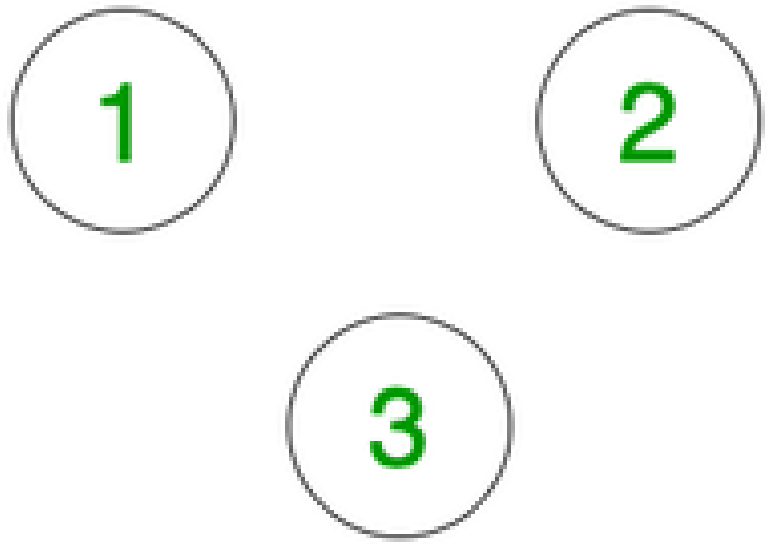
Find if there is a path between two vertices in an undirected graph

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

Given an **undirected graph** with **N** vertices and **E** edges and two vertices **(U, V)** from the graph, the task is to detect if a path exists between these two vertices. Print *"Yes"* if a path exists and *"No"* otherwise.

Examples:



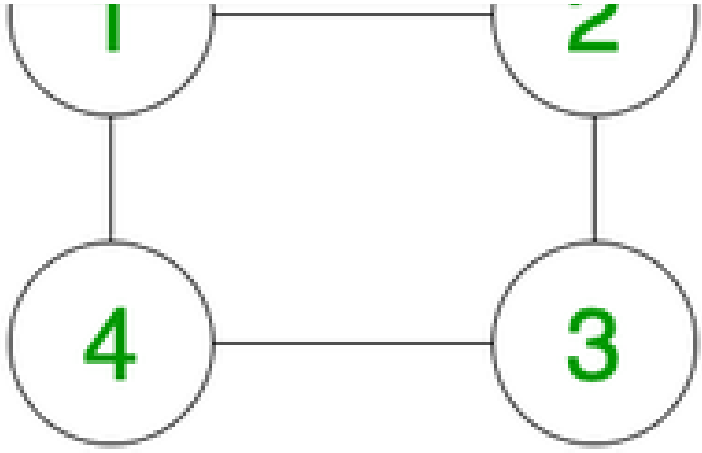
U = 1, V = 2

Output: *No*

Explanation:

There is no edge between the two points and hence its not possible to reach 2 from 1.

Input:



U = 1, V = 3

Output: *Yes*

Explanation: *Vertex 3 from vertex 1 via vertices 2 or 4.*

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

Naive Approach:

The idea is to use Floyd Warshall Algorithm. To solve the problem, we need to try out all intermediate vertices ranging **[1, N]** and check:

1. If there is a direct edge already which exists between the two nodes.
2. Or we have a path from node **i** to intermediate node **k** and from node **k** to node **j**.

Below is the implementation of the above approach:

C++

```
// C++ program to check if there is exist a path between
// two vertices of an undirected graph.

#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;
```

```
        adj[i][i]=1;
}

// Function to add edge between nodes
void addEdge(int a,int b)
{
    adj[a][b]=1;
    adj[b][a]=1;
}

// Function to compute the path
void computePaths(int n)
{
    // Use Floyd Warshall algorithm
    // to detect if a path exists
    for(int k = 1; k <= n; k++)
    {
        // Try every vertex as an
        // intermediate vertex
        // to check if a path exists
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                adj[i][j] = adj[i][j] | (adj[i][k] && adj[k][j]);
    }
}

// Function to check if nodes are reachable
bool isReachable(int s, int d)
{
    if (adj[s][d] == 1)
        return true;
    else
        return false;
}

// Driver Code
int main()
{
    int n = 4;
    adj = vector<vector<int>>(n+1,vector<int>(n+1,0));

    init(n);

    addEdge(1,2);
    addEdge(2,3);
    addEdge(1,4);

    computePaths(n);
```

```
    return 0;
}
```

