Esau Hervert
eah170630

Assignment 4

Problem

1. Implement the DFS traversal of a graph.
   (a) Make an undirected graph with at least 10 vertices and 15 edges.
   (b) Pick a vertex to start DFS. Print the nodes visited during the DFS.
   (c) Repeat the step b using a directed graph.
   (d) repeat step b with undirected but disconnected graph.
2. Do BFS traversal for a graph with at least 10 vertices and 15 edges. You may use a queue library.
   (a) Repeat steps a,b,c,d from part 1 above.

Submit the code and submit the screenshots.

Graphs

Here are the following graphs that I traversed.

Graph 1

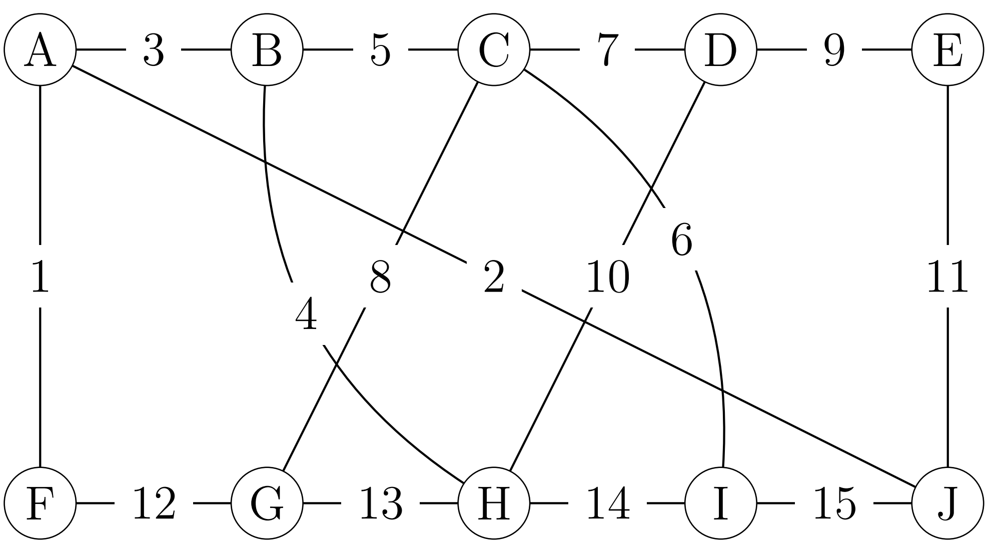
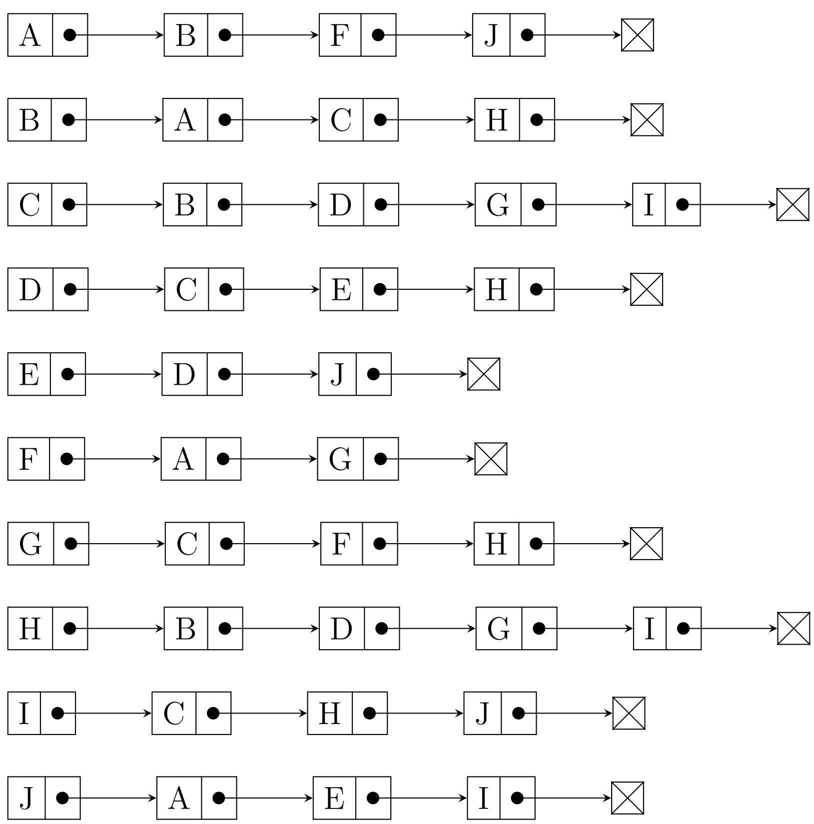


Figure 1: Graph 1

The Adjancency Matrix is as follows:

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$$

The LinkedList represetation of the graph is as follows:



