Chapter 41: Breadth-First Search

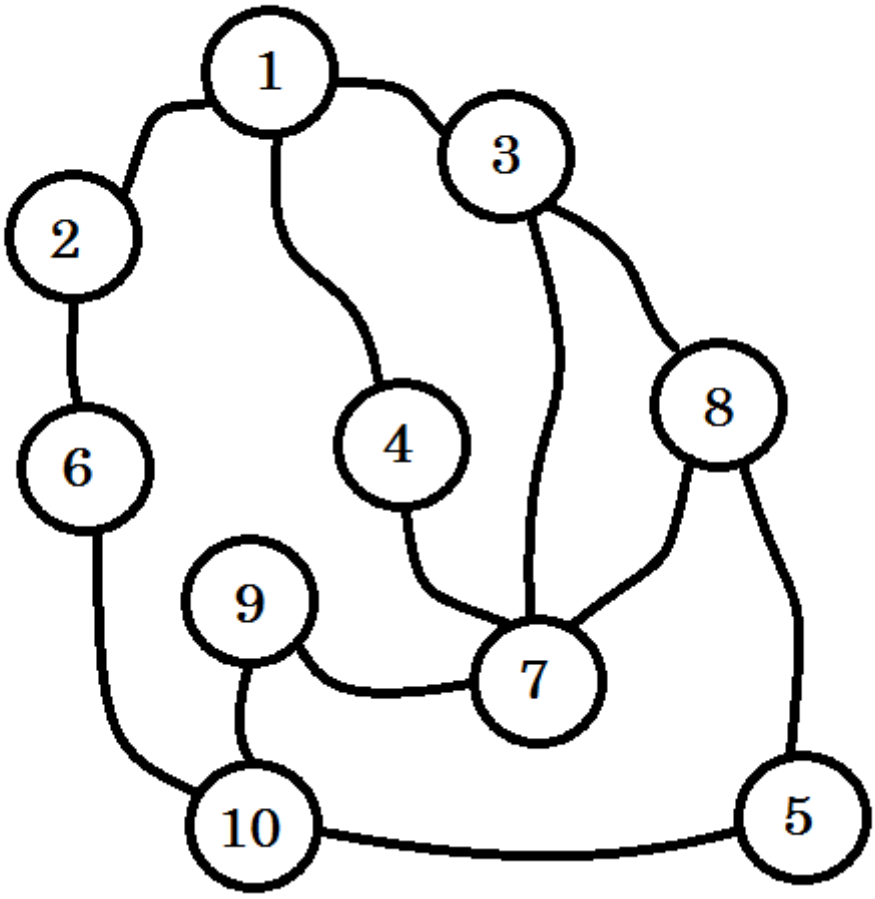
Section 41.1: Finding the Shortest Path from Source to other Nodes

Breadth-first-search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. BFS was invented in the late 1950s by Edward Forrest Moore, who used it to find the shortest path out of a maze and discovered independently by C. Y. Lee as a wire routing algorithm in 1961.