Java

```
// Java program to detect if a path
// exists between any two vertices
// for the given undirected graph
import java.util.Arrays;

class GFG{

// Class representing a undirected
// graph using matrix representation
static class Graph
{
    int V;
    int[][] g;

    public Graph(int V)
    {
        this.V = V;

        // Rows may not be contiguous
        g = new int[V + 1][V + 1];
        for(int i = 0; i < V + 1; i++)
        {

            // Initialize all entries
            // as false to indicate
            // that there are
            // no edges initially
            Arrays.fill(g[i], 0);
        }

        // Initializing node to itself
        // as it is always reachable
        for(int i = 1; i <= V; i++)
            g[i][i] = 1;
```



```
    // Function to add edge between nodes
    void addEdge(int v, int w)
    {
        g[v][w] = 1;
        g[w][v] = 1;
```

```
            return true;
        else
            return false;
    }

    // Function to compute the path
    void computePaths()
    {

        // Use Floyd Warshall algorithm
        // to detect if a path exists
        for(int k = 1; k <= V; k++)
        {

            // Try every vertex as an
            // intermediate vertex
            // to check if a path exists
            for(int i = 1; i <= V; i++)
            {
                for(int j = 1; j <= V; j++)
                    g[i][j] = g[i][j] | ((g[i][k] != 0 &&
                              g[k][j] != 0) ? 1 : 0);
            }
        }
    }
};

// Driver code
public static void main(String[] args)
{
    Graph _g = new Graph(4);
    _g.addEdge(1, 2);
    _g.addEdge(2, 3);
    _g.addEdge(1, 4);
    _g.computePaths();

    int u = 4, v = 3;
    if (_g.isReachable(u, v))
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by sanjeev2552
```

```
# Class representing a undirected
# graph using matrix
# representation
class Graph:

    def __init__(self, V):

        self.V = V

        # Initialize all entries
        # as false to indicate
        # that there are
        # no edges initially
        self.g = [[0 for j in range(self.V + 1)]
                     for i in range(self.V + 1)]

        # Initializing node to itself
        # as it is always reachable
        for i in range(self.V + 1):
            self.g[i][i] = 1

    # Function to add edge between nodes
    def addEdge(self, v, w):

        self.g[v][w] = 1
        self.g[w][v] = 1

    # Function to compute the path
    def computePaths(self):

        # Use Floyd Warshall algorithm
        # to detect if a path exists
        for k in range(1, self.V + 1):

            # Try every vertex as an
            # intermediate vertex
            # to check if a path exists
            for i in range(1, self.V + 1):
                for j in range(1, self.V + 1):
                    self.g[i][j] = (self.g[i][j] |
                                    (self.g[i][k] and
                                     self.g[k][j]))

    # Function to check if nodes
    # are reachable
    def isReachable(self, s, d):

        if (self.g[s][d] == 1):
```

```
    _g = Graph(4)
    _g.addEdge(1, 2)
    _g.addEdge(2, 3)
    _g.addEdge(1, 4)
    _g.computePaths()

    u = 4
    v = 3

    if (_g.isReachable(u, v)):
        print('Yes')
    else:
        print('No')

# This code is contributed by rutvik_56
```

