**ADVANCED DATA STRUCTURE**

**GROUP B**

**ASSIGNMENT 4**

**YEAR: 2017-18**

**COLLEGE: VIIT**

**Date Of Completion : 30/03/2018**

**Title:**

***Graphs:*** Storage and Traversal

**Problem Statement:**

There are flight paths between cities. If there is a flight between city A and city B then

there is an edge between the cities. The cost of the edge can be the time that flight takes

to reach city B from A, or the amount of fuel used for the journey. Represent this as a

graph. The node can be represented by airport name or name of the city. Use adjacency list

representation of the graph or use adjacency matrix representation of the graph. Justify the

storage representation used.

**Objective:**

1. To understand and implement different storage representation for graphs.

2. To traverse graph using DFS (Depth First Search) or BFS (Depth First Search).

**Software And Hardware Requirement:**

1. 64-bit Open source Linux or its derivative.

2. Open Source C++ Programming tool like G++/GCC.

**Theory:**

Graph is a data structure that consists of 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 same as (v, u) in case of directed graph(di-graph).

The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges

may contain weight/value/cost.

Graphs are used to represent many real life applications: 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 and locale. See this for more applications of graph.

Following is an example undirected graph with 5 vertices.

Following two are the most commonly used representations of graph.

1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice

of the 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:

2Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time.

Queries like whether there is an edge from vertex u to vertex v are efficient and can be done

O(1).

Cons: Consumes more space O(V 2 ). Even if the graph is sparse(contains less number of

edges), it consumes the same space. Adding a vertex is O(V 2 ) time

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let

the array be array[]. An entry array[i] represents the linked 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 stored in nodes of linked lists. Following is adjacency list representation of

the above graph

Tree Traversal

A Tree is typically traversed in two ways:

1. Breadth First Traversal (Or Level Order Traversal)

2. Depth First Traversals

(a) Inorder Traversal (Left-Root-Right)

(b) Preorder Traversal (Root-Left-Right)

(c) Postorder Traversal (Left-Right-Root)

How to Pick One?

1. Extra Space can be one factor

2. Depth First Traversals are typically recursive and recursive code requires function call

overheads.

3. The most important points is, BFS starts visiting nodes from root while DFS starts

visiting nodes from leaves. So if our problem is to search something that is more likely

to closer to root, we would prefer BFS. And if the target node is close to a leaf, we

would prefer DFS.

36

**Algorithm:**

BFS (Breadth First Search)

Step 1:Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert

Step 3:Visit all the adjacent vertices of the verex which is at front of the Queue whic

Step 4:When there is no new vertex to be visit from the vertex at front of the Queue th

Step 5:Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused ed

**Code :**

//==============Using Adjancy Matrix==============//

#include<iostream>

#include<string>

#include<queue>

using namespace std;

class Graph

{

private:

int n;

string city[10];

string city1,city2;

int mat[10][10];

queue <string> q;

public:

void input();

void Adc\_edge();

void Create\_AdcMat(string city1,string city2);

void Traversal();

int Index(string city\_name);

};

void Graph::input()

{

int i,j;

cout<<"\nEnter No. of Cities : ";

cin>>n;

for(i=0;i<n;i++)

{

cout<<"\nEnter Name of the "<<i+1<<" Cities : ";

cin>>city[i];

4}

}

void Graph::Adc\_edge()

{

int i,j;

char ch;

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

mat[i][j]=0;

}

}

do

{

cout<<"\nEnter Name of Source City : ";

cin>>city1;

cout<<"\nEnter Name of Destination City : ";

cin>>city2;

Create\_AdcMat(city1,city2);

cout<<"\nDo u want to Add More :";

cin>>ch;

}while(ch==’y’|| ch==’Y’);

cout<<"\n\t===== Adanjancy Matrix===== \n";

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

cout<<mat[i][j]<<" ";

}

cout<<"\n";

}

}

void Graph::Create\_AdcMat(string city1,string city2)

{

int i,j;

for(i=0;i<n;i++)

{

if(city[i]==city1)

{

for(j=0;j<n;j++)

{

if(city[j]==city2)

{

mat[i][j]=1;

5mat[j][i]=1;

break;

}

}

break;

}

}

}

int Graph::Index(string city\_name)

{

int k=0;

while(city[k]!=city\_name)

k++;

return k;

}

void Graph::Traversal()

{

string start\_city,top;

int visited[10],i,j,k=0;

for(i=0;i<n;i++)

visited[i]=0;

cout<<"\nEnter Starting City : ";

cin>>start\_city;

q.push(start\_city);

top=q.front();

//q.pop();

cout<<"\n\tGraph Traversal : ";

//cout<<start\_city<<" ";

k=Index(start\_city);

visited[k]=1;

while(!q.empty())

{

for(i=0;i<n;i++)

{

if(mat[k][i]==1)

{

if(visited[i]==0)

{

visited[i]=1;

q.push(city[i]);

}

}

}

top=q.front();

6cout<<top<<" ";

q.pop();

k=Index(top);

}

}

int main()

{

Graph g;

g.input();

g.Adc\_edge();

do

{

g.Traversal();

}while(1);

return 0;

