American International University-Bangladesh (AIUB)

FACULTY OF SCIENCE & INFORMATION TECHNOLOGY

Department of Computer Science



**LAB MANUAL 06**

**Fall 2015-2016**

**CSC2211 Algorithms** |

**TITLE**| Graph Representations and Traversals

**PREREQUISITE |** Knowledge of Arrays| Linked lists | STL

**OBJECTIVE |** Learn Graph Representation, Implementation & Graph Search Strategies

**THEORY | BASICS OF GRAPH**

**Graph:** A graph is a set of nodes (vertices) and edges. The node that holds data is called vertex and the line connecting two vertices is called edge. If G denotes a graph, G= (V, E) then V denotes set of vertices and E denotes set of edges.

**Weighted Graph**: If the value (cost) of each edge is given, the graph is called weighted graph.

**Path:** A path is a sequence of vertices where each pair of successive vertices is connected by an edge.

**Connected Graph:** A graph is called connected if there is a path between each pair of vertices

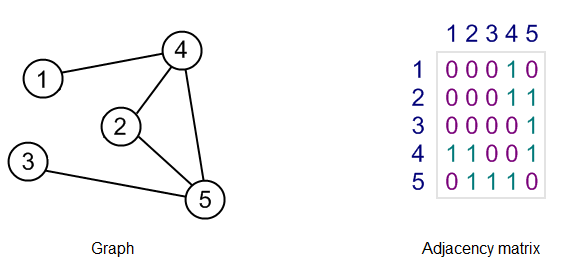
**Cycle:** A cycle is a path where first and last vertices are the same.

**GRAPH REPRESENTATION**

There are several possible ways to represent a graph inside the computer. We will discuss two of them: **Adjacency matrix** and **Adjacency list**.

## **Adjacency matrix**

Each cell aij of an adjacency matrix contains **1**, if there is an edge between i-th and j-th vertices, and **0** otherwise. Before discussing the advantages and disadvantages of this kind of representation, let us see an example.

**Creating Matrix**

Matrix in this example will be a dynamic two-dimensional array. We will use a pointer-to-pointer-to-integer type structure.

**Matrix Types**

row

2

col

2

p

p[0]

p [1]

**int \*p[row]**

**int \*\*p**

**Implementing Matrix**

**// dimensions, row × column**

**int row, col;**

**// initialized to all 0 entries**

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

{ for (int j = 0; j < col; j++)

{

Graph[i][j] = 0;

}

}

**// pointer to a pointer to a integer**

**int \*\*Graph;**

Allocate the array **Graph** and the **Graph[i]** arrays:

Graph = new int\*[row];

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

{

Graph[i] = new int[col];

}

## **Code snippets**

For reasons of simplicity, code snippets for adjacency matrix have been given below.

Notice, that it is an implementation for undirected graphs.

#include <iostream>

using namespace std;

class Graph {

private:

bool \*\* adjacencyMatrix;

int vertexCount;

public:

Graph(int vertexCount);

~Graph();

void addEdge(int i, int j);

void removeEdge(int i, int j);

bool isEdge(int i, int j);

void display();

};

Graph::Graph(int vertexCount) {

this->vertexCount = vertexCount;

adjacencyMatrix = new bool\*[vertexCount];

for (int i = 0; i < vertexCount; i++) {

adjacencyMatrix[i] = new bool[vertexCount];

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

adjacencyMatrix[i][j] = false;

}

}

Graph::~Graph() {

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

delete[] adjacencyMatrix[i];

delete[] adjacencyMatrix;

}

void Graph::removeEdge(int i, int j) {

if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount) {

adjacencyMatrix[i][j] = false;

adjacencyMatrix[j][i] = false;

}

}

void Graph::addEdge(int i, int j) {

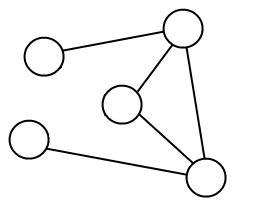
if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount) {

adjacencyMatrix[i][j] = true;

adjacencyMatrix[j][i] = true;

}

}

***Graph***

bool Graph::isEdge(int i, int j) {

if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount)

return adjacencyMatrix[i][j];

else

return false;

}

**D**

**A**

void Graph::display(){

int u,v; //vertex

for(u=0; u<vertexCount; ++u) {

cout << "\nadj[" << (char) (u+65) << "] -> ";

for(v=0; v<vertexCount; ++v) {

cout << " " << adjacencyMatrix[u][v];

}

}

cout << "\n\n";

}

**B**

**C**

**E**

***Task 1***

Consider the above graph and implemented methods to write a driving function ***main()*** so the Graph isrepresented in adjacency matrix. Also print the Matrix on output screen.

***Task 2***

Representation in a text file as a matrix and save it with name ***graph.txt*** in the same folder with your .cpp file. Now use the following code to load your graph into a multi dimensional array. Also print the Array.