Graph 2

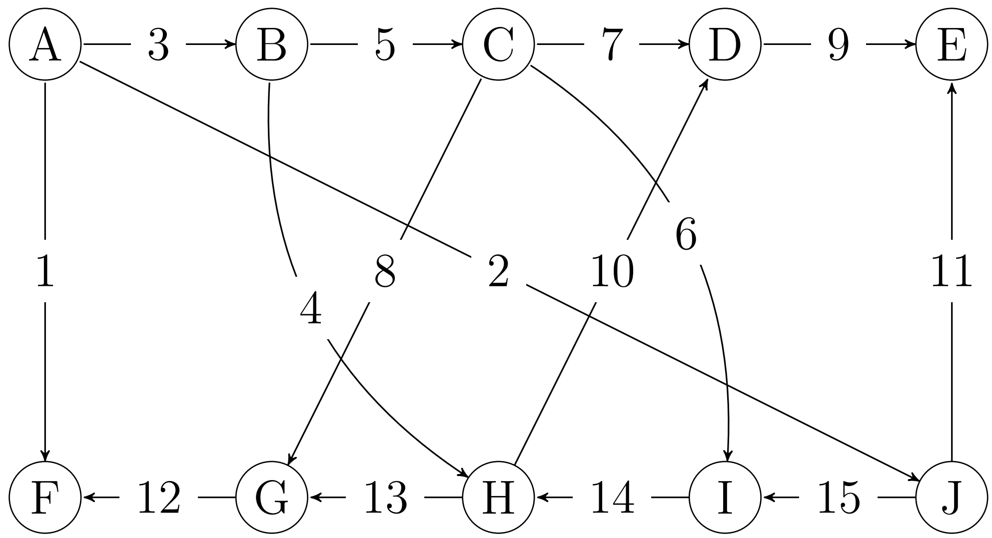
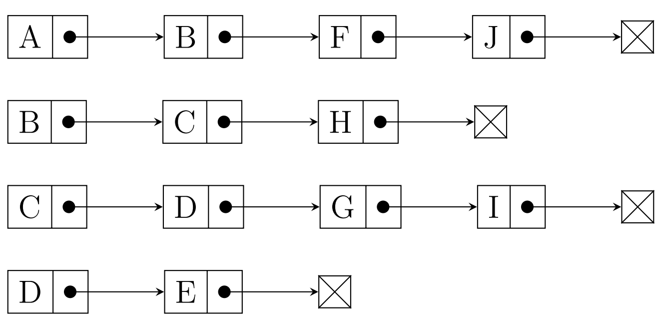


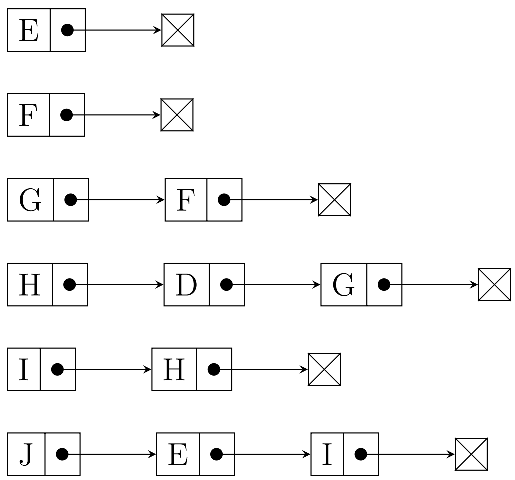
Figure 2: Graph 2

The Adjancency Matrix is as follows:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The LinkedList represetation of the graph is as follows:





Graph 3

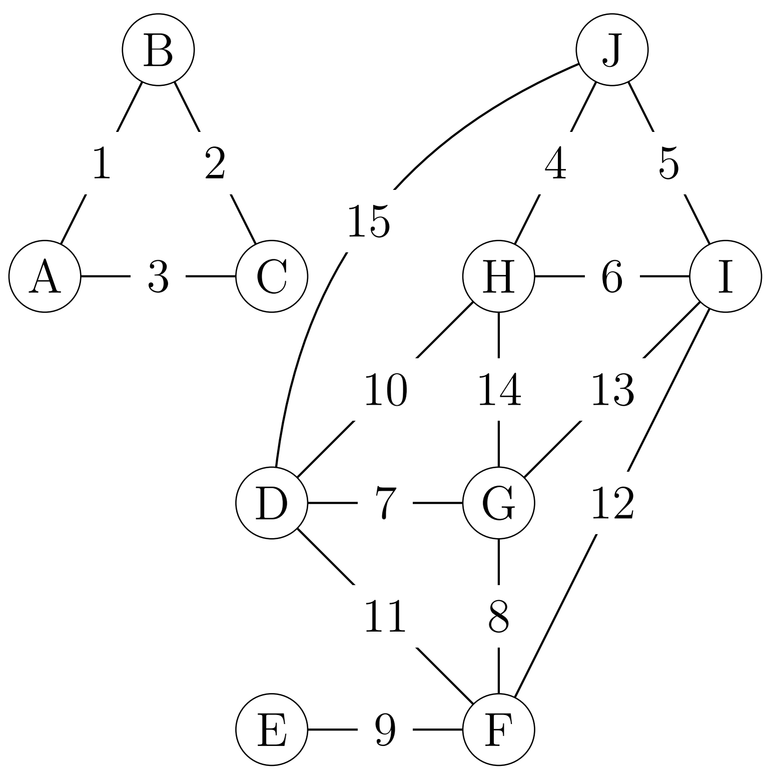
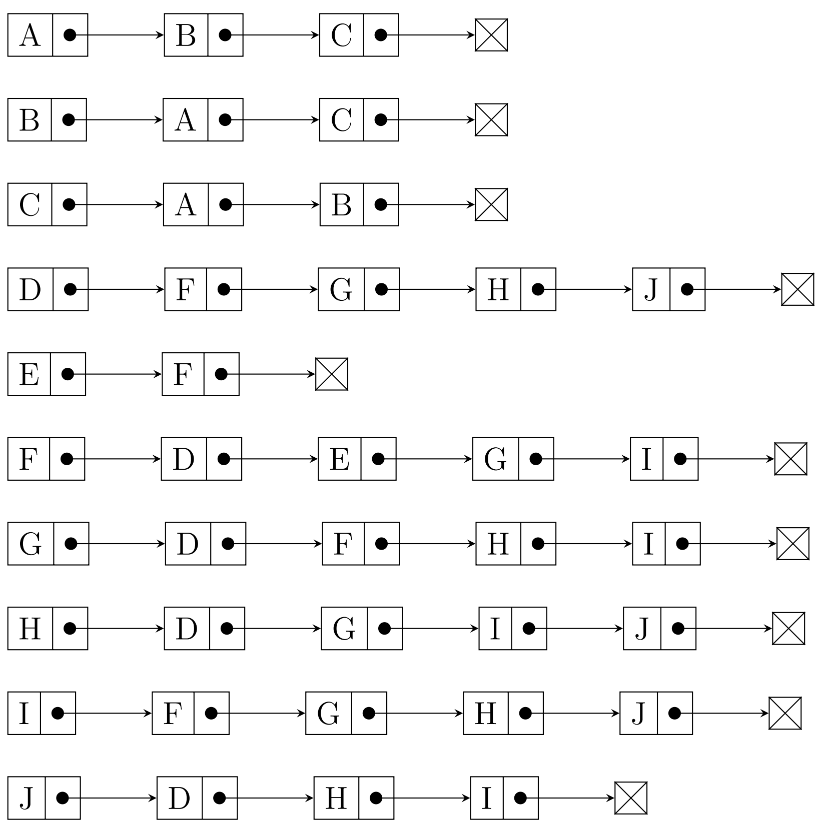


