Data Structures Lab 13

Course: Data Structures (CL2001) **Semester:** Fall 2022
Instructor: Muhammed Monis **T.A:** N/A

Note:

- Lab manual cover following topics
  {Simple representation of Graph ,Depth First Search, Minimum Spanning tree , Prims algorithm, Shortest path algorithm, Dijkstra's }
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

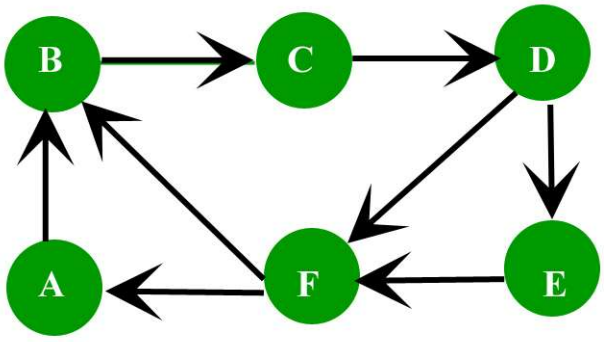
A simple Representation of graph

Introduction: A set of items connected by edges. **Each** item is called a vertex or node. **Formally,** a graph is a set of vertices and a binary relation between vertices, adjacency.

Graphs are ubiquitous in computer science. They are used to model real-world systems such as the Internet (each node represents a router and each edge represents a connection between routers); airline connections (each node is an airport and each edge is a flight); or a city road network (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

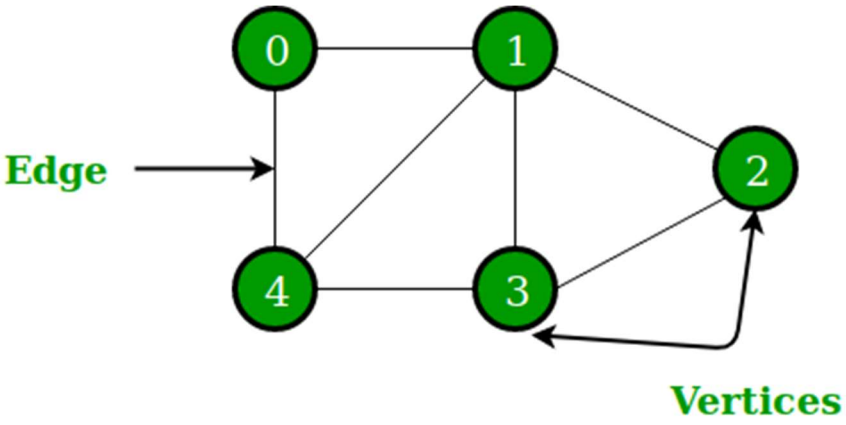
A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

A **directed graph** is a set of **vertices** (nodes) connected by **edges**, with each node having a direction associated with it.



Directed Graph

In an **undirected graph** the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected.



```
void addEdge(vector<int> adj[], int u, int v)
{
   adj[u].push_back(v);
   adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
   for (int v = 0; v < V; ++v)
   {
      cout << "\n Adjacency list of vertex "
           << v << "\n head ";
      for (auto x : adj[v])
         cout << "-> " << x;
      printf("\n");
   }
}
```

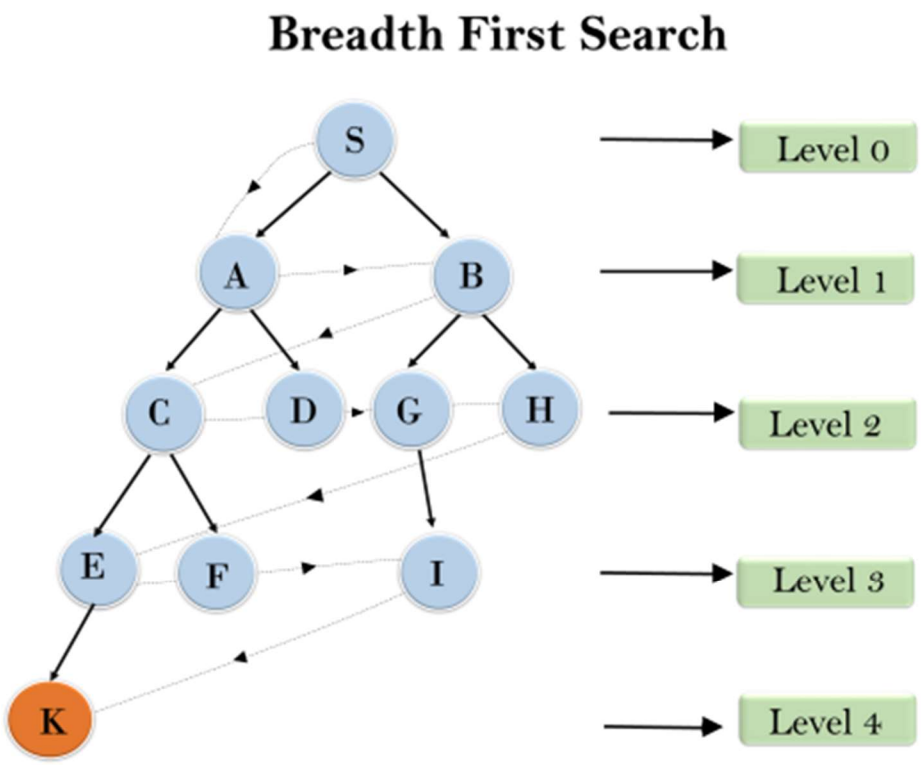
Task-1:

Create an implementation of Both Directed and Un-directed graphs and show which nodes is connected to which node. Search a Node N (Take 2 as an example) and delete that node (Take 2 as an example).

