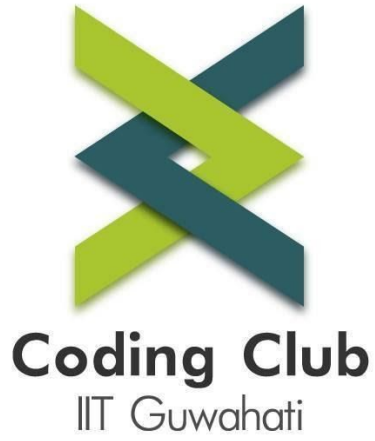


CRACKING THE PLACEMENT TESTS

A WEEKLY GUIDE CONTAINING TOPICS, QUESTIONS AND TRICKS COMMONLY ENCOUNTERED IN PLACEMENT TESTS

JULY 25, 2020
CODING CLUB, IIT GUWAHATI

Coding Club
IIT Guwahati

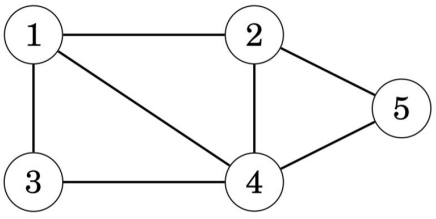
WEEK 5

GRAPHS

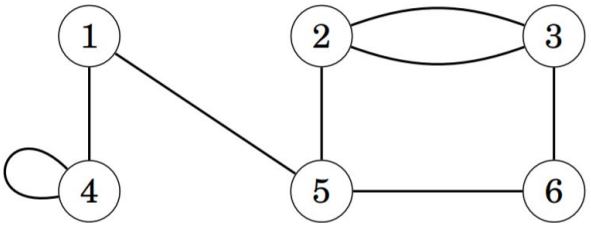
Many programming problems can be solved by modelling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. In this lecture, we will look upon its various aspects starting with the terms and definitions.

GRAPH TERMINOLOGY

- A **graph** consists of **nodes** and **edges**. Generally, **V** denotes the set of nodes in a graph, and **E** denotes the set of edges.
- A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often, we assume that graphs are simple.

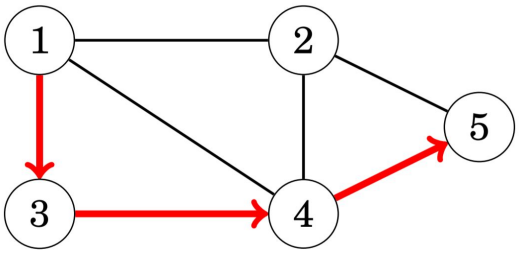


A Simple Graph



A Graph which is **NOT** simple.

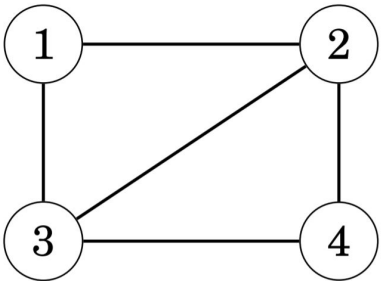
- A **path** leads from node *a* to node *b* through edges of the graph. The **length** of a path is the number of edges in it.
- A path is a **cycle** if the first and last node is the same. A path is **simple** if each node appears at most once in the path.



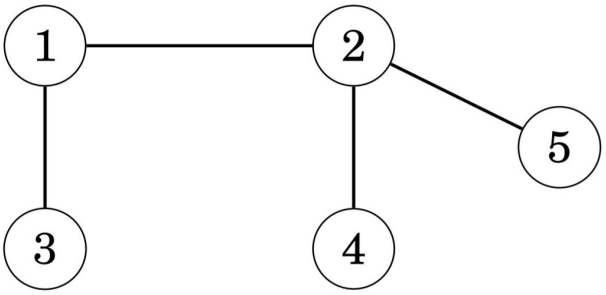
A Simple Path

WEEK 5

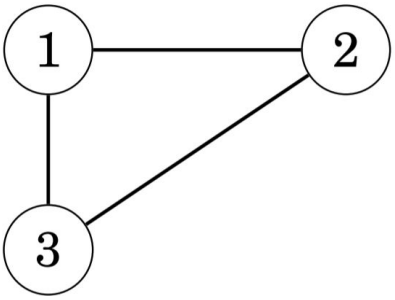
- A graph is **connected** if there is a path between any two nodes.
- The connected parts of a graph are called its **components**.
- A **tree** is a connected graph that consists of *n* nodes and *n −1* edges. There is a **unique path** between any two nodes of a tree.

