Graph Theory (Basic)

Programming Club
(2016-17)
Science and Technology Council
IIT Kanpur

Introduction

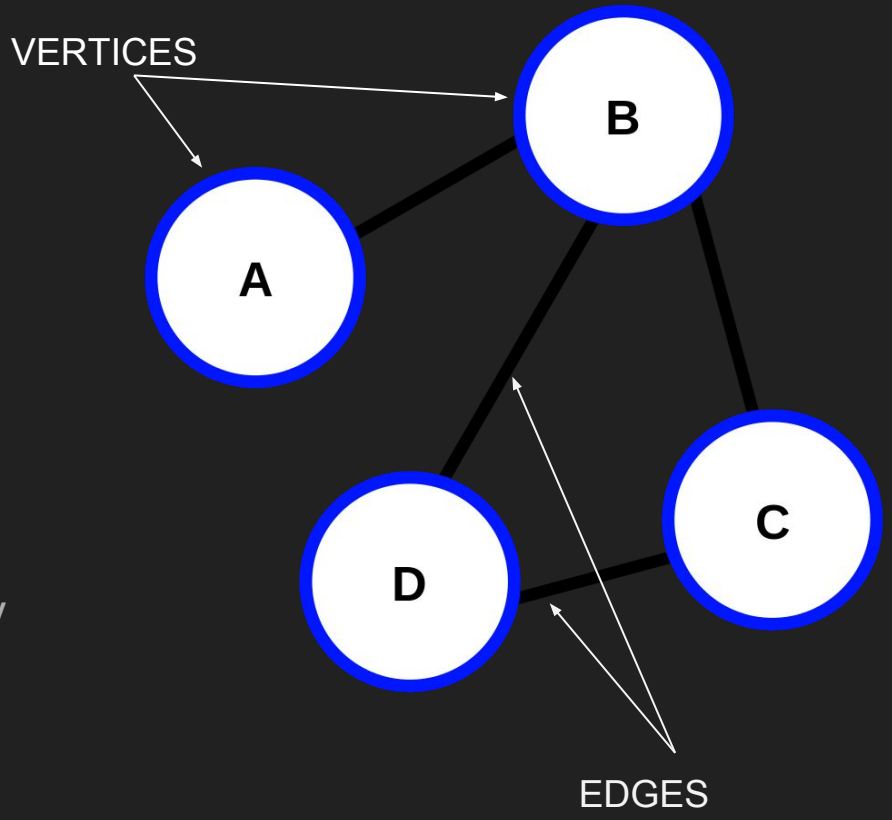
What is a Graph?

Anything that can be modelled as comprising of Vertices and Edges can be called a Graph.

Examples are:

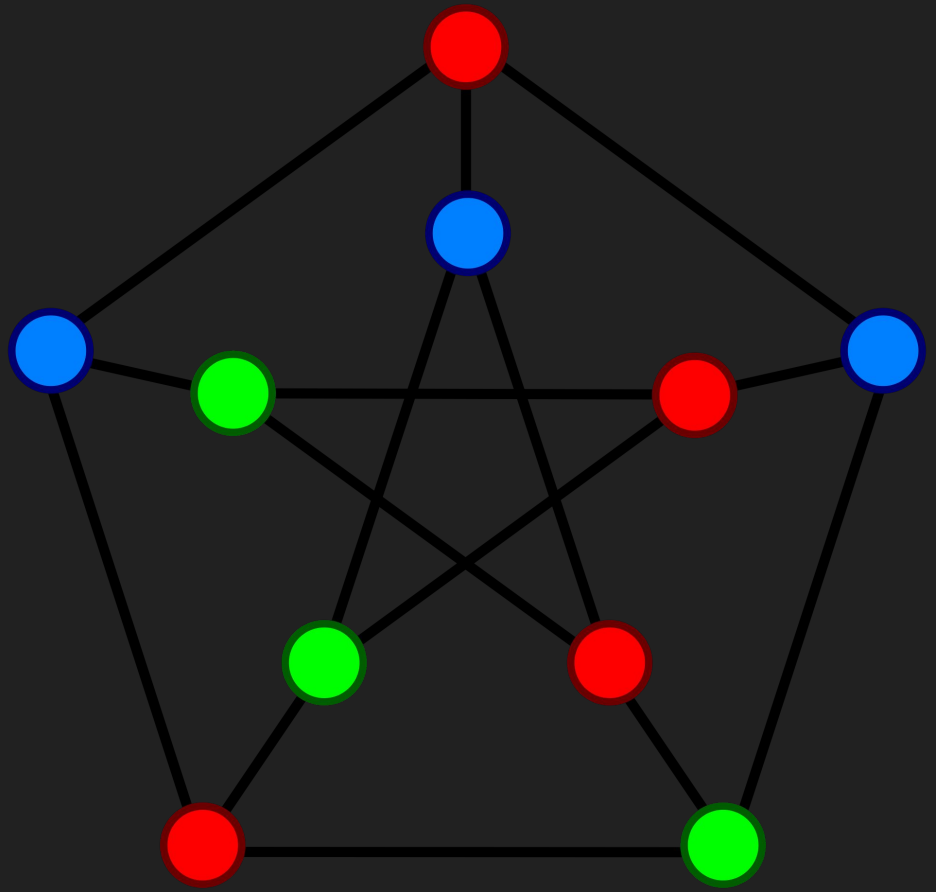
A number of cities*(vertices)* connected by roads(*edges*).