Take the starting node 0 for reference

Breadth First Search

BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.




```
S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K
```

```
Graph::Graph(int V)
{
   this->V = V;
   adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
```

```
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

Task:02
By using above program add at least 10 edges and traverse it from a given source.

Dry Run of Depth First Search

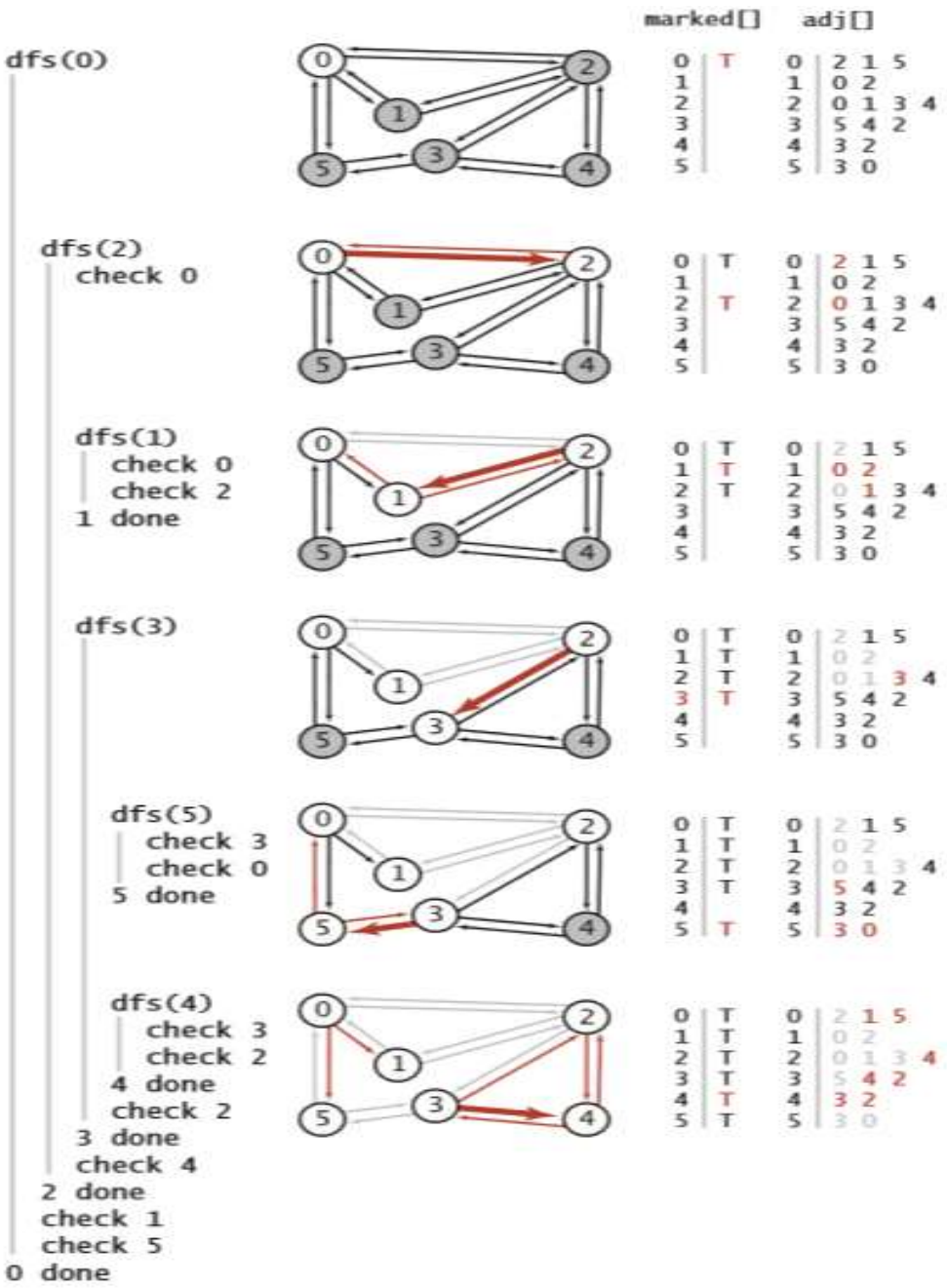


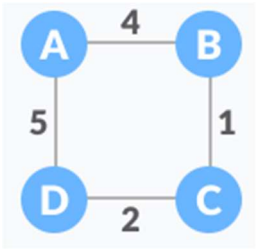
Figure-1

Task-3:

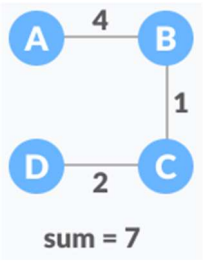
Implement a program that uses the DFS approach to traverse the graph by adding 10 nodes to the graph and showing its traversing pattern. Use the above BFS figure as an example.

Minimum Spanning Tree

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.



Weighted Graph



A possible spanning tree

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

1. form a tree that includes every vertex
2. has the minimum sum of weights among all the trees that can be formed from the graph
3. Prim's Algorithm is a famous greedy algorithm.
4. It is used for finding the Minimum Spanning Tree (MST) of a given graph.
5. To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

How Prim's algorithm works

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

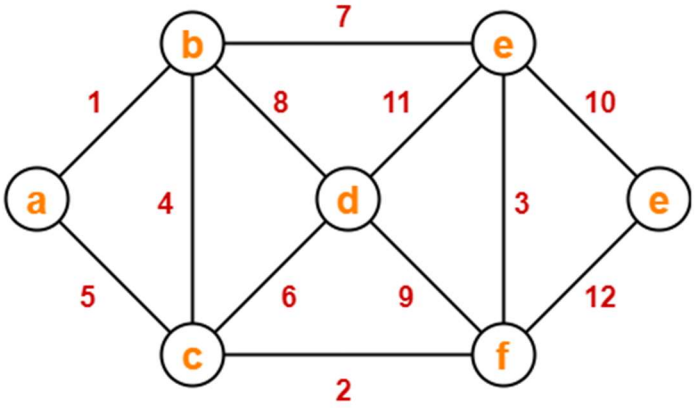
Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Example:

Please follow thee following link:
https://www.youtube.com/watch?v=ZtZaR7EcI5Y&t=290s

Task-4
Implement the below graph using Prims algorithm to find the minimum spanning tree.



Single source shortest path

Dijkstra's Algorithm:

Dijkstra Algorithm is a very famous greedy algorithm.

- It is used for solving the single source shortest path problem.
- It computes the shortest path from one source node to all other remaining nodes of the graph.

Conditions:

It is important to note the following points regarding Dijkstra Algorithm-

- Dijkstra algorithm works only for connected graphs.
- Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.
- The actual Dijkstra algorithm does not output the shortest paths.
- It only provides the value or cost of the shortest paths.
- By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.
- Dijkstra algorithm works for directed as well as undirected graphs.

How Dijkstra's algorithm works