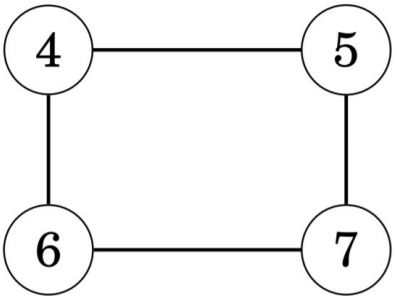


A connected Graph



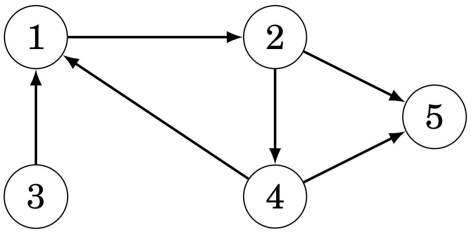
A Tree



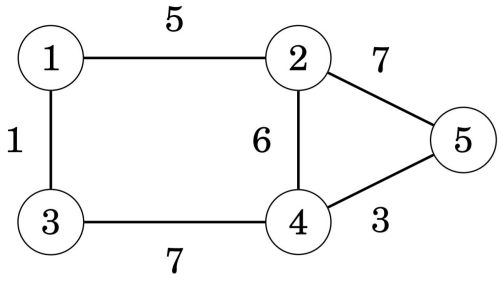


A Graph with three components: {1, 2, 3}, {4, 5, 6, 7} and {8}.

- A graph is **directed** if the edges can be traversed in one direction only. A graph is **undirected** if the edges can be traversed in both directions.
- In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths.



A Directed Graph



A weighted Graph

- Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors.
- A graph is **regular** if the degree of every node is a **constant** *d*. A graph is **complete** if the degree of every node is *n −1*, i.e., the graph contains all possible edges between the nodes.
- In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node.

Now we move on to implementing graphs in our code.

GRAPH REPRESENTATION AND IMPLEMENTATION

- **Adjacency list representation**: Most common form of graph/tree implementation. Optimal when Graph is **sparse**, i.e., number of edges is considerably less**.** Here, each node *x* in the graph is assigned an **adjacency lis**t that consists of nodes to which there is an edge from *x.*

Initiation (Graph of N nodes):

```
1. vector<int> adj[N];
```

Adding edge *u* -> *v* (In undirected graphs add both *u* -> *v* and *v* -> *u*):

```
1. adj[u].push_back(v);
```

Initiation of Weighted Graph:

```
1. vector< pair<int,int> > adj[N];
```

Adding edge *u* -> *v* with weight *w*:

```
1. adj[u].push_back({v,w});
```

Accessing adjacency list of *u*:

```
1. for(auto v : adj[u])
2. { //process v }
```

- **Adjacency matrix representation**: Optimal when Graph is **dense**, i.e., number of edges is close to the total possible number of edges**.** An **adjacency matrix** is a two-dimensional array that indicates which edges the graph contains.

Initiation (integers instead of booleans for weighted graphs):

```
1. bool adj[N][N];
2. for(int i=0;i<N;i++) for(int j=0;j<N;j++) adj[i][j]=false;
```

Adding edge *u -> v* (In undirected graphs add both *u -> v* and *v -> u*):

```
1. adj[u][v]=true;
```

- **Edge list representation** : An **edge list** contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node.