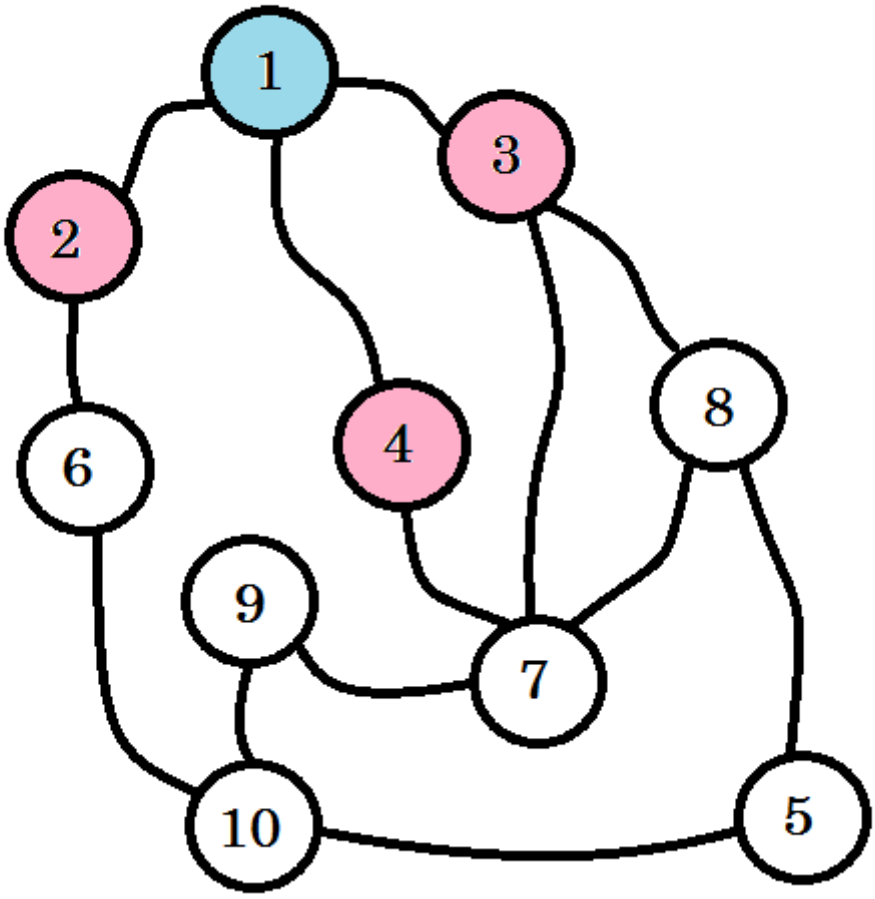
The processes of BFS algorithm works under these assumptions:

1. We won't traverse any node more than once.
2. Source node or the node that we're starting from is situated in level 0.
3. The nodes we can directly reach from source node are level 1 nodes, the nodes we can directly reach from level 1 nodes are level 2 nodes and so on.
4. The level denotes the distance of the shortest path from the source.

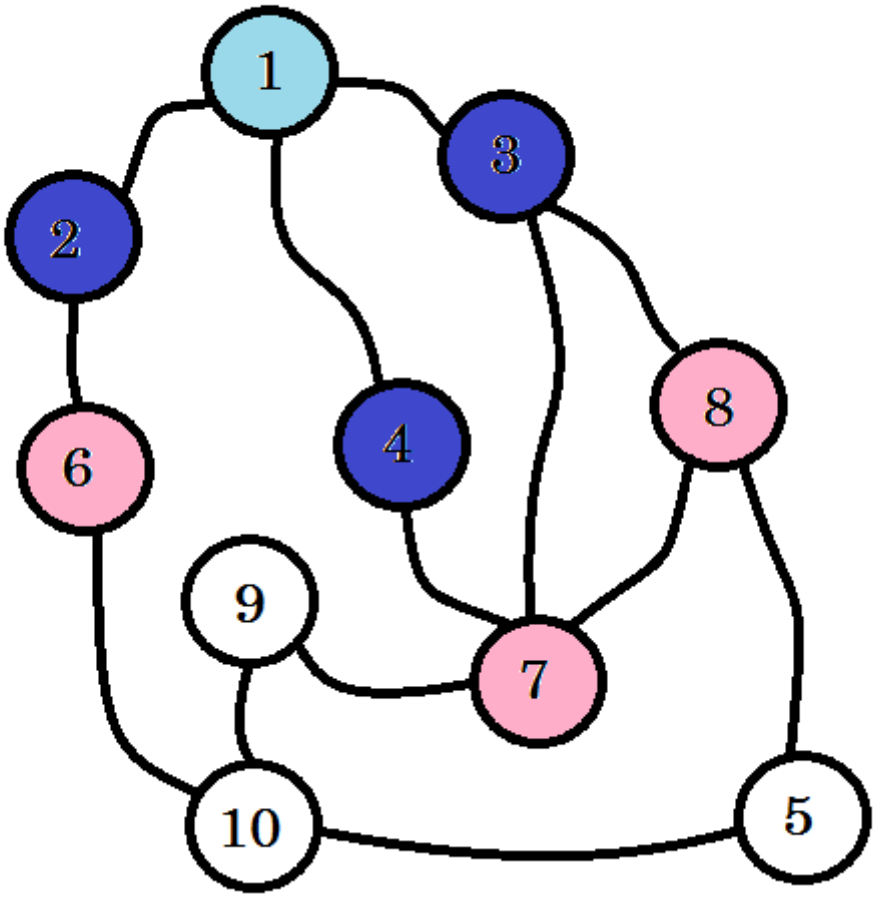
Let's see an example:



Let's assume this graph represents connection between multiple cities, where each node denotes a city and an edge between two nodes denote there is a road linking them. We want to go from **node 1** to **node 10**. So **node 1** is our **source**, which is **level 0**. We mark **node 1** as visited. We can go to **node 2**, **node 3** and **node 4** from here. So they'll be **level (0+1)** = **level 1** nodes. Now we'll mark them as visited and work with them.

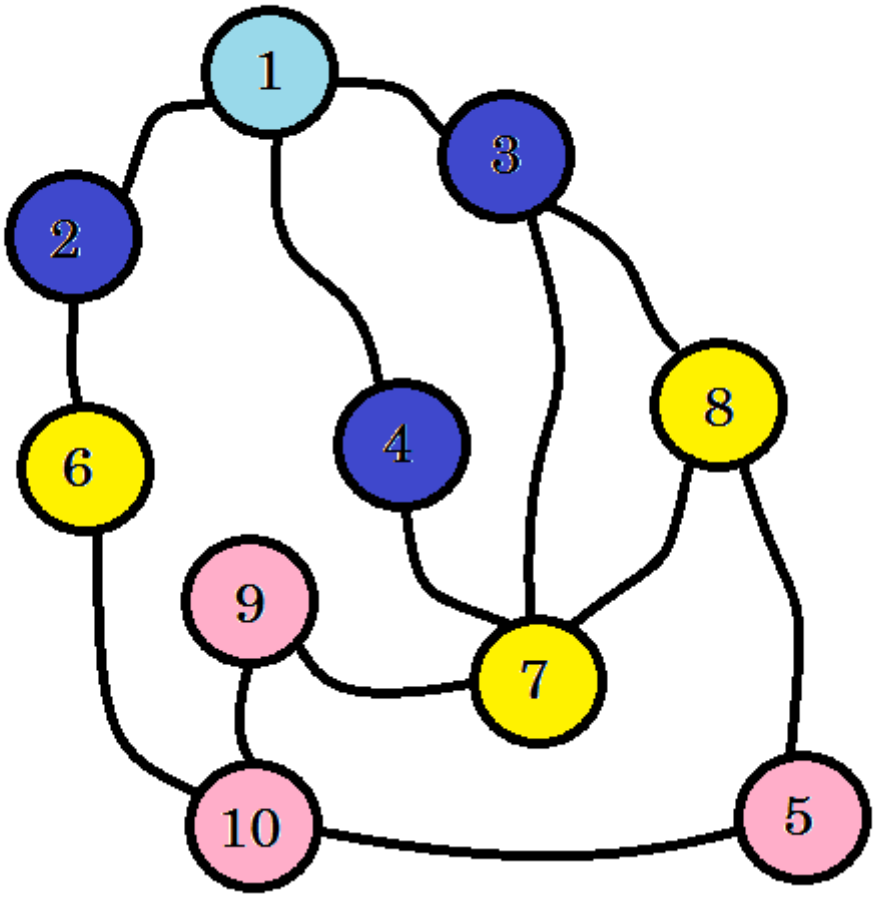


The colored nodes are visited. The nodes that we're currently working with will be marked with pink. We won't visit the same node twice. From **node 2**, **node 3** and **node 4**, we can go to **node 6, node 7** and **node 8**. Let's mark them as visited. The level of these nodes will be **level (1+1)** = **level 2**.