Figure 3: Graph 3

The Adjancency Matrix is as follows:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

The LinkedList represetation of the graph is as follows:



Code

The Code is broken up in the following way: there are two header files which contain all the functions, one for Depth First Search (DFS) and one for Breath First Search (BFS), and then two c++ files which contain the runs:

1. DFS for an Undirected Connected Graph (Graph 1), an Directed (Weakly) Connected Graph (Graph 2), and an Undirected Disconnected Graph (Graph 3)
2. BFS for an Undirected Connected Graph (Graph 1), an Directed (Weakly) Connected Graph (Graph 2), and an Undirected Disconnected Graph (Graph 3)

Code 1: DFS_Graph_Functions.h

```
#ifndef DFS_Graph_Functions
#define DFS_Graph_Functions

#include <iostream>

using namespace std;

// This will be used to make the linked lists.
struct Node {
        char vertex;
        Node* next;
};

class LinkedList {
public:
        Node *head = NULL;

        void add(char);
        void addarray(char *, int);
};

// Adding a vertex.
void LinkedList::add(char v) {
        Node *n = new Node();
        n->vertex = v;
        if (head == NULL) {
                head = n;
        }
        else
        {
                Node *temp = head;
                while (temp->next != NULL) {
                        temp = temp->next;
                }
                temp->next = n;
```

```
        }
}

// So that we can add all the vertices that are
// adjacent to L[i] efficiently.
void LinkedList::addarray(char *A,int n) {
        for (int i = 0; i < n; i++) {
                add(A[i]);
        }
}

// This will be used to traverse.
int index(char *A, int n, char c) {
        for (int i = 0; i < n; i++) {
                if (A[i] == c) {
                        return i;
                }
        }
        return -1;
}

// Here we recursively traverse the graph using DFS
// with the Array representation of a graph.
void DFSGraphTraversalA(int A[10][10], int i, bool *transit,
        bool *visited, char *vertices) {

        transit[i] = true;
        for (int j = 0; j < 10; j++) {
                if (j == i) {
                        continue;
                }
                else if (A[i][j] == 1) {
                        if (transit[j] == false) {
                                DFSGraphTraversalA(A, j, transit, visited,
                                        vertices);
                        }
                }
        }
        visited[i] = true;
        cout << vertices[i] << " ";
}

// This will be used to tell us if our traversal was
// complete in the sense that we traversed all the
// vertices.
int Complete(bool *visited) {
        for (int i = 0; i < 10; i++) {
                if (visited[i] == false) {
                        return i;
                }
```

```
        }
        return -1;
}

// Given that our original traversal algorithm does not take
// into consideration the fact that the graph may be disconnected,
// We must have a second algorithm that will traverse all the
// connected components of the graph.
void CDFSGraphTraversalA(int A[10][10], int i, bool *transit,
        bool *visited, char *vertices) {
        cout << "The First Connected Component is: ";
        DFSGraphTraversalA(A, i, transit, visited, vertices);

        // Here we check if we have traversed all the vertices.
        int j = Complete(visited);
        // If we have, then we are done.
        if (j == -1) {
                cout << endl;
                cout << "The Graph is Connected!" << endl;
        }
        // If we haven't, then we go to the first vertex that has not
        // been visited and do our traversal there to find its connected
        // component. We do this until there are no more vertices unvisited.
        else
        {
                cout << "\nThe Graph is Disconnected!" << endl;;
                cout << "The Other Connected Components are: " << endl;
                while (j != -1) {
                        cout << "Connected Component: ";
                        DFSGraphTraversalA(A, j, transit, visited, vertices);
                        j = Complete(visited);
                        cout << endl;
                }
        }
}

// Here we recursively traverse the graph using DFS
// with the Linked List representation of a graph.
void DFSGraphTraversalL(LinkedList *L, int i, bool *transit,
        bool *visited, char *vertices) {

        transit[i] = true;
        int j = 0;
        Node *temp = L[i].head->next;
        while (temp != NULL) {
                j = index(vertices, 10, temp->vertex);
                if (transit[j] == false) {
                        DFSGraphTraversalL(L, j, transit, visited, vertices);
                }
                temp = temp->next;
```

```
        }
        visited[i] = true;
        cout << vertices[i] << " ";
}

// Same arguments as for the Array representation.
void CDFSGraphTraversalL(LinkedList *L, int i, bool *transit,
        bool *visited, char *vertices) {
        cout << "The First Connected Component is: ";
        DFSGraphTraversalL(L, i, transit, visited, vertices);

        int j = Complete(visited);
        if (j == -1) {
                cout << endl;
                cout << "The Graph is Connected!" << endl;
        }
        else
        {
                cout << "\nThe Graph is Disconnected!" << endl;
                cout << "The Other Connected Components are: " << endl;
                while (j != -1) {
                        cout << "Connected Component: ";
                        DFSGraphTraversalL(L, j, transit, visited, vertices);
                        j = Complete(visited);
                        cout << endl;
                }
        }
}

#endif DFS_Graph_Functions
```

