**DATA STRUCTURES**

**Lab 15**

**Name: Ahmed Kasteer**

**Roll Number: 20F-0336**

**Section: 3D**

**Q1:**

#include<iostream>

#include<fstream>

#include<queue>

#include<stack>

using namespace std;

class AdjacencyList;

class SingleNode

{

char vertex;

double edge\_weight;

SingleNode\* next;

bool visited;

friend class AdjacencyList;

SingleNode()

{

visited = false;

next = NULL;

vertex = -1;

edge\_weight = 0;

}

};

class AdjacencyList

{

SingleNode\* arr;

int n;

public:

AdjacencyList();

AdjacencyList(int);

void Implementation();

void BFS();

void DFS();

void Display();

};

int main()

{

int n = 0;

cout << "Enter The Number of Vertices: ";

cin >> n;

AdjacencyList A(n);

while (true)

{

int c = 0;

cout << "Press 1 to give Data(0 to exit): ";

cin >> c;

if (c == 0) break;

else if (c == 1) A.Implementation();

else break;

}

cout << "\n\nBFS " << endl;

A.BFS();

cout << "\n\nDFS " << endl;

A.DFS();

}

AdjacencyList::AdjacencyList()

{

arr = NULL;

n = 0;

}

AdjacencyList::AdjacencyList(int n)

{

arr = new SingleNode[n];

this->n = n;

char ch = 'A';

for (int i = 0; i < n; ++i, ch++)

{

arr[i].vertex = ch;

}

}

void AdjacencyList::Implementation()

{

ifstream read;

read.open("bfs.txt");

char mainvertex, toconnect;

while (!read.eof())

{

read >> mainvertex;

read >> toconnect;

if (!read.eof())

{

SingleNode\* newNode = new SingleNode;

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

{

if (arr[i].vertex == mainvertex)

{

newNode->vertex = toconnect;

SingleNode\* curr = arr[i].next;

while (true)

{

if (curr == NULL)

{

arr[i].next = newNode;

break;

}

else if (curr->next == NULL)

{

curr->next = newNode;

break;

}

else

curr = curr->next;

}

break;

}

}

}

}

read.close();

}

void AdjacencyList::BFS()

{

queue<char> Q;

Q.push(arr[0].vertex);

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

{

SingleNode\* curr = arr[i].next;

while (curr)

{

Q.push(curr->vertex);

curr = curr->next;

}

}

for (int i = 0; i < Q.size(); ++i)

{

cout << Q.front() << " ";

Q.pop();

}

cout << endl;

}

void AdjacencyList::DFS()

{

stack<char> Q;

Q.push(arr[0].vertex);

for (int i = 0, count = 0; i < n + 1; ++i)

{

char c = Q.top();

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

{

if (arr[j].vertex == c)

{

i = j;

break;

}

}

cout << Q.top() << " ";

if (Q.top() == 'G')

break;

Q.pop();

SingleNode\* curr = arr[i].next;

while (curr)

{

Q.push(curr->vertex);

curr = curr->next;

}

}

cout << endl;

}

**Q2:**

Dijkstra's algorithm also known as a greedy algorithm which we can use to find shortest distances or minimum costs depending on what is represented in a graph. We're basically working backwards from the end to the beginning, finding the shortest leg each time. The steps to this algorithm are as follows:

Step 1: Start at the ending vertex by marking it with 0, because it's 0 units from the end. Call this vertex your current vertex and put a circle around it indicating as such.

Step 2: Identify all of the vertices that are connected to the current vertex with an edge. Calculate their distance to the end by adding the weight of the edge to the mark on the current vertex. Mark each of the vertices with their corresponding distance, but only change a vertex's mark if it's less than a previous mark. Each time you mark the starting vertex with a mark, keep track of the path that resulted in that mark.

Step 3: Label the current vertex as visited by putting an X over it. Once a vertex is visited, we won't look at it again.

Step 4: Of the vertices you just marked, find the one with the smallest mark, and make it your current vertex. Now, you can start again from step 2.

Step 5: Once you've labeled the beginning vertex as visited - stop. The distance of the shortest path is the mark of the starting vertex, and the shortest path is the path that resulted in that mark.

**Q3:**

The Bellman Ford algorithm is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) that computes [shortest paths](https://en.wikipedia.org/wiki/Shortest_path) from a single source [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) to all of the other vertices in a [weighted digraph](https://en.wikipedia.org/wiki/Weighted_digraph). It is slower than [Dijkstra's algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a negative cycle (i.e. a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)) whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path any path that has a point on the negative cycle can be made cheaper by one more [walk](https://en.wikipedia.org/wiki/Walk_(graph_theory)) around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.