Lab 13 Implementation of Graphs in C language

Objectives:

- To understand the concept of weighted and unweighted graph data structures.
- Learn to ***construct a graph*** using C language.
- Learn to perform breadth-first and depth-first graph ***traversals***.
- Learn to implement ***insertion*** and ***deletion*** of vertices and edges in a graph.

Pre Lab:

A graph is a ***pictorial representation*** of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as ***vertices***, and the links that connect the vertices are called ***edges***.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

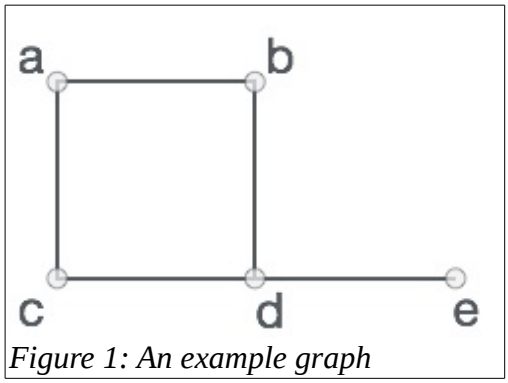

Figure 1: An example graph

In the above graph, V = {a, b, c, d, e} and E = {ab, ac, bd, cd, de}

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network etc. Consider a network of flights for PIA shown below. Here the cities, Islamabad, Karachi, Rome etc form the ***vertices*** of the graph, while the paths linking these vertices are the ***edges*** of the graph.

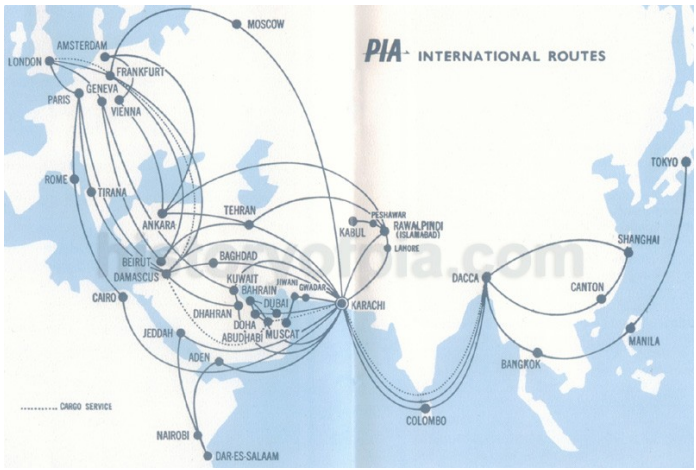

Figure 2: PIA Flight Network

Types of graphs:

While nodes and edges may have any number of interesting properties and labels, some properties are more common than others. In particular there are two properties of edges that stand out so much that they are said to change the type of graph. These two properties are edge weight, and edge directionality.

Directed vs Undirected Graphs:

If the edges in your graph have directionality then your graph is said to be a directed graph (sometimes shortened to digraph). In a directed graph all of the edges represent a one way relationship, they are a relationship from one node to another node — but not backwards. In an undirected graph all edges are bidirectional. It is still possible (even common) to have bidirectional relationships in a directed graph, but that relationship involves two edges instead of one, an edge from A to B and another edge from B to A.

Directed edges have a subtle impact on the use of the term neighbors. If an edge goes from A to B, then B is said to be A's neighbor; but the reverse is not true. A is not a neighbor of B unless there is an edge from B to A. In other words, a node's neighbors are the set of nodes that can be reached from that node.

Let's use two social networks as examples. On Facebook the graph of friends is undirected. If you are someone's friend on Facebook they are your friend too — friendship on Facebook is always bidirectional meaning the graph representation is undirected. On Twitter, however, "following" someone is a one way relationship. If you follow Shahid Afridi, that doesn't mean he follows you. The graph of Twitter users and their followers is a directed graph.