Code 2: BFS_Graph_Functions.h

```
#ifndef BFS_Graph_Functions
#define BFS_Graph_Functions

#include <iostream>
#include <queue>

using namespace std;

// This will be used to make the linked lists.
struct Node {
        char vertex;
        Node* next;
};

class LinkedList {
public:
        Node *head = NULL;

        void add(char);
        void addarray(char *, int);
};

// Adding a vertex.
void LinkedList::add(char v) {
        Node *n = new Node();
        n->vertex = v;
        if (head == NULL) {
                head = n;
        }
        else
        {
                Node *temp = head;
                while (temp->next != NULL) {
                        temp = temp->next;
                }
                temp->next = n;
        }
}

// So that we can add all the vertices that are
// adjacent to L[i] efficiently.
void LinkedList::addarray(char *A, int n) {
        for (int i = 0; i < n; i++) {
                add(A[i]);
        }
}

// This will be used to traverse.
int index(char *A, int n, char c) {
```

```
        for (int i = 0; i < n; i++) {
                if (A[i] == c) {
                        return i;
                }
        }
        return -1;
}

// Here we recursively traverse the graph using DFS
// with the Array representation of a graph.
void BFSGraphTraversalA(int A[10][10], int i, bool *visited,
        char *vertices) {

        // Start the queue with the starting node.
        queue<int> Traversal;
        visited[i] = true;
        Traversal.push(i);

        // We will add elements to the queue according to their
        // "level', i.e. going one traversal at a time.
        while (Traversal.empty() != true) {
                // We choose a vertex at level l and then add
                // univisited connected vertices and denote them as
                // level l+1.
                int j = Traversal.front();
                Traversal.pop();
                cout << vertices[j] << " ";
                for (int k = 0; k < 10; k++) {
                        if (k == j) {
                                continue;
                        }
                        if (A[j][k] == 1) {
                                if (visited[k] == false) {
                                        visited[k] = true;
                                        Traversal.push(k);
                                }
                        }
                }
        }
}

// This will be used to tell us if our traversal was
// complete in the sense that we traversed all the
// vertices.
int Complete(bool *visited) {
        for (int i = 0; i < 10; i++) {
                if (visited[i] == false) {
                        return i;
                }
        }
```

```
        return -1;
}

// Given that our original traversal algorithm does not take
// into consideration the fact that the graph may be disconnected,
// We must have a second algorithm that will traverse all the
// connected components of the graph.
void CBFSGraphTraversalA(int A[10][10], int i, bool *visited,
        char *vertices) {

        cout << "The First Connected Component is: ";
        BFSGraphTraversalA(A, i, visited, vertices);

        // Here we check if we have traversed all the vertices.
        int j = Complete(visited);
        // If we have, then we are done.
        if (j == -1) {
                cout << endl;
                cout << "The Graph is Connected!" << endl;
        }
        // If we haven't, then we go to the first vertex that has not
        // been visited and do our traversal there to find its connected
        // component. We do this until there are no more vertices unvisited.
        else
        {
                cout << "\nThe Graph is Disconnected!" << endl;;
                cout << "The Other Connected Components are: " << endl;
                while (j != -1) {
                        cout << "Connected Component: ";
                        BFSGraphTraversalA(A, j, visited, vertices);
                        j = Complete(visited);
                        cout << endl;
                }
        }
}

// Here we recursively traverse the graph using DFS
// with the Linked List representation of a graph.
void BFSGraphTraversalL(LinkedList *L, int i, bool *visited,
        char *vertices) {

        queue<int> Traversal;
        visited[i] = true;
        Traversal.push(i);

        while (Traversal.empty() != true) {
                int j = Traversal.front();
                Traversal.pop();
                cout << vertices[j] << " ";
```

```
                Node *temp = L[j].head->next;
                while (temp != NULL) {
                        int k = index(vertices, 10, temp->vertex);
                        if (visited[k] == false) {
                                Traversal.push(k);
                                visited[k] = true;
                        }
                        temp = temp->next;
                }

        }
}

// Same arguments as for the Array representation.
void CBFSGraphTraversalL(LinkedList *L, int i, bool *visited,
        char *vertices) {
        cout << "The First Connected Component is: ";
        BFSGraphTraversalL(L, i, visited, vertices);

        int j = Complete(visited);
        if (j == -1) {
                cout << endl;
                cout << "The Graph is Connected!" << endl;
        }
        else
        {
                cout << "\nThe Graph is Disconnected!" << endl;
                cout << "The Other Connected Components are: " << endl;
                while (j != -1) {
                        cout << "Connected Component: ";
                        BFSGraphTraversalL(L, j, visited, vertices);
                        j = Complete(visited);
                        cout << endl;
                }
        }
}

#endif BFS_Graph_Functions
```