}

/\*

\* Graph2.cpp

\*

\* Created on: Feb 24, 2017

\*

Author: student

\*/

/\*

\* Graph2.cpp

\*

\* Created on: Feb 23, 2017

\*

Author: student

\*/

//========Directed

Graph using Adjancancy List============//

#include<iostream>

#include<string>

#include<queue>

using namespace std;

typedef struct node

{

string city\_name;

int cost;

struct node \*next,\*down;

}node;

class Graph

{

private:

int n;

string city[10];

string city1,city2;

int mat[10][10];

queue <string> q;

node \*temp,\*New,\*head[10];

public:

node \*Create\_node(string city\_name);

void input();

void Adc\_edge();

void Create\_AdcMat(string city1,string city2);

void Create\_Adc\_List();

void Traversal();

};

node \*Graph::Create\_node(string city\_nm)

{

New=new node;

New->city\_name=city\_nm;

New->next=New->down=NULL;

return New;

}

void Graph::input()

{

int i,j;

cout<<"\nEnter No. of Cities : ";

cin>>n;

for(i=0;i<n;i++)

{

cout<<"\nEnter Name of the "<<i+1<<" Cities : ";

cin>>city[i];

}

}

void Graph::Adc\_edge()

{

int i,j;

char ch;

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

mat[i][j]=0;

}

}

do

{

8cout<<"\n Enter Name of Source City : ";

cin>>city1;

cout<<"\n Enter Name of Destination City : ";

cin>>city2;

Create\_AdcMat(city1,city2);

cout<<"\n\tDo u want to Add More :";

cin>>ch;

}while(ch==’y’|| ch==’Y’);

/\*cout<<"\n\t===== Adanjancy Matrix===== \n";

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

cout<<mat[i][j]<<" ";

}

cout<<"\n";

}

cout<<endl;\*/

}

void Graph::Create\_AdcMat(string city1,string city2)

{

int i,j;

for(i=0;i<n;i++)

{

if(city[i]==city1)

{

for(j=0;j<n;j++)

{

if(city[j]==city2)

{

mat[i][j]=1;

break;

}

}

break;

}

}

}

void Graph::Create\_Adc\_List()

{

int i,j,flag=0;

for(i=0;i<n;i++)

{

head[i]=new node;

head[i]->city\_name=city[i];

9}

for(i=0;i<n;i++)

{

flag=0;

for(j=0;j<n;j++)

{

if(mat[i][j]==1)

{

if(flag==0)

{

New=Create\_node(city[j]);

head[i]->next=New;

temp=New;

flag=1;

}

else

{

New=Create\_node(city[j]);

temp->next=New;

temp=New;

}

}

}

cout<<endl;

}

}

void Graph::Traversal()

{

int i;

string scity;

node \*temp1;

cout<<"\n\tGraph Traversal Using Adjancancy List :";

do

{

cout<<"\n\n Enter Start City Name : ";

cin>>scity;

for(i=0;i<n;i++)

{

if(city[i]==scity)

{

temp1=head[i];

cout<<"\t\t"<<head[i]->city\_name<<" => ";

while(temp1!=NULL)

{

if(head[i]->next!=NULL)

cout<<temp1->city\_name<<" -> ";

10temp1=temp1->next;

}

break;

}

cout<<endl;

}

}while(1);

}

int main()

{

Graph g;

g.input();

g.Adc\_edge();

g.Create\_Adc\_List();

g.Traversal();

return 0;

}

**Output :**

//=============Using Adancany matrix=========//

Enter No. of Cities : 4

Enter Name of the 1 Cities : a

Enter Name of the 2 Cities : b

Enter Name of the 3 Cities : c

Enter Name of the 4 Cities : d

Enter Name of Source City : a

Enter Name of Destination City : b

Do u want to Add More :y

Enter Name of Source City : b

Enter Name of Destination City : c

Do u want to Add More :y

Enter Name of Source City : a

11Enter Name of Destination City : c

Do u want to Add More :y

Enter Name of Source City : a

Enter Name of Destination City : d

Do u want to Add More :y

Enter Name of Source City : d

Enter Name of Destination City : b

Do u want to Add More :n

=====

0 1 1

1 0 1

1 1 0

1 1 0

Adanjancy Matrix=====

1

1

0

0

Enter Starting City : a

Graph Traversal : a b c d

Enter Starting City : b

Graph Traversal : b a c d

Enter Starting City : c

Graph Traversal : c a b d

Enter Starting City : d

Graph Traversal : d a b c

Enter Starting City :

//=====================================

Enter No. of Cities : 4

Enter Name of the 1 Cities : a

Enter Name of the 2 Cities : b

Enter Name of the 3 Cities : c

12Enter Name of the 4 Cities : d

Enter Name of Source City : a

Enter Name of Destination City : c

Do u want to Add More :y

Enter Name of Source City : a

Enter Name of Destination City : b

Do u want to Add More :y

Enter Name of Source City : d

Enter Name of Destination City : a

Do u want to Add More :y

Enter Name of Source City : b

Enter Name of Destination City : c

Do u want to Add More :y

Enter Name of Source City : b

Enter Name of Destination City : d

Do u want to Add More :n

Graph Traversal Using Adjancancy List :

Enter Start City Name : d

d => d -> a ->

Enter Start City Name : a

a => a -> b -> c ->

13Enter Start City Name : c

c => c ->

**Conclusion:**

Through this assignment we understood different storage representations of graphs and meth-

ods (DFS and BFS) to traverse the prepared graph and implemented linked list representa-

tion of graph and traversed it using DFS.