Computers(*vertices*) in a LAN Network connected by ethernet cables(*edges*).



Topics to be covered:

1. Types of Graphs
2. Representation of Graphs
   a. Adjacency Matrix Representation
   b. Adjacency List Representation
3. Graph Traversal Algorithms
   a. Depth First Search
   b. Breadth First Search
4. Special Graphs



Types of Graphs

1. Undirected Graphs : If there is an edge between u-v , it can be traversed both ways,i.e, from u to v and from v to u. Eg. A two way road.

2. Directed Graphs : Unlike undirected graphs an edge between u and v implies that one can go only from u to v and not the other way around. The edges are denoted with arrows like u->v. Eg.Following someone on a social networking website like Instagram. Following a person does not imply the other person follows you. Much like a directed Graph.

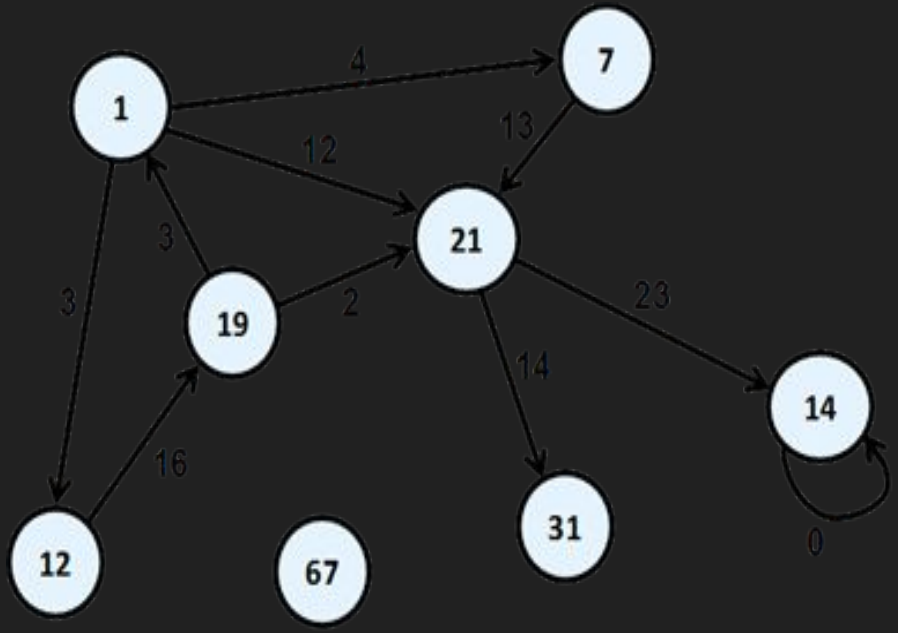
Types of Graphs

The weight of an edge refers the cost incurred to travel across it.Much like the length of a road,etc.This gives rise to 2 kinds of graphs:

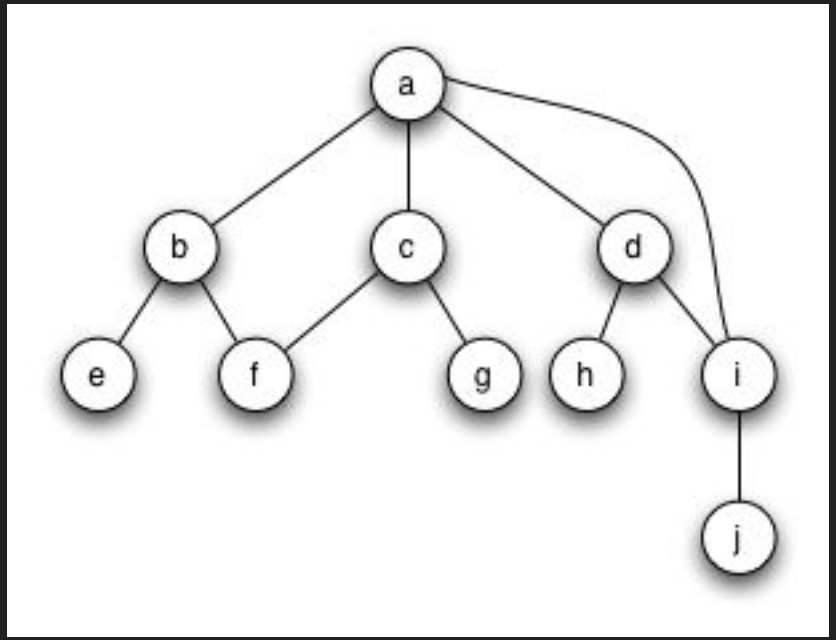
Unweighted Graphs:In these each edge is assigned a unit weight,i.e,all edges are equal with respect to weight.

Weighted Graphs:In these,each edge is assigned some weight.It may be positive or negative as per the situation.

In a contest the graph will be a combination of the two like a directed weighted graph or an unweighted undirected graph.



A WEIGHTED DIRECTED GRAPH



AN UNWEIGHTED UNDIRECTED GRAPH

Representation of Graphs

The symbols being used here will be used henceforward:

The Graph G has V vertices and E edges . An edge joining u to v is denoted by (u,v). For an undirected graph an edge (u,v) implies that we can move from u to v and v to u.

Degree of a vertex v denoted by Deg(v):Number of edges connected to a node.Note that for directed graphs there are two sets of degrees for a vertex-an Indegree and an Outdegree.

A graph can be represented using two methods:

1.Adjacency Matrix

2.Adjacency List

Adjacency Matrix

As the name suggests we will be storing the Graph in a matrix.

We will construct a *VxV* Matrix to store information about the edges of the graph.Let M denote the adjacency matrix. If M[u][v] is 1 it denotes that there is an edge going from u to v. Similiarly if it is 0 it denotes that there is no edge. Note that in an undirected graph M[u][v] = 1 implies that M[v][u] = 1. The space needed is $O(V^2)$.

Advantage:Easy Representation.More useful for smaller and denser(more edges) graphs.

Disadvantage: For Larger Graphs with number of edges much lesser than V^2 , the matrix is composed of a large number of 0s,leading to ineffiency in allocating memory and time taken in other algorithms (mentioned ahead).

Adjacency List

Unlike Adjacency Matrices , here we only keep information about the edges in the graph and not about those that aren't there. We maintain V linked lists , one for each vertex. If there is an edge (u,v) then we append v to u's linked list.The size of Adjacency List is O(E).
We can implement this using a vector of vectors. The code goes like this.

```
vector<vector<int> > AdjacencyList;
AdjacencyList.resize(V+5);
for(int i=1;i<=E;i++)
{
   //reading edges
   int u,v;
   scanf("%d %d",&u,&v);
   AdjacencyList[u].push_back(v);
   AdjacencyList[v].push_back(u);  //For undirected graphs
}
```

Handshaking Lemma

Before we start with Graph search algorithms , there is one lemma to go. The Handshaking lemma states that for an undirected graph ,

$$\sum_{v \in V} \deg(v) = 2|E|$$

Proof : This lemma is quite intuitive in itself. For a given node v , when we add deg (v) to the sum on the left we are adding the count of all edges connected to it. As we do this for all nodes, each edges is counted twice. Hence the sum is equal to 2E.

Graph Search Algorithms

The Problem:Given a graph G with V vertices and E edges , answer the following types of questions:

1. Given a vertex u and v , tell us whether we can reach v from u travlling across a sequence of edges.

2.Given a vertex v tell us how many vertices from the graph can it reach ,i.e, the number of vertices in its connected component.

3.Given a graph, tell us how many connected components does it have.