Code 3: Assignment_4_DFS.cpp

```
#include "DFS_Graph_Functions.h"

#include <iostream>

using namespace std;

int main() {

        // Adjacency Matrix representation of Graph 1.
        int AG1[10][10] = {
        {0,1,0,0,0,1,0,0,0,1},
        {1,0,1,0,0,0,0,1,0,0},
        {0,1,0,1,0,0,1,0,1,0},
        {0,0,1,0,1,0,0,1,0,0},
        {0,0,0,1,0,0,0,0,0,1},
        {1,0,0,0,0,0,1,0,0,0},
        {0,0,1,0,0,1,0,1,0,0},
        {0,1,0,1,0,0,1,0,1,0},
        {0,0,1,0,0,0,0,1,0,1},
        {1,0,0,0,1,0,0,0,1,0} };

        //LinkedList Representation of Graph 1.
        LinkedList LG1[10];
        char A1[4] = { 'A','B','F','J'};
        char B1[4] = { 'B','A','C','H' };
        char C1[5] = { 'C','B','D','G','I' };
        char D1[4] = { 'D','C','E','H' };
        char E1[3] = { 'E','D','J' };
        char F1[3] = { 'F','A','G' };
        char G1[4] = { 'G','C','F','H' };
        char H1[5] = { 'H','B','D','G','I' };
        char I1[4] = { 'I','C','H','J' };
        char J1[4] = { 'J','A','E','I' };
        LG1[0].addarray(A1, 4);
        LG1[1].addarray(B1, 4);
        LG1[2].addarray(C1, 5);
        LG1[3].addarray(D1, 4);
        LG1[4].addarray(E1, 3);
        LG1[5].addarray(F1, 3);
        LG1[6].addarray(G1, 4);
        LG1[7].addarray(H1, 5);
        LG1[8].addarray(I1 , 4);
        LG1[9].addarray(J1 , 4);

        // Adjacency Matrix representation of Graph 2.
        int AG2[10][10] = {
        {0,1,0,0,0,1,0,0,0,1},
        {-1,0,1,0,0,0,0,1,0,0},
        {0,-1,0,1,0,0,1,0,1,0},
```

```
{0,0,-1,0,1,0,0,-1,0,0},
{0,0,0,-1,0,0,0,0,0,-1},
{-1,0,0,0,0,0,-1,0,0,0},
{0,0,-1,0,0,1,0,-1,0,0},
{0,-1,0,1,0,0,1,0,-1,0},
{0,0,-1,0,0,0,0,1,0,-1},
{-1,0,0,0,1,0,0,0,1,0} };

//LinkedList Representation of Graph 2.
LinkedList LG2[10];
char A2[4] = { 'A','B','F','J' };
char B2[3] = { 'B','C','H' };
char C2[4] = { 'C','D','G','I' };
char D2[2] = { 'D','E' };
char E2[1] = { 'E' };
char F2[1] = { 'F' };
char G2[2] = { 'G', 'F' };
char H2[3] = { 'H','D','G' };
char I2[2] = { 'I','H' };
char J2[3] = { 'J','E','I' };
LG2[0].addarray(A2, 4);
LG2[1].addarray(B2, 3);
LG2[2].addarray(C2, 4);
LG2[3].addarray(D2, 2);
LG2[4].addarray(E2, 1);
LG2[5].addarray(F2, 1);
LG2[6].addarray(G2, 2);
LG2[7].addarray(H2, 3);
LG2[8].addarray(I2, 2);
LG2[9].addarray(J2, 3);

// Adjacency Matrix representation of Graph 3.
int AG3[10][10] = {
{0,1,1,0,0,0,0,0,0,0},
{1,0,1,0,0,0,0,0,0,0},
{1,1,0,0,0,0,0,0,0,0},
{0,0,0,0,0,1,1,1,0,1},
{0,0,0,0,0,1,0,0,0,0},
{0,0,0,1,1,0,1,0,1,0},
{0,0,0,1,0,1,0,1,1,0},
{0,0,0,1,0,0,1,0,1,1},
{0,0,0,0,0,1,1,1,0,1},
{0,0,0,1,0,0,0,1,1,0} };

//LinkedList Representation of Graph 3.
LinkedList LG3[10];
char A3[3] = { 'A','B','C' };
char B3[3] = { 'B','A','C' };
char C3[3] = { 'C','A','B' };
char D3[5] = { 'D','F','G','H','J' };
```

```
char E3[2] = { 'E','F' };
char F3[5] = { 'F','D','E','G','I' };
char G3[5] = { 'G','D','F','H','I' };
char H3[5] = { 'H','D','G','I','J' };
char I3[5] = { 'I','F','G','H','J' };
char J3[4] = { 'J','D','H','I' };
LG3[0].addarray(A3, 3);
LG3[1].addarray(B3, 3);
LG3[2].addarray(C3, 3);
LG3[3].addarray(D3, 5);
LG3[4].addarray(E3, 2);
LG3[5].addarray(F3, 5);
LG3[6].addarray(G3, 5);
LG3[7].addarray(H3, 5);
LG3[8].addarray(I3, 5);
LG3[9].addarray(J3, 4);

// The vertices will be stored in this array.
char Vertices[10] = { 'A','B','C','D','E','F',
'G','H','I','J' };

bool Atransit1[10] = { 0,0,0,0,0,0,0,0,0,0 };
bool Avisited1[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 1 using Adjancency Matrix." << endl;
CDFSGraphTraversalA(AG1, 0, Atransit1, Avisited1, Vertices);

cout << endl;

bool Ltransit1[10] = { 0,0,0,0,0,0,0,0,0,0 };
bool Lvisited1[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 1 using Linked List." << endl;
CDFSGraphTraversalL(LG1, 0, Ltransit1, Lvisited1, Vertices);

cout << endl;

bool Atransit2[10] = { 0,0,0,0,0,0,0,0,0,0 };
bool Avisited2[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 2 using Adjancency Matrix." << endl;
CDFSGraphTraversalA(AG2, 0, Atransit2, Avisited2, Vertices);

cout << endl << endl;

bool Ltransit2[10] = { 0,0,0,0,0,0,0,0,0,0 };
bool Lvisited2[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 2 using Linked List." << endl;
CDFSGraphTraversalL(LG2, 0, Ltransit2, Lvisited2, Vertices);
```

```
    cout << endl;

    bool Atransit3[10] = { 0,0,0,0,0,0,0,0,0,0 };
    bool Avisited3[10] = { 0,0,0,0,0,0,0,0,0,0 };

    cout << "DFS traversal of Graph 3 using Adjancency Matrix." << endl;
    CDFSGraphTraversalA(AG3, 0, Atransit3, Avisited3, Vertices);

    cout << endl;

    bool Ltransit3[10] = { 0,0,0,0,0,0,0,0,0,0 };
    bool Lvisited3[10] = { 0,0,0,0,0,0,0,0,0,0 };

    cout << "DFS traversal of Graph 3 using Linked List." << endl;
    CDFSGraphTraversalL(LG3, 0, Ltransit3, Lvisited3, Vertices);
}
```

