TUF hits 100M views on YT
Last Massive sale before prices go up in V2 launch in Dec
00 DAYS : 13 HOURS : 00 MINUTES : 10 SECONDS
Click Here

Profile
Blogs
DSA Sheets
System Design
Striver's DSA Playlist
SDE Core Sheets
Saved Notes
CW Fullstack Final
Upgrade to Plus

Graph Representation in C++

156 2

In this article, we are going to study the different ways of representing a graph in memory, but before that first, let us understand how to take the input of the graph.

Input Format

In the question, they will mention whether it is a directed or undirected graph. The first line contains two space-separated integers n and m denoting the number of nodes and the number of edges respectively. Next m lines contain two integers u and v representing an edge between u and v. In the case of an undirected graph if there is an edge between u and v, it means there is an edge between v and u as well. Now the question arises if there is any boundation on the number of edges, i.e., the value of m? The answer is NO. If we add more edges, then the value of m will increase.

Graph Representations

After understanding the input format, let us try to understand how the graph can be stored. The two most commonly used representations for graphs are

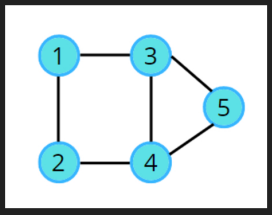
- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

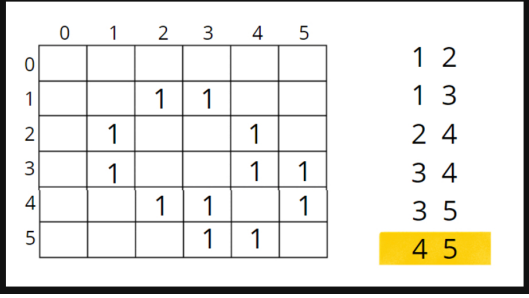
An adjacency matrix of a graph is a two-dimensional array of size n x n, where n is the number of nodes in the graph, with the property that a[i][j] = 1 if the edge (v_i, v_j) is in the set of edges, and a[i][j] = 0 if there is no such edge.

Consider the example of the following undirected graph,

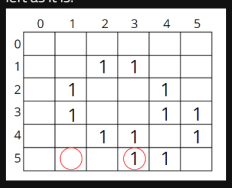
```
Input:
5 6
1 2
1 3
2 4
3 4
3 5
4 5

Explanation:
Number of nodes, n = 5
Number of edges, m = 6
Next m lines represent the edges.
```



We need to store these edges so that future algorithms can be performed. Are the nodes zero-based or one-based? In this case, the nodes follow one-based indexing as the last node is 5 and the total number of nodes is also 5. Now, define an adjacency matrix of size (n+1) x (n+1), i.e., adj[n+1][n+1]. If there is an edge between 1 and 2, mark 1 at (1,2) and (2,1) as there is an edge between 2 and 1 as well (in the case of an undirected graph). Similarly, follow for other edges.



All the edges are marked in the adjacency matrix, remaining spaces in the matrix are marked as zero or left as it is.



	0	1	2	3	4	5
0						
1			1	1		
2		1			1	
3		1			1	1
4			1	1		1
5				1	1	

This matrix will tell if there is an edge between two particular nodes. For example, there is an edge between 5 and 3 as 1 is at (5,3) but there is no edge between 5 and 1 as the space is empty (or can be filled with 0) at position (5,1) in the adjacency matrix.

The space needed to represent a graph using its adjacency matrix is n^2 locations. Space complexity = (n*n), It is a costly method as n^2 locations are consumed.

Code:

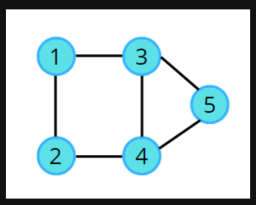
C++

```
using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;
    // adjacency matrix for undirected graph
    // time complexity: O(n)
    int adj[n+1][n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
```

Adjacency Lists

In the previous storing method, we saw it was taking n^2 space to store the graph, this is where the adjacency list comes into the picture, it takes a very less amount of space.

This is a node-based representation. In this representation, we associate with each node a list of nodes adjacent to it. Normally an array is used to store the nodes. The array provides random access to the adjacency list for any particular node.
Consider the example of the following **undirected graph**,



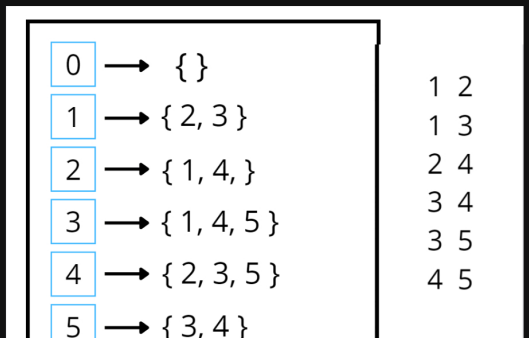
To create an adjacency list, we will create an array of size n+1 where n is the number of nodes. This array will contain a list, so in C++ list is nothing but the vector of integers.

```
vector <int> adj[n+1];
```

Now every index is containing an empty vector/ list. With respect to the example, 6 indexes contain empty vectors.

What is the motive of the list?

In the example, we can clearly see that node 4 has nodes 2, 3, and 5 as its adjacent neighbors. So, to store its immediate neighbors in any order, we use the list.



Hence, we stored all the neighbors in the particular indexes. In this representation, for an undirected graph, each edge data appears twice. For example, nodes 1 and 2 are adjacent hence node 2 appears in the list of node 1, and node 1 appears in the list of node 2. So, the space needed to represent an undirected graph using its adjacency list is 2 x E locations, where E denotes the number of edges.

Space complexity = O(2xE)

This representation is much better than the adjacency matrix, as matrix representation consumes n^2 locations, and most of them are unused.

Code:

C++

```
    int n, m;
    cin >> n >> m;
    // adjacency list for undirected graph
    // time complexity: O(2E)
    vector<int> adj[n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}
```

For **directed graphs**, if there is an edge between u and v it means the edge only goes from u to v, i.e., v is the neighbor of u, but vice versa is not true. The space needed to represent a directed graph using its adjacency list is E locations, where E denotes the number of edges, as here each edge data appears only once.

Space complexity = O(E)

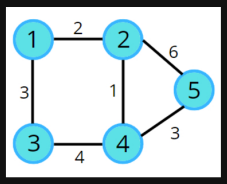
Code:

C++

```
    int n, m;
    cin >> n >> m;
    // adjacency list for directed graph
    // time complexity: O(E)
    vector<int> adj[n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        // u -> v
        cin >> u >> v;
        adj[u].push_back(v);
    }
    return 0;
}
```

Weighted Graph Representation

As of now, we were considering graphs with unit weight edges (i.e., if there is an edge between two nodes then the weight on the edge is unit weight), now what if there are weights on its edges as shown in the following example?



For the **adjacency matrix,** it is much simpler.

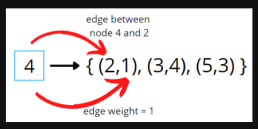
Undirected Graph	Directed Graph

int u, v, wt;
cin >> u >> v >> wt;
adj[u][v] = wt;
adj[v][u] = wt;
	int u, v, wt;
	cin >> u >> v >> wt;
	adj[u][v] = wt;

But how are we going to implement it in the **adjacency list**?

Earlier in the adjacency list, we were storing a list of integers in each index, but for weighted graphs, we will store pairs (node, edge weight) in it.

```
vector< pair <int,int> > adjList[n+1];
```



*Special thanks to **Vanshika Singh Gour** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*