C#

```
// C# program to detect if a path
// exists between any two vertices
// for the given undirected graph
using System;

public class GFG {

    // Class representing a undirected
    // graph using matrix representation
    public

        class Graph {
        public

            int V;
        public

            int[, ] g;

        public Graph(int V)
        {
            this.V = V;

            // Rows may not be contiguous
            g = new int[V + 1, V + 1];
            for (int i = 0; i < V + 1; i++) {
```

```
        }

        // Initializing node to itself
        // as it is always reachable
        for (int i = 1; i <= V; i++)
            g[i, i] = 1;
    }

    // Function to add edge between nodes
    public void addEdge(int v, int w)
    {
        g[v, w] = 1;
        g[w, v] = 1;
    }

    // Function to check if nodes are reachable
    public bool isReachable(int s, int d)
    {
        if (g[s, d] == 1)
            return true;
        else
            return false;
    }

    // Function to compute the path
    public void computePaths()
    {

        // Use Floyd Warshall algorithm
        // to detect if a path exists
        for (int k = 1; k <= V; k++) {

            // Try every vertex as an
            // intermediate vertex
            // to check if a path exists
            for (int i = 1; i <= V; i++) {
                for (int j = 1; j <= V; j++)
                    g[i, j] = g[i, j]
                              | ((g[i, k] != 0
                                  && g[k, j] != 0)
                                     ? 1
                                     : 0);
            }
        }
    }
};

// Driver code
```

```
        _g.computePaths();

        int u = 4, v = 3;
        if (_g.isReachable(u, v))
            Console.WriteLine("Yes");
        else
            Console.WriteLine("No");
    }
}

// This code is contributed by umadevi9616
```

Javascript

```
<script>
// Javascript program to detect if a path
// exists between any two vertices
// for the given undirected graph

// Class representing a undirected
// graph using matrix representation
class Graph
{
    constructor(V)
    {
        this.V = V;

        // Rows may not be contiguous
        this.g = new Array(V + 1);
        for(let i = 0; i < V + 1; i++)
        {
            this.g[i] = new Array(V+1);


            // Initialize all entries
            // as false to indicate
            // that there are
            // no edges initially
            for(let j = 0; j < (V + 1); j++)
            {
                this.g[i][j] = 0;
            }

        }

        // Initializing node to itself
```

```
    addEdge(v, w)
    {
        this.g[v][w] = 1;
        this.g[w][v] = 1;
    }

    // Function to check if nodes are reachable
    isReachable(s, d)
    {
        if (this.g[s][d] == 1)
            return true;
        else
            return false;
    }

    // Function to compute the path
    computePaths()
    {
        // Use Floyd Warshall algorithm
        // to detect if a path exists
        for(let k = 1; k <= this.V; k++)
        {

            // Try every vertex as an
            // intermediate vertex
            // to check if a path exists
            for(let i = 1; i <= this.V; i++)
            {
                for(let j = 1; j <= this.V; j++)
                    this.g[i][j] = this.g[i][j] | ((this.g[i][k] != 0 &&
                                this.g[k][j] != 0) ? 1 : 0);
            }
        }
    }

}

// Driver code
let _g = new Graph(4);
    _g.addEdge(1, 2);
    _g.addEdge(2, 3);
    _g.addEdge(1, 4);
    _g.computePaths();

    let u = 4, v = 3;
    if (_g.isReachable(u, v))
        document.write("Yes<br>");
    else
```

Output

```
Yes
```

Time Complexity: $O(V^3)$

Auxiliary Space: $O(V^2)$

Efficient Solution

We can either use BFS or DFS to find if there is a path from u to v. Below is a BFS based solution

C++

```
// C++ program to check if there is exist a path between
// two vertices of an undirected graph.

#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> adj;

// function to add an edge to graph
void addEdge(int v,int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}

// A BFS based function to check whether d is reachable from s.
bool isReachable(int s,int d)
{
    // Base case
    if(s == d)
        return true;

    int n= (int)adj.size();

    // Mark all the vertices as not visited
    vector<bool> visited(n,false);

    // Create a queue for BFS
    queue<int> q;

    // Mark the current node as visited and enqueue it
```

```
        q.pop();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it
        // visited  and enqueue it
        for(auto x:adj[s])
        {

            // If this adjacent node is the destination node,
            // then return true
            if(x == d)
                return true;

            // Else, continue to do BFS
            if(!visited[x])
            {
                visited[x] = true;
                q.push(x);
            }
        }
    }

 // If BFS is complete without visiting d
    return false;
}

// Driver program to test methods of graph class
int main()
{
    int n = 4;
    // Create a graph in the above diagram
    adj = vector<vector<int>>(n);

    addEdge(0,1);
    addEdge(0,2);
    addEdge(1,2);
    addEdge(2,0);
    addEdge(2,3);
    addEdge(3,3);

    int u = 1, v = 3;
    if (isReachable(u, v))
        cout << "\n There is a path from " << u << " to " << v;
    else
        cout << "\n There is no path from " << u << " to " << v;

    return 0;
}
```