Code 4: Assignment_4_BFS.cpp

```
#include "BFS_Graph_Functions.h"

#include <iostream>

using namespace std;

int main() {

        // Adjacency Matrix representation of Graph 1.
        int AG1[10][10] = {
        {0,1,0,0,0,1,0,0,0,1},
        {1,0,1,0,0,0,0,1,0,0},
        {0,1,0,1,0,0,1,0,1,0},
        {0,0,1,0,1,0,0,1,0,0},
        {0,0,0,1,0,0,0,0,0,1},
        {1,0,0,0,0,0,1,0,0,0},
        {0,0,1,0,0,1,0,1,0,0},
        {0,1,0,1,0,0,1,0,1,0},
        {0,0,1,0,0,0,0,1,0,1},
        {1,0,0,0,1,0,0,0,1,0} };

        //LinkedList Representation of Graph 1.
        LinkedList LG1[10];
        char A1[4] = { 'A','B','F','J' };
        char B1[4] = { 'B','A','C','H' };
        char C1[5] = { 'C','B','D','G','I' };
        char D1[4] = { 'D','C','E','H' };
        char E1[3] = { 'E','D','J' };
        char F1[3] = { 'F','A','G' };
        char G1[4] = { 'G','C','F','H' };
        char H1[5] = { 'H','B','D','G','I' };
        char I1[4] = { 'I','C','H','J' };
        char J1[4] = { 'J','A','E','I' };
        LG1[0].addarray(A1, 4);
        LG1[1].addarray(B1, 4);
        LG1[2].addarray(C1, 5);
        LG1[3].addarray(D1, 4);
        LG1[4].addarray(E1, 3);
        LG1[5].addarray(F1, 3);
        LG1[6].addarray(G1, 4);
        LG1[7].addarray(H1, 5);
        LG1[8].addarray(I1, 4);
        LG1[9].addarray(J1, 4);

        // Adjacency Matrix representation of Graph 2.
        int AG2[10][10] = {
        {0,1,0,0,0,1,0,0,0,1},
        {-1,0,1,0,0,0,0,1,0,0},
        {0,-1,0,1,0,0,1,0,1,0},
```

