Comparison: Matrix vs. List

Operation	Adjacency Matrix	Adjacency List
Space	$O(V^2)$	$O(V + E)$

Operation	Adjacency Matrix	Adjacency List
Add Edge	`O(1)`	`O(1)`
Check Edge (u, v)	`O(1)` (Fast)	`O(deg(u))` (Slow)
Find Neighbors of u	`O(V)` (Slow)	`O(deg(u))` (Fast)
Remove Edge (u, v)	`O(1)`	`O(deg(u))`
Best For	**Dense Graphs** (where `E` ≈ `V²`)	**Sparse Graphs** (most common case)

Revision #3
Created 2025-11-11 11:22:10 UTC by RE
Updated 2025-11-11 11:53:43 UTC by RE