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

// Java program to check if there is exist a path between
// two vertices of an undirected graph.
public class Graph {
    // This class represents an undirected graph
    // using adjacency list representation
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    ArrayList<ArrayList<Integer>> adj;

    Graph(int V){
        this.V = V;
        adj = new ArrayList<>();
        for(int i=0;i<V;i++)
            adj.add(new ArrayList<>());
    }

    // function to add an edge to graph
    void addEdge(int v, int w)
    {
        adj.get(v).add(w);
        adj.get(w).add(v);
    }


    // A BFS based function to check whether d is reachable from s.
    boolean isReachable(int s, int d)
    {
        // Base case
        if (s == d)
            return true;

        // Mark all the vertices as not visited
        boolean[] visited = new boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Create a queue for BFS
        Queue<Integer> queue = new LinkedList<>();

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.add(s);
```

```
            // visited  and enqueue it
            for (int i=0; i<adj.get(s).size();i++) {

                // If this adjacent node is the destination node,
                // then return true
                if (adj.get(s).get(i) == d)
                return true;

                // Else, continue to do BFS
                if (!visited[adj.get(s).get(i)]) {
                    visited[adj.get(s).get(i)] = true;
                    queue.add(adj.get(s).get(i));
                }
            }
        }

        // If BFS is complete without visiting d
        return false;
    }

    // Driver program to test methods of graph class
    public static void main(String[] args)
    {

        // Create a graph given in the above diagram
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        int u = 1, v = 3;
        if (g.isReachable(u, v))
            System.out.println("\n There is a path from "+u+" to "+v);
        else
            System.out.println("\n There is no path from "+u+" to "+v);

    }

}

// This code is contributed by hritikrommie.
```

Python3

```
    adj[v].append(w)
    adj[w].append(v)

# A BFS based function to check whether d is reachable from s.
def isReachable(s, d):

    # Base case
    if (s == d):
        return True

    # Mark all the vertices as not visited
    visited = [False for i in range(V)]

    # Create a queue for BFS
    queue = deque()

    # Mark the current node as visited and enqueue it
    visited[s] = True
    queue.append(s)

    while (len(queue) > 0):

        # Dequeue a vertex from queue and print
        s = queue.popleft()
        # queue.pop_front()

        # Get all adjacent vertices of the dequeued vertex s
        # If a adjacent has not been visited, then mark it
        # visited  and enqueue it
        for i in adj[s]:

            # If this adjacent node is the destination node,
            # then return true
            if (i == d):
                return True

            # Else, continue to do BFS
            if (not visited[i]):
                visited[i] = True
                queue.append(i)
    # If BFS is complete without visiting d
    return False

# Driver program to test methods of graph class
if __name__ == '__main__':

    # Create a graph given in the above diagram
    V = 4
```

```
    addEdge(3, 3)
    u,v = 1, 3
    if (isReachable(u, v)):
        print("There is a path from",u,"to",v)
    else:
        print("There is no path from",u,"to",v)

        # This code is contributed by mohit kumar 29.
```

C#

```
using System;
using System.Collections.Generic;


// C# program to check if there is exist a path between
// two vertices of an undirected graph.
public class Graph
{

    // This class represents an undirected graph
    // using adjacency list representation
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    List<List<int>> adj;

    Graph(int V){
        this.V = V;
        adj = new List<List<int>>();
        for(int i = 0; i < V; i++)
            adj.Add(new List<int>());
    }