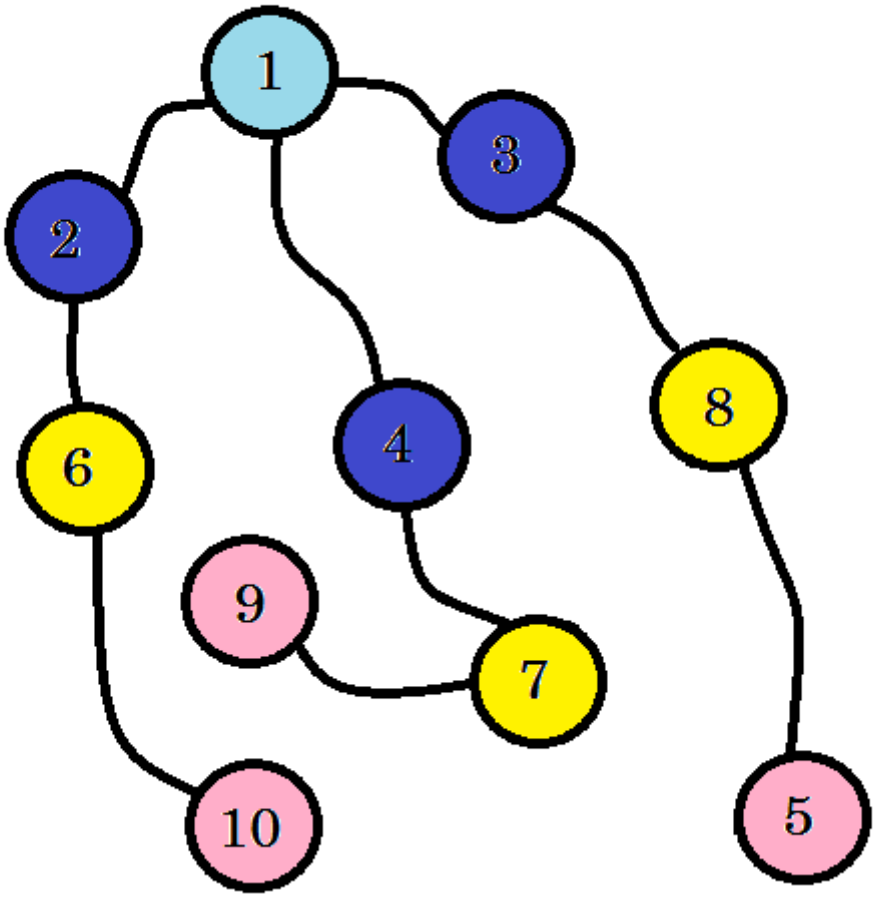


If you haven't noticed, the level of nodes simply denote the shortest path distance from the **source**. For example: we've found **node 8** on **level 2**. So the distance from **source** to **node 8** is **2**.

We didn't yet reach our target node, that is **node 10**. So let's visit the next nodes. we can directly go to from **node 6**, **node 7** and **node 8**.



We can see that, we found **node 10** at **level 3**. So the shortest path from **source** to **node 10** is **3.** We searched the graph level by level and found the shortest path. Now let's erase the edges that we didn't use:



After removing the edges that we didn't use, we get a tree called BFS tree. This tree shows the shortest path from **source** to all other nodes.

So our task will be, to go from **source** to **level 1** nodes. Then from **level 1** to **level 2** nodes and so on until we reach our destination. We can use *queue* to store the nodes that we are going to process. That is, for each node we're going to work with, we'll push all other nodes that can be directly traversed and not yet traversed in the queue.

The simulation of our example:

First we push the source in the queue. Our queue will look like:

```
 front
+-----+
|  1  |
+-----+
```

The level of **node 1** will be 0. **level[1]** = **0**. Now we start our BFS. At first, we pop a node from our queue. We get **node 1**. We can go to **node 4**, **node 3** and **node 2** from this one. We've reached these nodes from **node 1**. So **level[4]** = **level[3]** = **level[2]** = **level[1]** + **1** = **1**. Now we mark them as visited and push them in the queue.

```
                  front
+-----+  +-----+  +-----+
|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

Now we pop **node 4** and work with it. We can go to **node 7** from **node 4**. **level[7]** = **level[4]** + **1** = **2**. We mark **node 7** as visited and push it in the queue.