We can solve these using Depth First Search (DFS) and Breadth First Search (BFS).

Depth First Search (DFS)

DFS is the more intuitive of the two search algorithms. It goes like this:

```
DFS(v)
        Mark v as visited
        For every neighbour u of v
                If u is unvisited
                        DFS(u)
```

This algorithm finds all the vertices that can be reached from the initial search vertex.

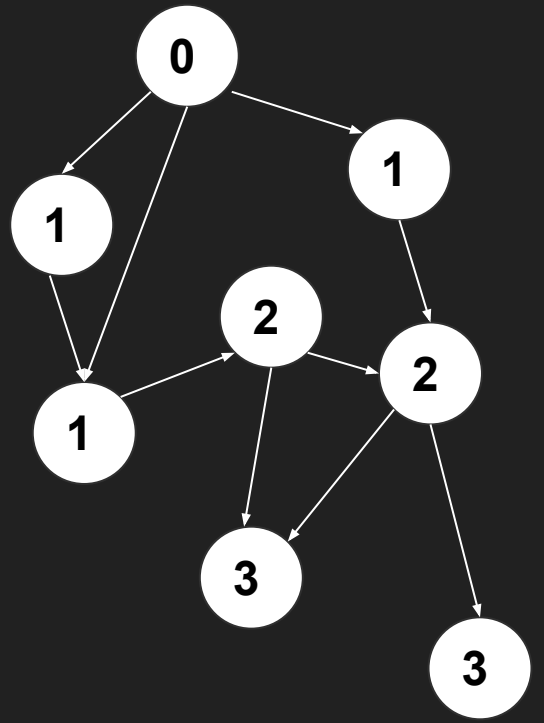
Time Complexity:As we visit each vertex once and travel over each edge once the complexity is O (V+E). (Note that the complexity is O(V+E) only if we use Adjacency Lists.It is $O(V^2)$ if we use Adjacency Matrices.)

Code For DFS

```
vector<int> adj[1000];
vector<bool> visited(1000, false);
void dfs(int u) {
visited[u] = true;
for (int i = 0; i < adj[u].size(); i++) {
                int v = adj[u][i];
                if (visited[v]) {
                        continue;
                        }
                dfs(v);
                }
        }
```

Breadth First Search (BFS)

In BFS , we traverse the graph in a different manner. Assuming we have started the search from a source s, let us a new term:Layer. A vertex belongs to a layer i if it can reach the source in a minimum of i steps and no less. Hence , the Source belongs to layer 0.All its neighbours belong to layer 1. And so on. We process the graph layer by layer. To do this we use a queue. First we add the source to the queue and mark it as Layer 0. Then we do this repeatedly. We pop off the front of the queue.Let me call this vertex u.For all unvisited neighbours of u we mark them as Layer[u]+1 and push them to the back of the queue. When the queue is empty we have processed all vertices that could be reached from u. Complexity: O (V+E), same as DFS.



Showing the layers found after BFS

BFS Code

```
vector<int> adj[1000];
vector<int> Layer[1000];
vector<bool> visited(1000, false);
queue<int> Q;
Layer[start]=0;
Q.push(start);
visited[start] = true;
while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = 0; i < adj[u].size(); i++) {
                int v = adj[u][i];
                f (!visited[v]) {
                        Q.push(v);Layer[v]=Layer[u]+1;
                        visited[v] = true;
                        }
                }
        }
```

Applications

1.To Find the number of Connected Components of a graph:

```
Mark all vertices as unvisited
CountComponents=0
For i from 1 to n
    If i is unvisited
        DFS(i)
        CountComponents++
```

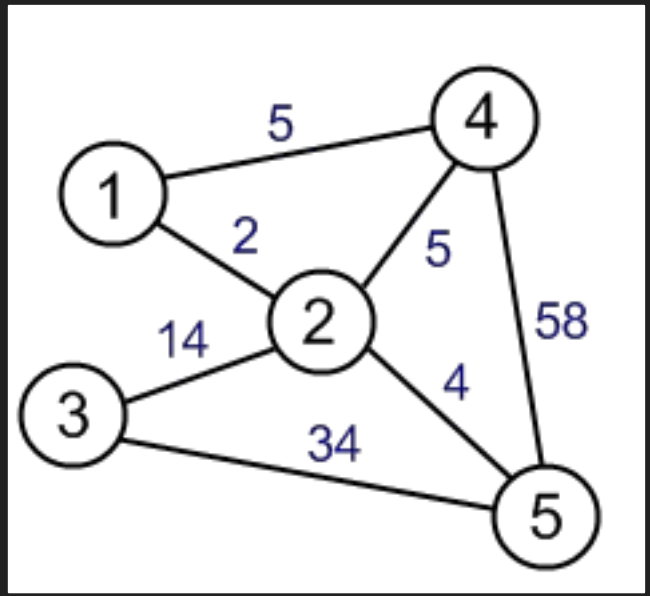
2. To find the shortest path between two vertices in an unweighted graph:

If one observes carefully,for a given source S, layer[u] is nothing but the shortest number of edges to travel from S to u ,i.e,the shortest path in an unweighted graph from S to u. So to find the shortest path in an unweighted graph (or with all edge weights equal to 1) one can use BFS.

3.BFS/DFS are often used as subroutines for more complex algorithms .

Dijkstra’s Algorithm

We saw that BFS can be used to find single source shortest paths when the graph is unweighted (or all edge weights are equal to 1). However when the weights can be different then we have to resort to a different algorithm . It turns out that when the edge weights are non-negative we can find the shortest path from a vertex to another using a rather simple and intuitive “greedy” algorithm proposed by E. Dijkstra. When implemented without any data structure it has a complexity of $O(n^2)$ but when implemented using a Priority Queue it has a complexity of O(ElogV).
When the graph is small and dense it is advisable to use the $O(n^2)$ version while when it is large and less dense it is better to use the O(ElogV) version.



Dijkstra's Algorithm

A few notes regarding the pseudocode on the right:

1. H is a Priority Queue. Basically the usage of H in this algorithm is to update the distance values of the vertices , report the minimum and delete the minimum.
2. We have used the adjacency list representation in this pseudocode.The complexity is O(ElogV). Try to come up with the $O(n^2)$ one!

Input: Graph $G = (V, E)$, directed or undirected; positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$

Output: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u.

procedure DIJKSTRA(G, l, s)

```
for all u ∈ V do
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = MAKEQUEUE(V)                          ▷ using dist-values as keys
while H is not empty do
    u = DELETEMIN(H)
    for all edges (u, v) ∈ E do
        if dist(v) > dist(u) + l(u, v) then
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            DECREASEKEY(H, v)
```

Practice problems in Graph Theory

Being one of the most common topics in competitive programming there are a lot of graph problems in various online judges.We recommend Codeforces before switching to any other online judge because : There is a gradual difficulty gradient in the problems which is good for a beginner and also tutorials for every problem . So here are the problems we suggest you solve:

1. http://www.codeforces.com/problemset/problem/500/A (simple BFS)
2. http://codeforces.com/problemset/problem/330/B (graph construction)
3. http://codeforces.com/problemset/problem/20/C (straightforward Dijkstra)
4. http://codeforces.com/problemset/problem/320/B (DFS)
5. http://codeforces.com/problemset/problem/369/C (DFS : need to compute additional information)
6. http://codeforces.com/problemset/problem/475/B (BFS/DFS on a grid)

There are many additional topics in graph theory and many other data structures that help in solving graph problems.
A list of some topics you may choose to study next:

1. Disjoint Set Union (also known as union find data structure)
2. Minimum Spanning Tree (Kruskal's and Prim's algorithm)
3. Floyd-Warshall algorithm (All Pairs Shortest Path Algorithm)
4. Bridges/Articulation Points .

Further Reading and Practice

The study of Graphs and their algorithms is far from complete as so we recommend these links to read up more about what we learnt today and more:

1.https://www.topcoder.com/community/data-science/data-science-tutorials/introduction-to-graphs-and-their-data-structures-section-2/

2.This contains the codes we used today and more comprehensive visualization of the Algorithms: https://algo.is/aflv16/aflv_07_graphs_1.pdf

3.More advanced Graph Algorithms (with code and links):http://codeforces.com/blog/entry/16221

4.You can practice on http://codeforces.com/problemset/tags/graphs?order=BY_SOLVED_DESC

5.Reading the chapter on graph theory from CLRS is highly advised. The theory is introduced and proved extensively and is an enlightening experience.

6.The Coursera Algorithms course (I covers data stuctures and II covers many graph algorithms) is a good course for beginners.

Thank You.