**PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE - 411043 Department of Computer Engineering** 

**S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043**

**Data Structures and Algorithms Laboratory**

**Batch-IV (H4)**

**Name: Rushikesh Vidye**

**Roll No.: 21471**

**Class: SE4**

**Assignment No. 6**

**Title:** Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

**Software Requirement:**

a) OS : Microsoft Windows 10.

b) Browser: Google Chrome.

c) VS Code.

**Hardware Requirement:**

a) Processor: Intel Core i5-8265U.

b) Ram: 8 GB DDR4 2800Mhz.

**Theory:**

**Graph:**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.

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. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

For example:

Lightbox

The following two are the most commonly used representations of a graph.   
**1.** Adjacency Matrix   
**2.** 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.

The adjacency matrix for the above example graph is

Adjacency Matrix Representation

**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 *i*th 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.

Adjacency List Representation of Graph

**Breadth First Search or BFS for a graph:**

Breadth First Traversal for a graph is like Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



**Depth First Search for Graph:**

Depth First Traversal for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a Boolean visited array.

**Algorithm :**

Create Graph: int V, Int E :

1. Accept number of Vertex and Edges.
2. Initialize V\*V AdjMatrix with 0.
3. Initialize AdjList with V size.

Add Edge: Int Start, Int E:

1. Accept Start and E (Considering Bidirectional Edge)
2. Set element at AdjMatix[start][e] = 1
3. Set element at AdjMatrix[e][start] = 1
4. Push AdjList[start] with E

DFS: Int Start, Vector visitedNode:

1. Print start
2. Visited[start] = true
3. Find node adjacent to start that is not visited:
   1. Call DFS ( adjecentNode, visitedList)

BFS: Int Start:

1. Initialize visitedList the size of V.
2. Create a queue and Push start
3. Set visited[start] = true
4. Until queue is not empty
   1. Start = quere\_front()
   2. Print start
   3. Pop\_front()
   4. Add elements from AdjList to queue if they are not in visitedList

**Time Complexity:**

|  |  |  |
| --- | --- | --- |
| Sr.No | Methods | Complexity |
| 1 | Create() | O(n^2) |
| 2 | AddEdge() | O(1) |
| 3 | DFS() | O(V^2) |
| 4 | BFS() | O(V+E) |

**Test Cases:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr.no | Input | Output | Result |
| 1 | Create Graph with structure  0 --- 1 --- \  | X 2  4 ---- 3 ---/ | Graph created | pass |
| 2 | DFS | 0 1 2 3 4 | pass |
| 3 | BFS | 0 1 4 3 2 | pass |

**Conclusion:**

Hence we learnt the implementation of BFS and DFS traversing techniques with Graph using adjacent matrix and adjacent list.

**Code:**

|  |
| --- |
| #pragma once  #include<iostream>  #include<vector>  #include<list>  using namespace std;  class Graph{      int v,e;      int \*\* adj;      list<int> \*adjList;      public:      Graph(int v);      Graph(int v,int e);      void addEdge(int start,int e);      void DFS(int start,std::vector<bool>& visited);      void BFS(int s);};  Graph::Graph(int v,int e){      this->v = v;      this->e = e;      adj = new int\*[v];      for(int row = 0 ; row < v; row++){          adj[row] = new int[v];          for(int column = 0;column < v; column++){              adj[row][column] = 0;    }}      adjList = new list<int>[v];}  void Graph::addEdge(int start,int e){      adj[start][e] = 1;      adj[e][start] = 1;      adjList[start].push\_back(e);}  void Graph::DFS(int start,std::vector<bool>& visited){      printf("%d\t",start);      visited[start] = true;      for (int i = 0; i < v; i++) {          if (adj[start][i] == 1 && (!visited[i])) {              DFS(i, visited);}}}  void Graph::BFS(int s)  {      bool \*visited = new bool[v];      for(int i = 0; i< v;i++)          visited[i] = false;      list<int>queue;      visited[s] = true;      queue.push\_back(s);      list<int>::iterator i;      while(!queue.empty()){          s = queue.front();          cout<< s << " ";          queue.pop\_front();          for(i = adjList[s].begin();i!=adjList[s].end();++i)          {   if(!visited[\*i]){                  visited[\*i] = true;                  queue.push\_back(\*i);           }}}}  #include"header.h"  int main(void)  {      int v = 5, e = 7;      Graph G(v, e);      G.addEdge(0, 1);      G.addEdge(0, 4);      G.addEdge(0,3);      G.addEdge(1, 2);      G.addEdge(1, 4);      G.addEdge(2, 3);      G.addEdge(3, 4);      std::vector<bool> visited(v, false);      G.DFS(0, visited);      cout<<endl<<" BFS::"<<endl;      G.BFS(0);  } |

**Output:**

|  |
| --- |
| **DFS ::**  0 1 2 3 4  BFS::  0 1 4 3 2 |