```
                  front
+-----+  +-----+  +-----+
|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

From **node 3**, we can go to **node 7** and **node 8**. Since we've already marked **node 7** as visited, we mark **node 8** as visited, we change **level[8]** = **level[3]** + **1** = **2**. We push **node 8** in the queue.

```
                  front
+-----+  +-----+  +-----+
|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

This process will continue till we reach our destination or the queue becomes empty. The **level** array will provide us with the distance of the shortest path from **source**. We can initialize **level** array with *infinity* value, which will mark that the nodes are not yet visited. Our pseudo-code will be:

```
Procedure BFS(Graph, source):
Q = queue();
level[] = infinity
level[source] := 0
Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
        end if
    end for
end while
Return level
```

By iterating through the **level** array, we can find out the distance of each node from source. For example: the distance of **node 10** from **source** will be stored in **level[10]**.

Sometimes we might need to print not only the shortest distance, but also the path via which we can go to our destined node from the **source**. For this we need to keep a **parent** array. **parent[source]** will be NULL. For each update in **level** array, we'll simply add `parent[v] := u` in our pseudo code inside the for loop. After finishing BFS, to find the path, we'll traverse back the **parent** array until we reach **source** which will be denoted by NULL value. The pseudo-code will be:

```
Procedure PrintPath(u):  //recursive
if parent[u] is not equal to null
    PrintPath(parent[u])
end if
print -> u
```

```
Procedure PrintPath(u):   //iterative
S =  Stack()
while parent[u] is not equal to null
    S.push(u)
    u := parent[u]
end while
while S is not empty
    print -> S.pop
end while
```

Complexity:

We've visited every node once and every edges once. So the complexity will be **O(V + E)** where **V** is the number of nodes and **E** is the number of edges.

Section 41.2: Finding Shortest Path from Source in a 2D graph

Most of the time, we'll need to find out the shortest path from single source to all other nodes or a specific node in a 2D graph. Say for example: we want to find out how many moves are required for a knight to reach a certain square in a chessboard, or we have an array where some cells are blocked, we have to find out the shortest path from one cell to another. We can move only horizontally and vertically. Even diagonal moves can be possible too. For these cases, we can convert the squares or cells in nodes and solve these problems easily using BFS. Now our **visited**, **parent** and **level** will be 2D arrays. For each node, we'll consider all possible moves. To find the distance to a specific node, we'll also check whether we have reached our destination.

There will be one additional thing called direction array. This will simply store the all possible combinations of directions we can go to. Let's say, for horizontal and vertical moves, our direction arrays will be:

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

Here *dx* represents move in x-axis and *dy* represents move in y-axis. Again this part is optional. You can also write all the possible combinations separately. But it's easier to handle it using direction array. There can be more and even different combinations for diagonal moves or knight moves.

The additional part we need to keep in mind is:

- If any of the cell is blocked, for every possible moves, we'll check if the cell is blocked or not.
- We'll also check if we have gone out of bounds, that is we've crossed the array boundaries.
- The number of rows and columns will be given.

Our pseudo-code will be:

```
Procedure BFS2D(Graph, blocksign, row, column):
for i from 1 to row
    for j from 1 to column
        visited[i][j] := false
    end for
end for
visited[source.x][source.y] := true
level[source.x][source.y] := 0
Q = queue()
Q.push(source)
m := dx.size
while Q is not empty
    top := Q.pop
    for i from 1 to m
        temp.x := top.x + dx[i]
        temp.y := top.y + dy[i]
        if temp is inside the row and column and top doesn't equal to blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
```

