Graph Representations

* 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 (vᵢ, vⱼ) is in the set of edges, and a[ i ][ j ] = 0 if there is no such edge.

If the nodes follow one-based indexing as the last node is n and the total number of nodes is also n. Now, define an adjacency matrix of size (n+1) x (n+1), i.e., adj[n+1][n+1].

The space needed to represent a graph using its adjacency matrix is n² locations. Space complexity = (n\*n), It is a costly method as n² locations are consumed.

Code :

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;

adj[v][u] = 1 // this statement will be removed in case of directed graph

}

Adjacency List

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.

To create an adjacency list, we will create an array of size n+1 where n is the number of nodes.

Now every index is containing an empty vector/ list. vector<int> adj[n+1];

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 n² locations, and most of them are unused.

Code :

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);

}

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)

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

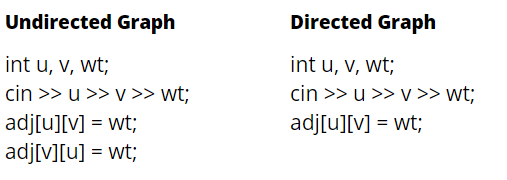
cin >> u >> v;

adj[u].push\_back(v);

}

For weighted graph:

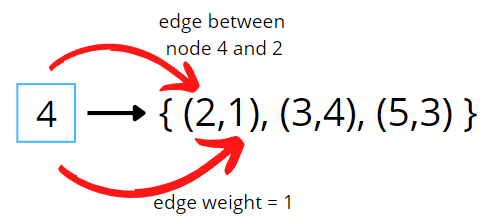
For the adjacency matrix, it is much simpler.



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];



DFS: