Graph Representations

Graphs can be represented in multiple ways in memory. The choice affects **time complexity**, **space efficiency**, and **ease of implementation** for algorithms like DFS, BFS, Dijkstra, etc.

1. Adjacency Matrix

An **Adjacency Matrix** is a `V × V` 2D array where each cell `(i, j)` indicates the presence (and possibly weight) of an edge from vertex `i` to `j`.

Unweighted Graph (Undirected)

Graph:

```
A -- B -- C
```

Index Map:
A → 0, B → 1, C → 2

Adjacency Matrix:

```
   A B C
  --------
A | 0 1 0
B | 1 0 1
C | 0 1 0
```

Each 1 means an edge exists between those vertices. Since the graph is undirected, the matrix is symmetric.

Weighted Graph (Directed)

Graph:

```
A --(5)--> B --(2)--> C
```

Matrix:

```
   A B C
  --------
A | 0 5 0
B | 0 0 2
C | 0 0 0
```

Weights represent the cost of moving from one vertex to another. No symmetry here as it's a **directed graph.**

Pros:

- Easy to implement
- Constant time edge lookup: $O(1)$

Cons:

- High space usage: $O(V^2)$
- Inefficient for sparse graphs

2. Adjacency List

An **Adjacency List** uses an array of lists where each index stores neighbors of the vertex.

Unweighted Graph (Undirected)

Graph:

```
A -- B -- C
```

Adjacency List:

```
A → B
B → A, C
C → B
```

Here, each vertex lists all directly connected vertices.

Weighted Graph (Directed)

Graph:

```
A --(5)--> B --(2)--> C
```

Adjacency List:

```
A → (B, 5)
B → (C, 2)
C → –
```

Each vertex stores a list of pairs: (neighbor, weight)

Pros:

- Space-efficient: `O(V + E)`
- Easy neighbor traversal

Cons:

- Slower edge lookup: `O(k)` (k = neighbors)

3. Edge List

An **Edge List** stores all edges as a list of pairs (unweighted) or triplets (weighted).

Unweighted Graph:

Graph:

```
A -- B -- C
```

Edge List:

```
(A, B)
(B, C)
```

Weighted Graph:

Graph:

```
A --(5)--> B --(2)--> C
```

Edge List:

```
(A, B, 5)
(B, C, 2)
```

Pros:

- Very space-efficient for sparse graphs
- Great for sorting edges (e.g., Kruskal's Algorithm)

Cons:

- Not efficient for neighbor lookup or traversal

Coding Implementation

1. Adjacency Matrix (Undirected & Weighted Directed)

```
#include <iostream>
using namespace std;

const int V = 3; // A, B, C

void printMatrix(int graph[V][V]) {
    cout << "Adjacency Matrix:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++)
```

```
            cout << graph[i][j] << " ";
        cout << endl;
    }
}

int main() {
    // Example: A — B — C (Unweighted Undirected)
    int graph1[V][V] = {0};

    graph1[0][1] = 1; // A–B
    graph1[1][0] = 1;
    graph1[1][2] = 1; // B–C
    graph1[2][1] = 1;

    cout << "Unweighted Undirected Graph:\n";
    printMatrix(graph1);

    // Example: A → B (5), B → C (2) (Weighted Directed)
    int graph2[V][V] = {0};
    graph2[0][1] = 5;
    graph2[1][2] = 2;

    cout << "\nWeighted Directed Graph:\n";
    printMatrix(graph2);

    return 0;
}
```

2. Adjacency List (Unweighted & Weighted)

```
#include <iostream>
#include <vector>
using namespace std;

const int V = 3;

void printUnweightedList(vector<int> adj[]) {
    cout << "Adjacency List (Unweighted):\n";
    for (int i = 0; i < V; i++) {
        cout << i << " → ";
        for (int x : adj[i])
            cout << x << " ";
        cout << endl;
    }
}

void printWeightedList(vector<pair<int, int>> adj[]) {
    cout << "\nAdjacency List (Weighted):\n";
    for (int i = 0; i < V; i++) {
        cout << i << " → ";
        for (auto p : adj[i])
            cout << "(" << p.first << ", " << p.second << ") ";
        cout << endl;
    }
}

int main() {
    // Unweighted Undirected Graph: A-B-C
    vector<int> unweighted[V];
    unweighted[0].push_back(1);
```

```
    unweighted[1].push_back(0);
    unweighted[1].push_back(2);
    unweighted[2].push_back(1);

    printUnweightedList(unweighted);

    // Weighted Directed Graph: A→B(5), B→C(2)
    vector<pair<int, int>> weighted[V];
    weighted[0].push_back({1, 5});
    weighted[1].push_back({2, 2});

    printWeightedList(weighted);

    return 0;
}
```

3. Edge List (Unweighted & Weighted)

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // Unweighted Edge List: A–B, B–C
    vector<pair<int, int>> edges;
    edges.push_back({0, 1}); // A–B
    edges.push_back({1, 2}); // B–C

    cout << "Edge List (Unweighted):\n";
    for (auto edge : edges)
        cout << edge.first << " -- " << edge.second << endl;
```

```
    // Weighted Edge List: A→B(5), B→C(2)
    vector<tuple<int, int, int>> weightedEdges;
    weightedEdges.push_back({0, 1, 5});
    weightedEdges.push_back({1, 2, 2});

    cout << "\nEdge List (Weighted):\n";
    for (auto edge : weightedEdges) {
        int u, v, w;
        tie(u, v, w) = edge;
        cout << u << " --(" << w << ")--> " << v << endl;
    }

    return 0;
}
```