Graph::Graph(char filename[],int vertexCount){

this->vertexCount = vertexCount;

adjacencyMatrix = new bool\*[vertexCount];

ifstream file;

file.open(filename, ios::in);

if( !file) {

cout << "\nError: Cannot open file\n";

return;

}

if(file.is\_open())

{

for (int i = 0; i < vertexCount; i++) {

adjacencyMatrix[i] = new bool[vertexCount];

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

file>>adjacencyMatrix[i][j];

}

}

}

***Task:*** Add additional member function called ***findAdjacencyVertices(int vertex),*** which displays adjacency nodes/vertices of that vertex.

***Task:*** Add additional member function called ***findDegree(int vertex),*** which displays the number of vertices that are connected of that vertex.

**GRAPH SEARCH METHODS**

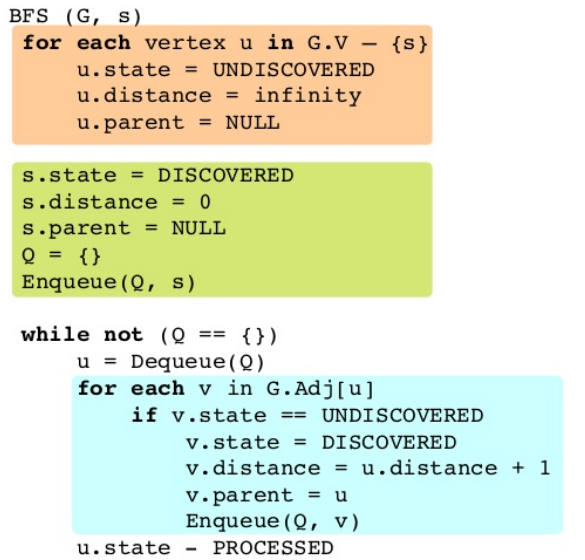
There are two principal methods for graph traversal, which are as follows:

1. Breadth First Search(BFS)
2. Depth First Search(DFS)

**BREADTH FIRST SEARCH (BFS)**

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim’s minimum-spanning tree algorithm and Dijkstra’s single-source shortest-paths algorithm use ideas similar to those in breadth-first search. Given a graph G= (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s. It computes the distance (smallest number of edges) from **s** to each reachable vertex. It also produces a “breadth-first tree” with root s that contains all reachable vertices. For any vertex reachable from s, the simple path in the breadth-first tree from s to v corresponds to a “shortest path” from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

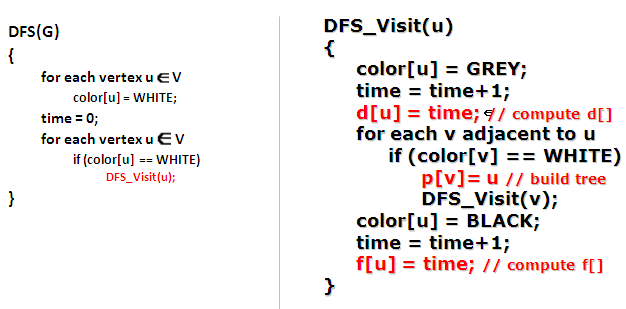
The pseudo-code for Breadth-first search (BFS) is given below:



# Depth-first search (DFS) for undirected graphs

**Depth-first search**, or **DFS**, is a way to traverse the graph. Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.

The pseudocode for Depth-first search (DFS)is give



n below:

## ***Algorithm***

In DFS, each vertex has three possible colors representing its state:

http://www.algolist.net/img/graphs/DFS/DFS-states-white.pngwhite: vertex is unvisited;http://www.algolist.net/img/graphs/DFS/DFS-states-gray.pnggray: vertex is in progress;http://www.algolist.net/img/graphs/DFS/DFS-states-black.pngblack: DFS has finished processing the vertex.

Initially all vertices are white (unvisited). DFS starts in arbitrary vertex and runs as follows:

1. Mark vertex **u** as gray (visited).
2. For each edge **(u, v)**, where **u** is white, run depth-first search for **u** recursively.
3. Mark vertex **u** as black and backtrack to the parent.

***Task 3***

Use the following code and above given graph to perform depth first search on it. Print the DFS visited node order on the output screen.

void Graph::DFS() {

VertexState \*state = new VertexState[vertexCount];

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

state[i] = White;

runDFS(0, state);

delete [] state;

}

void Graph::runDFS(int u, VertexState state[]) {

state[u] = Gray;

for (int v = 0; v < vertexCount; v++)

if (isEdge(u, v) && state[v] == White)

runDFS(v, state);

state[u] = Black;

cout<<u<<",";

}

***Task 4***

Show BFS generated nodes for the same graph on the other side of this page. Also write your own bit of code to do BFS traversal on the same graph.

**EXERCISE**

1. The node that holds data is called and the line connecting two vertices is called edge.
2. If the value of each edge is given, the graph is called graph.
3. A graph is stored in the computer memory using the adjacency or adjacency .
4. There are two principal methods for graph traversal, which are and .
5. A is a sequence of vertices where each pair of successive vertices is connected by an edge.