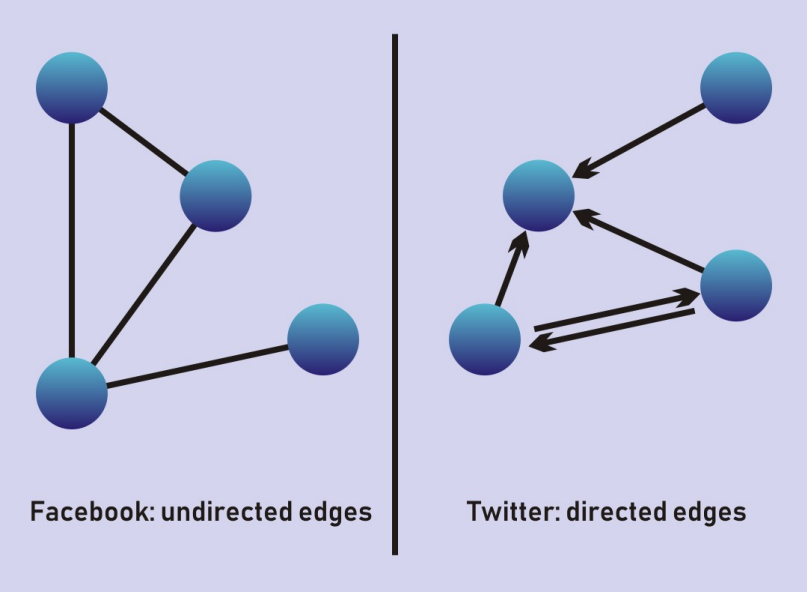

Figure 3: Example of Directed and Undirected Graphs

Weighted vs Unweighted Graphs

If edges in your graph have weights then your graph is said to be a weighted graph, if the edges do not have weights, the graph is said to be unweighted. A weight is a numerical value attached to each individual edge. In a weighted graph relationships between nodes have a magnitude and this

magnitude is important to the relationship we're studying. In an unweighted graph the existence of a relationship is the subject of our interest.

As an example of a weighted graph, imagine you run an airline and you'd like a model to help you estimate fuel costs based on the routes you fly. In this example the nodes would be airports, edges would represent flights between airports, and the edge weight would be the estimated cost of flying between those airports. Such a model could be used to determine the cheapest path between two cities, or run simulations of different potential flight offerings.

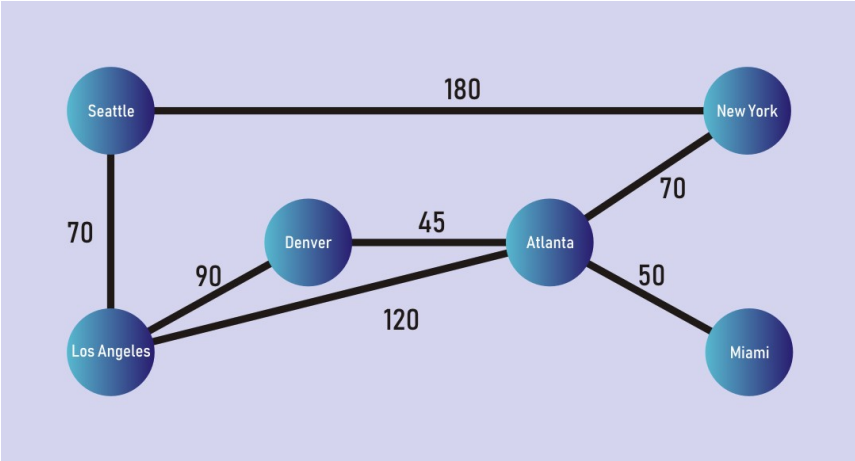

Figure 4: A Weighted Graph

Representation of Graphs

There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation. Here, we'll see three ways to represent graphs. Lets consider the following graph:

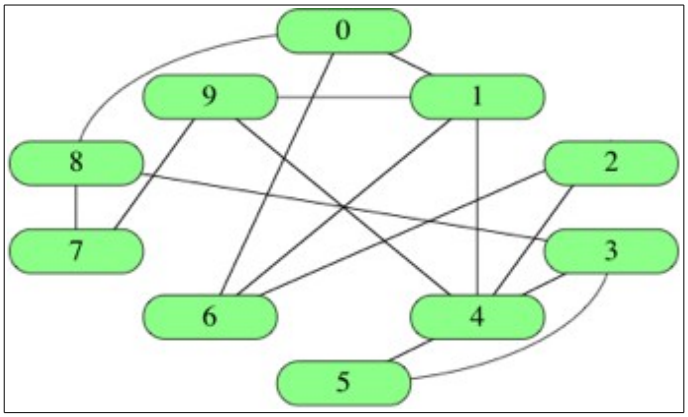

Figure 5: A Social Media Graph

Adjacency List:

Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex iii, store an array of the vertices adjacent to it. We typically have an array of |V|⎪V⎪vertical bar, V, vertical bar adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:

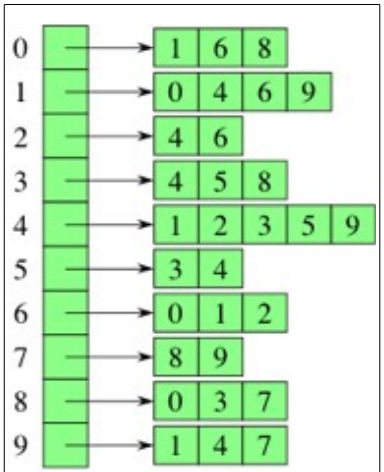

Figure 6: Adjacency List for Graph of Figure 5

Adjacency Matrix:

For a graph with |V|∣V∣vertical bar, V, vertical bar vertices, an adjacency matrix is a |V| \times |V| ∣V∣×∣V∣vertical bar, V, vertical bar, times, vertical bar, V, vertical bar matrix of 0s and 1s, where the entry in row iii and column jjj is 1 if and only if the edge (i,j)(i,j)left parenthesis, i, comma, j, right parenthesis is in the graph. If you want to indicate an edge weight, put it in the row iii, column jjj entry, and reserve a special value (perhaps null) to indicate an absent edge. Here's the adjacency matrix for the social network graph:

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Figure 7: Adjacency Matrix Representation of Graph shown in Figure 5

The adjacency matrix for a weighted graph will have the weights in place of ones (1s). A programming implementation may initialize the adjacency matrix with a negative number to denote and infinite weight (not adjacent).

Depth First Search (DFS):

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

1. Pick a starting node and push all its adjacent nodes into a stack.
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
3. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

In-Lab Tasks:

Task 1:

Complete the Adjacency Matrix given as '`my_graph[][]`' in the main function of the skeleton code provided. You will use the following figure for completing this matrix. Call the function '`add_edge()`' with correct weights to fill in the matrix.

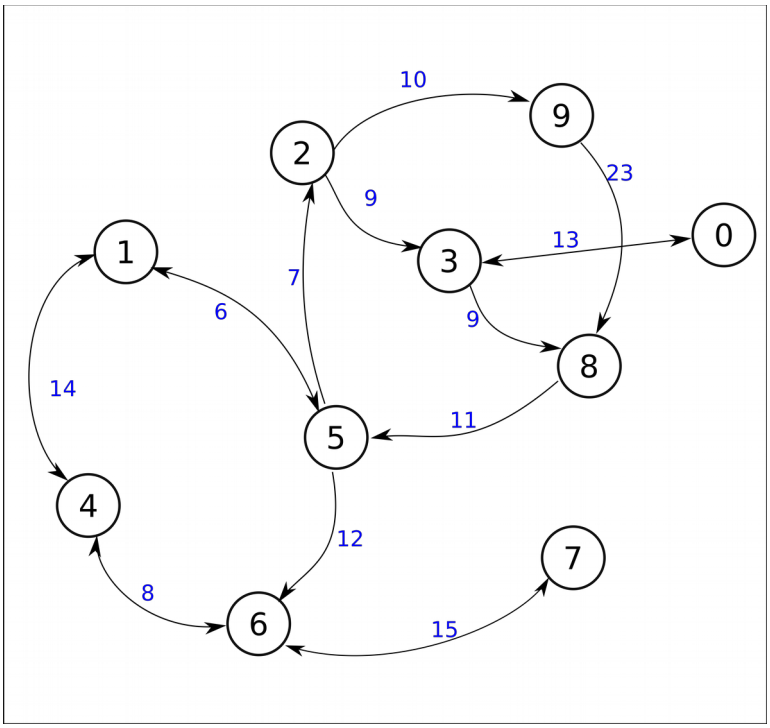

Figure 8: A Weighted Directed Graph with 10 Vertices

Task 2:

For this part you will have to complete the function '`find_path_dfs()`' which finds the cost of going from the source vertex (***src***) to destination vertex (***dst***) using Depth First Search (**DFS**) algorithm. You will use a stack for implementing DFS algorithm.

Post Lab Task:

Complete the code for Task 2 and submit along with a report explaining your implementation.