    // function to add an edge to graph
    void addEdge(int v, int w)
    {
        adj[v].Add(w);
        adj[w].Add(v);
    }


    // A BFS based function to check whether d is reachable from s.
    bool isReachable(int s, int d)
    {
        // Base case
```

```
            visited[i] = false;

        // Create a queue for BFS
        Queue<int> queue = new Queue<int>();

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.Enqueue(s);

        while (queue.Count != 0)
        {

            // Dequeue a vertex from queue and print it
            s = queue.Dequeue();

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited  and enqueue it
            for (int i = 0; i < adj[s].Count; i++) {

                // If this adjacent node is the destination node,
                // then return true
                if (adj[s][i] == d)
                return true;

                // Else, continue to do BFS
                if (!visited[adj[s][i]]) {
                    visited[adj[s][i]] = true;
                    queue.Enqueue(adj[s][i]);
                }
            }
        }

        // If BFS is complete without visiting d
        return false;
    }

    // Driver program to test methods of graph class
    public static void Main(String[] args)
    {

        // Create a graph given in the above diagram
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
```

```
            Console.WriteLine("\n There is no path from "+u+" to "+v);
    }
}

// This code is contributed by umadevi9616
```

Javascript

```
<script>
// javascript program to check if there is exist a path between
// two vertices of an undirected graph.

    // This class represents an undirected graph
    // using adjacency list representation
    var V; // No. of vertices

    // Pointer to an array containing adjacency lists
    var adj;
 V = 4;
        adj = new Array();
        for (var i = 0; i < V; i++)
            adj.push(new Array());


    // function to add an edge to graph
    function addEdge(v , w) {
        adj[v].push(w);
        adj[w].push(v);
    }

    // A BFS based function to check whether d is reachable from s.
    function isReachable(s , d) {
        // Base case
        if (s == d)
            return true;

        // Mark all the vertices as not visited
        var visited = new Array(V).fill(false);


        // Create a queue for BFS
        var queue = new Array();

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.push(s);
```

```
            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            for (var i = 0; i < adj[s].length; i++) {

                // If this adjacent node is the destination node,
                // then return true
                if (adj[s][i] == d)
                    return true;

                // Else, continue to do BFS
                if (!visited[adj[s][i]]) {
                    visited[adj[s][i]] = true;
                    queue.push(adj[s][i]);
                }
            }
        }

        // If BFS is complete without visiting d
        return false;
    }

    // Driver program to test methods of graph class

        // Create a graph given in the above diagram
        addEdge(0, 1);
        addEdge(0, 2);
        addEdge(1, 2);
        addEdge(2, 0);
        addEdge(2, 3);
        addEdge(3, 3);

        var u = 1, v = 3;
        if (isReachable(u, v))
            document.write("\n There is a path from " + u + " to " + v);
        else
            document.write("\n There is no path from " + u + " to " + v);

// This code is contributed by gauravrajput1
</script>
```

Output

```
There is a path from 1 to 3
```

C++

```
// C++ program to check if there is exist a path between
// two vertices of an undirected graph.

#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> graph;

void addEdge(int v,int w)
{
    graph[v].push_back(w);
    graph[w].push_back(v);
}

 bool dfs(vector<int> adj[], vector<int> &vis, int start, int end);
 bool validPath(int n, vector<vector<int>>& edges, int start, int end);

 bool validPath(int n, vector<vector<int>>& edges, int start, int end) {
        vector<int> adj[n];
        for(int i=0; i<edges.size(); i++){
            int u = edges[i][0];
            int v = edges[i][1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }

        vector <int> vis(n, 0);
        for(int i=0; i<n; i++)
            if(vis[i] == 0)
                if(dfs(adj, vis, start, end))
                    return true;

        return false;
    }

 bool dfs(vector<int> adj[], vector<int> &vis, int start, int end){
        if(end == start)
            return true;

        vis[start] = 1;
        for(auto it: adj[start])
            if(vis[it]==0)
                if(dfs(adj, vis, it, end))
                    return true;
```