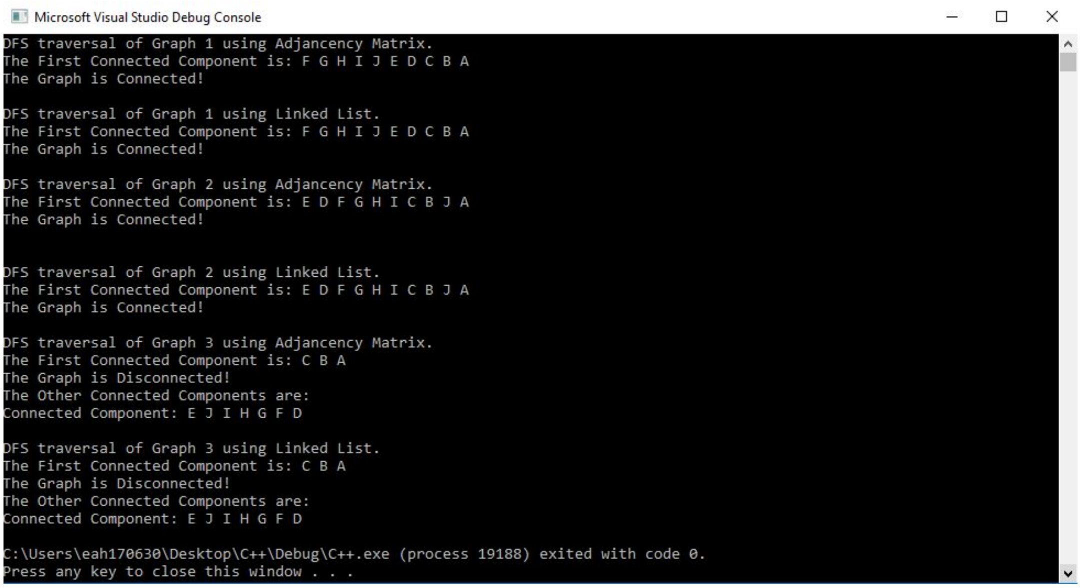
```
{0,0,-1,0,1,0,0,-1,0,0},
{0,0,0,-1,0,0,0,0,0,-1},
{-1,0,0,0,0,0,-1,0,0,0},
{0,0,-1,0,0,1,0,-1,0,0},
{0,-1,0,1,0,0,1,0,-1,0},
{0,0,-1,0,0,0,0,1,0,-1},
{-1,0,0,0,1,0,0,0,1,0} };

//LinkedList Representation of Graph 2.
LinkedList LG2[10];
char A2[4] = { 'A','B','F','J' };
char B2[3] = { 'B','C','H' };
char C2[4] = { 'C','D','G','I' };
char D2[2] = { 'D','E' };
char E2[1] = { 'E' };
char F2[1] = { 'F' };
char G2[2] = { 'G', 'F' };
char H2[3] = { 'H','D','G' };
char I2[2] = { 'I','H' };
char J2[3] = { 'J','E','I' };
LG2[0].addarray(A2, 4);
LG2[1].addarray(B2, 3);
LG2[2].addarray(C2, 4);
LG2[3].addarray(D2, 2);
LG2[4].addarray(E2, 1);
LG2[5].addarray(F2, 1);
LG2[6].addarray(G2, 2);
LG2[7].addarray(H2, 3);
LG2[8].addarray(I2, 2);
LG2[9].addarray(J2, 3);

// Adjacency Matrix representation of Graph 3.
int AG3[10][10] = {
{0,1,1,0,0,0,0,0,0,0},
{1,0,1,0,0,0,0,0,0,0},
{1,1,0,0,0,0,0,0,0,0},
{0,0,0,0,0,1,1,1,0,1},
{0,0,0,0,0,1,0,0,0,0},
{0,0,0,1,1,0,1,0,1,0},
{0,0,0,1,0,1,0,1,1,0},
{0,0,0,1,0,0,1,0,1,1},
{0,0,0,0,0,1,1,1,0,1},
{0,0,0,1,0,0,0,1,1,0} };

//LinkedList Representation of Graph 3.
LinkedList LG3[10];
char A3[3] = { 'A','B','C' };
char B3[3] = { 'B','A','C' };
char C3[3] = { 'C','A','B' };
char D3[5] = { 'D','F','G','H','J' };
```

```
char E3[2] = { 'E','F' };
char F3[5] = { 'F','D','E','G','I' };
char G3[5] = { 'G','D','F','H','I' };
char H3[5] = { 'H','D','G','I','J' };
char I3[5] = { 'I','F','G','H','J' };
char J3[4] = { 'J','D','H','I' };
LG3[0].addarray(A3, 3);
LG3[1].addarray(B3, 3);
LG3[2].addarray(C3, 3);
LG3[3].addarray(D3, 5);
LG3[4].addarray(E3, 2);
LG3[5].addarray(F3, 5);
LG3[6].addarray(G3, 5);
LG3[7].addarray(H3, 5);
LG3[8].addarray(I3, 5);
LG3[9].addarray(J3, 4);

// The vertices will be stored in this array.
char Vertices[10] = { 'A','B','C','D','E','F',
'G','H','I','J' };

bool Avisited1[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "BFS traversal of Graph 1 using Adjancency Matrix." << endl;
CBFSGraphTraversalA(AG1, 0, Avisited1, Vertices);

cout << endl;

bool Lvisited1[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "BFS traversal of Graph 1 using Linked List." << endl;
CBFSGraphTraversalL(LG1, 0, Lvisited1, Vertices);

cout << endl;

bool Avisited2[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 2 using Adjancency Matrix." << endl;
CBFSGraphTraversalA(AG2, 0, Avisited2, Vertices);

cout << endl;

bool Lvisited2[10] = { 0,0,0,0,0,0,0,0,0,0 };

cout << "DFS traversal of Graph 2 using Linked List." << endl;
CBFSGraphTraversalL(LG2, 0, Lvisited2, Vertices);

cout << endl;

bool Avisited3[10] = { 0,0,0,0,0,0,0,0,0,0 };
```

```
    cout << "DFS traversal of Graph 3 using Adjancency Matrix." << endl;
    CBFSGraphTraversalA(AG3, 0, Avisited3, Vertices);

    cout << endl;

    bool Lvisited3[10] = { 0,0,0,0,0,0,0,0,0,0 };

    cout << "DFS traversal of Graph 3 using Linked List." << endl;
    CBFSGraphTraversalL(LG3, 0, Lvisited3, Vertices);
}
```

