**PROJECT REPORT**

**BY:**

Antara Mahale

Darshana K

Shayanaa Mirsha R M

Simran Singh

**Finding connected components of a graph**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TOPIC** | **Pg** |
| **1.** | Aim |  |
| **2.** | Introduction |  |
| **3.** | Implementation |  |
| **4.** | Algorithm |  |
| **5.** | Code |  |
| **6.** | Samples |  |
| **7.** | Applications |  |
| **8.** | Conclusion |  |

**­­­­­­**

**AIM**

The aim of the project is to find the count of connected components in a graph and also display the vertices in each component.

**INTRODUCTION**

In a graph G, subgraph H is said to be a connected component if there exist a path between any two vertices in H and such that there doesn’t exist a path from any vertex in H to any vertex in G-H.

Eg: The following graph has three components.

Component b

Component a Component c

Fig 1.1

Taking vertex 1 :

Path to 2-> 1-2

Path to 5->1-2-5

Path to 6->1-2-5-6

Path to 7->1-7

As we can clearly see there exists a path from vertex 1 to vertex 2, 5, 6 and 7, but no path to vertex 3, 4 and 8. Therefore vertices 1,2,5,6 and 7 form one component.

Vertex 4 and 8 are connected therefore they form one component.

Vertex 3 is isolated and hence it forms one component.

In order to represent a graph in a program we have two popular methods, one using adjacency matrix and the other using adjacency list. And to traverse through a graph we can either use BFS-Breadth First Search or DFS-Depth First search.

**IMPLEMENTATION**

* **REPRESENTATION**

1. Using Adjacency Matrix

Adjacency matrix is a 2d matrix of order vxv, where v is the number of vertices in the graph. Let M be the adjacency matrix of graph G. By default M[i][j]=0 (where i,j=0,1,2,3…v-1). If there exists an edge between i and j then M[i][j] is set to 1. In case of weighted graph M[i][j] is set to w where w is the weight of edge (i,j).

Eg: The adjacency matrix for [Fig 1.1](#Fig1)

1 2 3 4 5 6 7 8

1 0 1 0 0 0 0 1 0

2 1 0 0 0 1 0 0 0

3 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 1

5 0 1 0 0 0 1 1 0

6 0 0 0 0 1 0 0 0

7 1 0 0 0 1 0 0 0

1. 0 0 0 1 0 0 0 0

2. Using Adjacency List

In this method an array of lists of size v is used where v is the number of vertices. Let L be the array L[i] (where i=0,1,2,…,n-1)is the list of vertices which form an edge with i.

Eg. The adjaceny list for [Fig 1.1](#Fig1)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  | 2 |  |  | 7 |  |  |  |  |
| 2 |  |  | 1 |  |  | 5 |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  | 8 |  |  |  |  |  |  |  |
| 5 |  |  | 2 |  |  | 6 |  |  | 7 |  |
| 6 |  |  | 5 |  |  |  |  |  |  |  |
| 7 |  |  | 6 |  |  |  |  |  |  |  |
| 8 |  |  | 4 |  |  |  |  |  |  |  |

* **TRAVERSAL**

In both cases, a visited array is maintained to keep track of the vertices that have been traversed through already.

1. Breadth First Search

A queue data structure is used. First the starting vertex is enqueued and visited[starting vertex] is set to 1.Then it is dequeued and its adjacent vertices are enqueued and marked as visited. Then the queue is dequeued and adjacent vertices of that vertex which aren’t visited are enqueued and the process repeated until the queue is empty.

Eg: Bfs of component a of [Fig 1.1](#Fig1)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

Visited=

1.Queue-> [1](starting vertex 1 is enqueued)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 |

Visited=

2.Queue-> [2,7] (1 has been dequeued and its adjacent vertices 2 and 7 are enqueued)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 |

Visited=

Output-> 1

3.Queue-> [7,5] (2 has been dequeued and its adjacent vertex 5 has been enqueued but 1 has already been visited so it is ignored)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 1 |

Visited=

Output-> 1 2

3.Queue-> [5] (7 has been dequeued and its adjacent vertices have already been visited)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 7

4.Queue-> [6] (5 has been dequeued and its only unvisited adjacent vertex 6 has been enqueued)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 7 5

3.Queue-> [] (empty queue-the process stops here)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 7 5 6

1. Depth First Search

It is a recursion based algorithm. It starts from a vertex ,marks it as visited, then traverses to one of its adjacent nodes marks it as visited and then traverses to an unvisited adjacent node of this vertex , so on until there is no unmarked adjacent nodes. Then backtracks and repeats the process until all the vertices are visited.