The implementation of above Dijkstra Algorithm is explained in the following steps:

1. **S**et all vertices distances = infinity except for the source vertex, set the source distance = 0.
2. Push the source vertex in a min-priority queue in the form (distance , vertex),as the comparison in the min-priority queue will be according to vertices distances.
3. Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
4. Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.

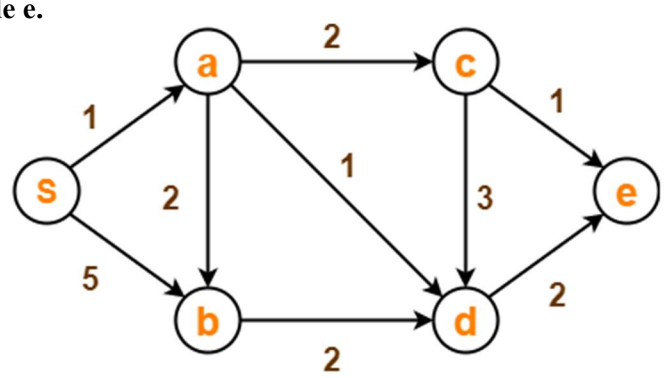
5. If the popped vertex is visited before, just continue without using it.
6. Apply the same algorithm again until the priority queue is empty.

Example:

Please follow thee following link:
https://www.youtube.com/watch?v=smHnz2RHJBY&t=1816s

Task-5

1. **Implement the below graph using the shortest path algorithm start with the node S and traverse to the node e.**



Summary Discussion

Graph Data structure

A graph is a data structure that consists of the following two components:

- ✓ A finite set of vertices also called as nodes.
- ✓ A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Representation of Graph

The following two are the most commonly used representations of a graph.

- ✓ Adjacency Matrix
- ✓ Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Depth-first search

We often learn properties of a graph by systematically examining each of its vertices and each of its edges. Determining some simple graph properties—for example, computing the degrees of all the vertices—is easy if we just examine each edge (in any order whatever). But many other graph properties are related to paths, so a natural way to learn them is to move from vertex to vertex along the graph's edges. Nearly all of the graph-processing algorithms that we consider use this same basic abstract model, albeit with various different strategies.

The simplest is a classic method that we now consider. Searching in a maze. It is instructive to think about the process of searching through a graph in terms of an equivalent problem that has a long and distinguished history—finding our way through a maze that consists of passages connected by intersections. Some mazes can be handled with a simple rule, but most mazes require a more sophisticated strategy. Using the terminology maze instead of graph, passage instead of edge, and intersection instead of vertex is making mere semantic distinctions, but, for the moment, doing so will help to give us an intuitive feel for the problem.