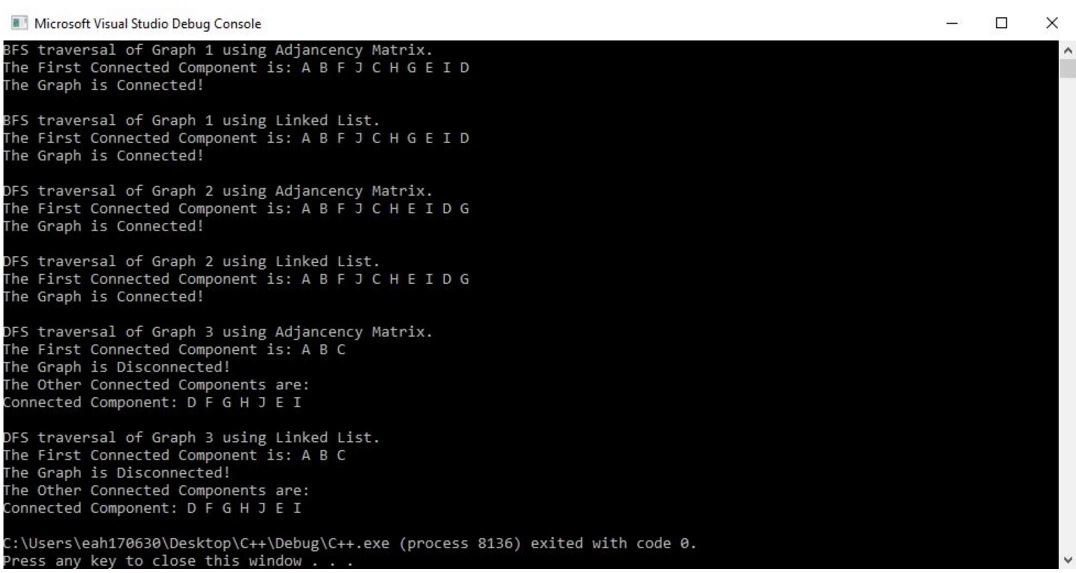
Results

The following are the runs that I did. Note that I did both algorithms using an Adjancency Matrix and Linked List representation of a graph. Given that I got the same results from both methods, it shows that my method is correct.



```
Microsoft Visual Studio Debug Console
DFS traversal of Graph 1 using Adjancency Matrix.
The First Connected Component is: F G H I J E D C B A
The Graph is Connected!

DFS traversal of Graph 1 using Linked List.
The First Connected Component is: F G H I J E D C B A
The Graph is Connected!

DFS traversal of Graph 2 using Adjancency Matrix.
The First Connected Component is: E D F G H I C B J A
The Graph is Connected!


DFS traversal of Graph 2 using Linked List.
The First Connected Component is: E D F G H I C B J A
The Graph is Connected!

DFS traversal of Graph 3 using Adjancency Matrix.
The First Connected Component is: C B A
The Graph is Disconnected!
The Other Connected Components are:
Connected Component: E J I H G F D

DFS traversal of Graph 3 using Linked List.
The First Connected Component is: C B A
The Graph is Disconnected!
The Other Connected Components are:
Connected Component: E J I H G F D

C:\Users\eah170630\Desktop\C++\Debug\C++.exe (process 19188) exited with code 0.
Press any key to close this window . . .
```

Figure 4: Depth First Search



```
Microsoft Visual Studio Debug Console
BFS traversal of Graph 1 using Adjancency Matrix.
The First Connected Component is: A B F J C H G E I D
The Graph is Connected!

BFS traversal of Graph 1 using Linked List.
The First Connected Component is: A B F J C H G E I D
The Graph is Connected!

DFS traversal of Graph 2 using Adjancency Matrix.
The First Connected Component is: A B F J C H E I D G
The Graph is Connected!

DFS traversal of Graph 2 using Linked List.
The First Connected Component is: A B F J C H E I D G
The Graph is Connected!

DFS traversal of Graph 3 using Adjancency Matrix.
The First Connected Component is: A B C
The Graph is Disconnected!
The Other Connected Components are:
Connected Component: D F G H J E I

DFS traversal of Graph 3 using Linked List.
The First Connected Component is: A B C
The Graph is Disconnected!
The Other Connected Components are:
Connected Component: D F G H J E I

C:\Users\eah170630\Desktop\C++\Debug\C++.exe (process 8136) exited with code 0.
Press any key to close this window . . .
```

Figure 5: Breath First Search