Eg:DFS of component a of [Fig 1.1](#Fig1)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

Visited=

1.Call stack-> [DFS(1)] (DFS(1) has been called, visited[1] is set to 1)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 |

Visited=

Output-> 1

2.Call Stack-> [DFS(2),DFS(1)] (2 being the first adjacent vertex of 1, DFS(2) is called and visited[2]=1)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 |

Visited=

Output-> 1 2

3.Call Stack-> [DFS(5),DFS(2),DFS(1)] (5 being the first unvisited adjacent vertex of 2, DFS(5) is called)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 0 |

Visited=

Output-> 1 2 5

4.Call Stack-> [DFS(6),DFS(5),DFS(2),DFS(1)](5 being the first unvisited adjacent vertex of 2, DFS(5) is called)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 |

Visited=

Output-> 1 2 5 6

5.Call Stack-> [DFS(5),DFS(2),DFS(1)](6’s adjacent vertex 5 has already been visited so the control is backtracked to the previous call)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 |

Visited=

Output-> 1 2 5 6

6.Call Stack-> [DFS(7),DFS(5),DFS(2),DFS(1)](7 an adjacent vertex of 5 hasn’t been visited so DFS(7) is called)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 5 6 7

7.Call Stack-> [DFS(5),DFS(2),DFS(1)](Adjacent vertex of 7 has already been visited so backtracking happens)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 5 6 7

8.Call Stack-> [DFS(2),DFS(1)](Adjacent vertices of 5 have already been visited so backtracking happens)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 5 6 7

9.Call Stack-> [DFS(1)](Adjacent vertices of 2 have already been visited so backtracking happens)

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 5 6 7

10.Call Stack-> Adjacent vertex of 1 have already been visited and there are no more functions in the call stack , so control returns to the main function and the program terminates.

1 2 5 6 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |

Visited=

Output-> 1 2 5 6 7

**ALGORITHM**

(We have used adjacency list and DFS in our project)

1.Input number of vertices-a and edges-b;

2.Created the visited array and set the values to 0, initiate counter c=0(no of components)

3.Create array of lists, say L.

4.Input b edges in the form of v1 v2(where v1 and v2 are vertices are connected by an edge).

5.Add v1 to the list L[v2] and v2 to the list L[v1]

6.The first unvisited vertex is taken as the starting vertex and dfs is performed and all the vertices connected to it are found . c is incremented by 1

7.Step 6 is repeated until all vertices are visited

8.Terminate when v[i]=1 for all i=0,1,2,3,…v-1

**CODE**

#include<stdio.h>

#include<stdlib.h>

int a, b;

struct node{

    int data;

    struct node\* nxt;

};

//adjacency list

void insert(struct node\*\*arr,int h, int data){

    struct node\* newnode= (struct node\*)malloc(sizeof(struct node));

    newnode->data=data;

    newnode->nxt=NULL;

    if(arr[h]==NULL){

        arr[h]=newnode;

        return;

    }

    struct node\* temp=arr[h];

    while(temp->nxt!=NULL){

        temp=temp->nxt;

    }

    temp->nxt=newnode;

}

struct node\*\* create\_ll(){

    struct node\*\* arr= (struct node\*\*)malloc(sizeof(struct node)\*a);

    for(int i=0; i<a; i++){

        arr[i]=NULL;

    }

    printf(“Enter the edges:\n”)

    for(int i=0; i<b; i++){

        int a, b;

        scanf(“%d %d”,&a, &b);

        insert(arr,a,b);

        insert(arr,b,a);

    }

    return arr;

}

//DFS

int \*visited;;

void DFS(int start, struct node\*\* arr){

    struct node\* temp=arr[start];

    visited[start]=1;

    printf(“%d “,start);

    if(temp!=NULL){

    while(temp->nxt !=NULL){

        if(visited[temp->data]==0)

           DFS(temp->data, arr);

        temp=temp->nxt;

    }

    if(visited[temp->data]==0)

      DFS(temp->data,arr);

    }

}

//to find starting vertex of next component

int next\_start(){

    for(int i=0; i<a; i++){

        if(visited[i]==0){

            return I;

        }

    }

    return -1;

}

int main(){

    struct node\*\* arr;

    printf(“Enter the no of vertices and edges:”);

    scanf(“%d %d”, &a, &b); //a=no\_vertices, b=no\_edges

    visited=(int \*)malloc(sizeof(int)\*a);

    arr=create\_ll(a,b);

    for(int i=0;i<a;i++)

    {

        visited[i]=0;

    }

    int k=0;

    int t=next\_start(a);

    printf(“CONNECTED COMPONENTS:\n”);

    while(t!=-1){

        DFS(t,arr);

        printf(“\n”);

        k++;

        t=next\_start(a);

    }

    printf(“NUMBER OF CONNECTED COMPONENTS: %d”, k);

    return 0;

}

**SAMPLE CASES**

Case 1:

Input:

Enter the number of vertices and edges:10 9

Enter the edges:

0 4

0 5

4 2

2 1

1 9

9 7

7 6

5 3

3 8

Output:

CONNECTED COMPONENTS:

0 4 2 1 9 7 6 5 3 8

NUMBER OF CONNECTED COMPONENTS:1

CASE 2:

INPUT:

Enter the number of vertices and edges:6 0

Enter the edges:

OUTPUT:

CONNECTED COMPONENTS:

0

1

2

3

4

5

NUMBER OF CONNECTED COMPONENTS:6

CASE 3:

INPUT:  
Enter the number of vertices and edges:8 6

Enter the edges:

1 2

2 5

5 6

5 7

1 7

4 0

OUTPUT:

CONNECTED COMPONENTS:  
0 4

1 2 5 6 7

3

NUMBER OF CONNECTED COMPONENTS:3

**APPLICATIONS**

1.Image analysis-Used to map and identify different objects in an image.

2. Vehicle routing applications

3. Social network analysis- To depict people on social networks who have common friends or interests

**CONCLUSION**

The program has been executed successfully and the outputs have been verified. With further improvement on the frontend the code can be used for image analysis and related purposes.