Graphs Tutorial

DSA

What is a graph?

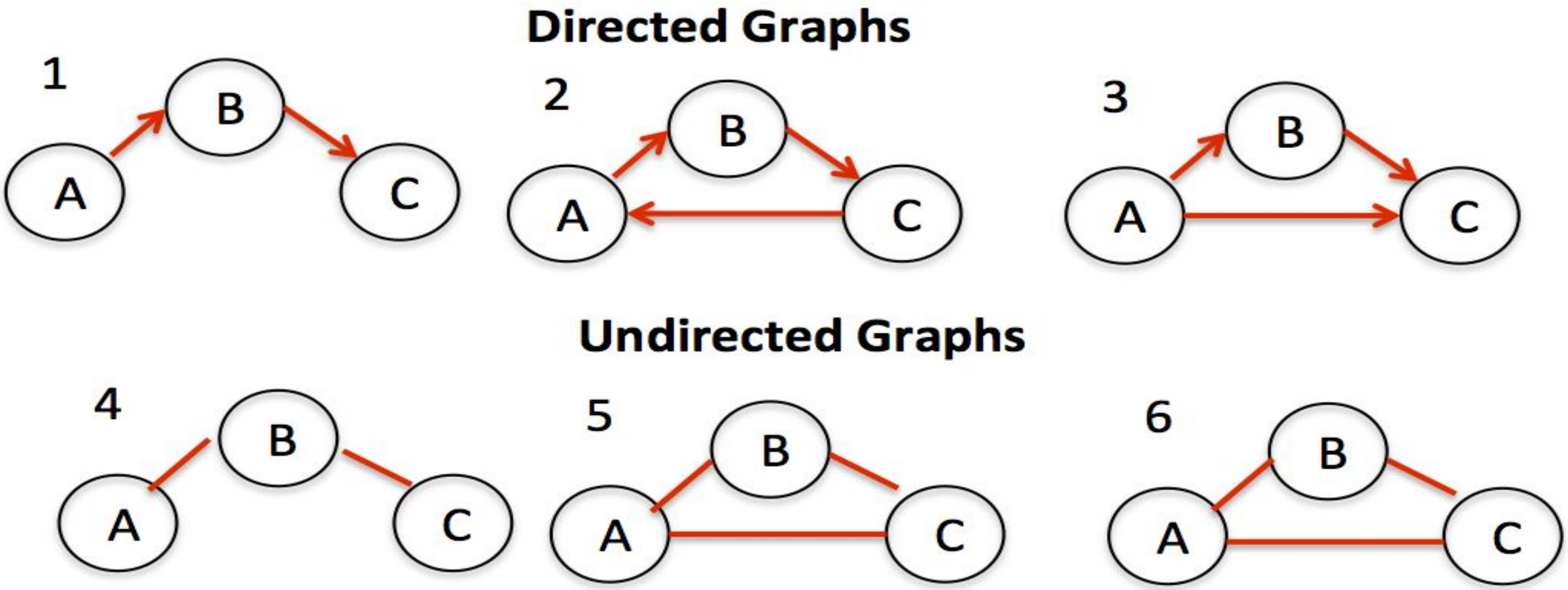
A graph G is defined as follows:

$$G=(V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)

Directed (digraph) vs. undirected(no directions) graphs



Graph terminology

1. **Adjacent Nodes -** two nodes are adjacent if they are connected by an edge
2. **Path -** a sequence of vertices that connect two nodes in a graph
3. **Cycle -** path which starts and ends on the same node
4. **Complete Graph -** Every vertex is directly connected to every other vertex
5. **Simple Graph -** Connected graph with no self loops and multiple edges
6. **Weighted graph** - a graph in which each edge carries a value
7. **Un-weighted graph** -No weight or each edge carries equal weight.

Tree vs Graph

1. Tree is Special Case of Graph.
2. Tree has always n nodes and n-1 edges.
3. So a connected graph with no cycle can be called as a tree.

No. of Edges in Complete directed graph: $n*(n-1)$

No. of Edges in Complete Undirected graph: $n*(n-1)/2$

Graph Implementation

1. **Adjacency matrix**
    a. Good for dense graphs -- |E|~O(|V|^2)
    b. Memory requirements: O(|V| + |E|) = O(|V|^2)
    c. Connectivity between two vertices can be tested quickly
2. **Adjacency list**
    a. Good for sparse graphs -- |E|~O(|V|)
    b. Memory requirements: O(|V| + |E|)=O(|V|)
    c. Vertices adjacent to another vertex can be found quickly

Graph searching

Methods:

1. Depth-First-Search (DFS) - Stack
2. Breadth-First-Search (BFS) - Queue

Depth-First-Search (DFS)

1. **Idea:** Travel as far as you can down a path
2. DFS can be implemented efficiently using a **stack**.
3. **Code:**

```
void dfs(vector<int> graph[], int u){
    visited[u] = true;
    for(int v : graph[u]){
        if(!visited[v]){
            dfs(graph, v);
        }
    }
}
```

Breadth-First-Searching (BFS)

1. **Idea:** Look at all possible paths at the same depth before you go at a deeper level
2. BFS can be implemented efficiently using a **Queue**.
3. **Code:**

```
void bfs(vector<int> graph[], int src){
    queue<int> q;
    q.push(src);
    visited[src] = true;
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int v : graph[u]){
            if(!visited[v]){
                visited[v]=true;
                q.push(v);
            }
        }
    }
}
```

Shortest Path Algorithms

1. Single Pair Shortest Path
2. Single Source Shortest Path
   a. Dijkstra’s algorithm
   b. Bellman-Ford algorithm
3. All pair Shortest Path
   a. Floyd-Warshall algorithm

Dijkstra’s Algorithm

Complexity: O(V^2) but when adjacency list is used, it is O(E*log(V)).

```
dist[src]=0;

for(int i=0; i<n-1; i++){
    int u = min_edge();
    visited[u] = true;

    for(int v=0; v<n; v++){
        if(!visited[v] && graph[u][v])
            if(dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
}
```

Bellman-Ford Algorithm

Complexity: O(N*E)

```
for (int i = 1; i <= V - 1; i++) {
    for (int j = 0; j < E; j++) {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}
```

Floyd-Warshall's Algorithm

Complexity: O(N^3)

```
for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
```

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices.

Kosaraju's algorithm

The following linear-time (i.e., $\Theta(V+E)$-time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^T.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

Minimum Spanning Tree

A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

1. Kruskal's MST Algorithm
2. Prim's MST Algorithm