```
        end if
    end for
end while
Return level
```

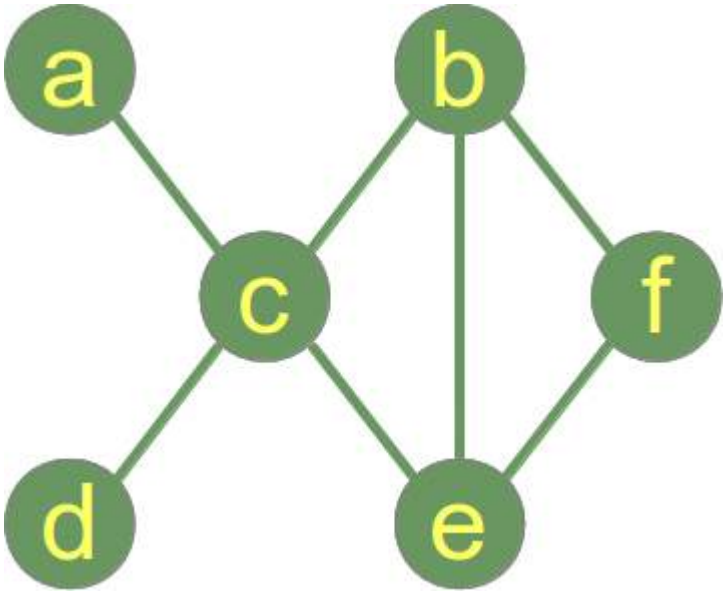
As we have discussed earlier, BFS only works for unweighted graphs. For weighted graphs, we'll need Dijkstra's algorithm. For negative edge cycles, we need Bellman-Ford's algorithm. Again this algorithm is single source shortest path algorithm. If we need to find out distance from each nodes to all other nodes, we'll need Floyd-Warshall's algorithm.

Section 41.3: Connected Components Of Undirected Graph Using BFS

BFS can be used to find the connected components of an undirected graph. We can also find if the given graph is connected or not. Our subsequent discussion assumes we are dealing with undirected graphs.The definition of a connected graph is:

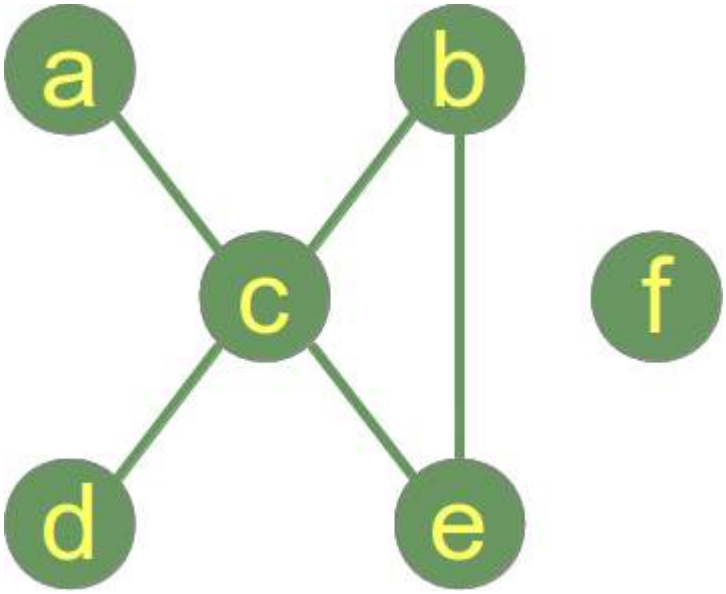
> A graph is connected if there is a path between every pair of vertices.

Following is a **connected graph**.



Following graph is **not connected** and has 2 connected components:

1. Connected Component 1: {a,b,c,d,e}
2. Connected Component 2: {f}



BFS is a graph traversal algorithm. So starting from a random source node, if on termination of algorithm, all nodes are visited, then the graph is connected,otherwise it is not connected.

PseudoCode for the algorithm.

```
boolean isConnected(Graph g)
{
 BFS(v)//v is a random source node.
 if(allVisited(g))
 {
  return true;
 }
 else return false;
}
```

C implementation for finding the whether an undirected graph is connected or not:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//Queue node depicts a single Queue element
//It is NOT a graph node.
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//array that keeps track of visited vertices.

int main()
```

```
{
    int n,e;//n is number of vertices, e is number of edges.
    int i,j;
    char **graph;//adjacency matrix

    printf("Enter number of vertices:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "Please enter a valid positive integer from 1 to %d",MAXVERTICES);
     return -1;
    }

    graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
        graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//initially all vertices are not visited.
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("enter number of edges and then enter them in pairs:");
    scanf("%d",&e);

    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("The graph is connected");
    else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if(Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int deque()
{
```

```
    if(Qfront == NULL)
    {
        printf("Q is empty , returning -1\n");
        return -1;
    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
            Qfront = Qfront->next;
            Qrear = NULL;
        }
        else
            Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //let random source vertex be vertex 0;
    BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 implies false;

    return 1;//1 implies true;
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
    visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

For Finding all the Connected components of an undirected graph, we only need to add 2 lines of code to the BFS function. The idea is to call BFS function until all vertices are visited.

The lines to be added are:

```
printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function
```

AND

```
printf("%d ",vertex+1);
add this as first line of while loop in BFS
```

and we define the following function:

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```