Initiation (Graph of N nodes):

```
1. vector<pair<int,int>> edges;
```

Adding edge *u -> v*:

```
1. edges.push_back({u,v});
```

Initiation of Weighted Graph:

```
1. vector<tuple<int,int,int>> edges;
```

Adding edge *u -> v* with weight *w*:

```
1. edges.push_back({u,v,w});
```

Coding Club
IIT Guwahati

WEEK 5

DEPTH-FIRST SEARCH (DFS)

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

Implementation:

```
1. vector<int> adj[N];
2. bool visited[N];
3.
4. void dfs(int s) {
5.     if (visited[s]) return;
6.     visited[s] = true;
7.     // process node s
8.     for (auto u: adj[s]) {
9.         dfs(u);
10.      }
11.  }
```

Time Complexity: O(V + E)

BREADTH-FIRST SEARCH (BFS)

Breadth-first search (BFS) visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search.

Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

Implementation:

```
1. queue<int> q;
2. bool visited[N];
3. int distance[N];
4.
5. visited[x] = true;
6. distance[x] = 0;
7. q.push(x);
8. while (!q.empty()) {
9.     int s = q.front(); q.pop();
10.       // process node s
11.       for (auto u : adj[s]) {
12.           if (visited[u]) continue;
13.           visited[u] = true;
14.           distance[u] = distance[s]+1;
15.           q.push(u);
16.       }
17.   }
```

Time Complexity: O(V + E)

BELLMAN-FORD ALGORITHM

The **Bellman–Ford algorithm** finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all

other nodes is infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Implementation:

```
1. for (int i = 1; i <= n; i++) distance[i] = INF;
2. distance[x] = 0;
3.
4. for (int i = 1; i <= n-1; i++) {
5.     for (auto e : edges) {
6.         int a, b, w;
7.         tie(a, b, w) = e;
8.         distance[b] = min(distance[b], distance[a]+w);
9.     }
10.   }
```

Time Complexity: O(V x E)

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm finds shortest paths from the starting node to all nodes of the graph, like the Bellman–Ford algorithm. The benefit of Dijsktra's algorithm is that it is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Like the Bellman–Ford algorithm, Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

An efficient implementation of Dijkstra's algorithm requires that it is possible to efficiently find the minimum distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the nodes

ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

Implementation:

```
priority_queue< pair<int,int> > q;

for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});

while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

Time Complexity: O((E + V) x log(V))

FLOYD-WARSHALL ALGORITHM

The **Floyd–Warshall algorithm** provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms, it finds all shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Implementation:

```
1. for (int i = 1; i <= n; i++) {
2.     for (int j = 1; j <= n; j++) {
3.         if (i == j) distance[i][j] = 0;
4.         else if (adj[i][j]) distance[i][j] = adj[i][j];
5.         else distance[i][j] = INF;
6.     }
7. }
8.
9. for (int k = 1; k <= n; k++) {
10.       for (int i = 1; i <= n; i++) {
11.           for (int j = 1; j <= n; j++) {
12.               distance[i][j] = min(distance[i][j],
13.               distance[i][k]+distance[k][j]);
14.           }
15.       }
16.   }
```

Time Complexity: $O(V^3)$

FURTHER READING

Search for connected components in a graph:

https://cp-algorithms.com/graph/search-for-connected-components.html

0-1 BFS:

https://cp-algorithms.com/graph/01_bfs.html

Check whether a graph is Bipartite:

https://cp-algorithms.com/graph/bipartite-check.html

Minimum spanning tree - Prim's algorithm:

https://cp-algorithms.com/graph/mst_prim.html

Minimum spanning tree - Kruskal's algorithm:

https://cp-algorithms.com/graph/mst_kruskal.html

PROBLEMS

Coding Club
IIT Guwahati

WEEK 5

https://www.interviewbit.com/courses/programming/topics/graph-data-structure-algorithms/

https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/practice-problems/

https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/practice-problems/

https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/practice-problems/

https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/

https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

https://leetcode.com/tag/graph/

https://leetcode.com/tag/tree/