```
    // Create a graph in the above diagram
    graph = vector<vector<int>>(n);

    addEdge(0,1);
    addEdge(0,2);
    addEdge(1,2);
    addEdge(2,0);
    addEdge(2,3);
    addEdge(3,3);

    int u = 1, v = 3;
    if (validPath(n, graph, u, v))
        cout << "\n There is a path from " << u << " to " << v;
    else
        cout << "\n There is no path from " << u << " to " << v;

    return 0;
}
```

Python3

```
# Python program for the above approach:
graph = []

def addEdge(v, w):
    global graph
    graph[v].append(w)
    graph[w].append(v)

def dfs(adj, vis, start, end):
    if(end == start):
        return True

    vis[start] = 1;

    for it in adj[start]:
        if(vis[it]==0):
            if(dfs(adj, vis, it, end)):
                return True

    return False

def validPath(n, edges, start, end):
    adj = [[] for _ in range(n)]

    for i in range(len(edges)):
```

```
    for i in range(n):
        if(vis[i] == 0):
            if(dfs(adj, vis, start, end)):
                return True

    return False

## Driver code

n = 4
## Create a graph in the above diagram
graph = [[] for _ in range(n)];

addEdge(0,1)
addEdge(0,2)
addEdge(1,2)
addEdge(2,0)
addEdge(2,3)
addEdge(3,3)

u = 1
v = 3
if (validPath(n, graph, u, v)):
    print("There is a path from", u, "to", v)
else:
    print("There is no path from", u, "to", v)

    # This code is contributed by subhamgoyal2014.
```

Output

```
There is a path from 1 to 3
```

RECOMMENDED ARTICLES

Page : 1 2 3

01 Find if there is a path between two vertices in a directed graph | Set 2
07, Jul 20

02 Find if there is a path between two vertices in a directed graph
10, Apr 12

03 Convert the undirected graph into directed graph such that there is no path of length greater than 1
05, Apr 19

04 Construct a graph using N vertices whose shortest distance between K pair of vertices is 2
04, Jan 21

05 Find K vertices in the graph which are connected to at least one of remaining vertices
01, Jul 19

06 Maximize the number of uncolored vertices appearing along the path from root vertex and the colored vertices
18, Apr 20

07 Minimize cost to color all the vertices of an Undirected Graph
23, Jul 20

08 Minimize cost to color all the vertices of an Undirected Graph using given operation
27, Jul 20

mehant
@mehant

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Improved By : rutvik_56, sanjeev2552, mohit kumar 29, unknown2108, hritikrommie, ashutoshsinghgeeksforgeeks, sweetyty, umadevi9616, GauravRajput1, simranarora5sos, aggarwalsajal19, Shikhar Jaiswal 1, subhamgoyal2014

Article Tags : BFS, graph-connectivity, Algorithms, Data Structures, Dynamic Programming, Graph

Practice Tags : Algorithms, BFS, Data Structures, Dynamic Programming, Graph

Improve Article | Report Issue

Files

main.py

graph.png

CPU RAM Storage

main.py

```python
# Find if two nodes in a undirected acyclic graph are connected
from typing import List, Tuple


def is_connected(edges: List[Tuple[str, str]], start: str, end: str) -> bool:
    # Your code here

    #undirected

    if start == end:
        return True

    nodes = []

    for e in edges:
        if e[0] not in nodes:
            nodes.append(e[0])
        if e[1] not in nodes:
            nodes.append(e[1])

    visited = {}
    neighbors = {}

    for n in nodes:
        visited[n] = False
        neighbors[n] = []

    queue = []

    queue.append(str)

    visited[str] = True

    #building the lookup of neighbors of nodes

    for e in edges:
        neighbors[e[0]].append(e[1])
        neighbors[e[1]].append(e[0])

    while (len(queue) > 0):

        s = queue.leftpop()

        for neighbor in neighbors[s]:
            if neighbor == end:
                return True

            if visited[neighbor] is False:
                visited[neighbor] = True
                queue.append(neighbor)

    return False


#Example case setup & run
edges = [("c", "d"), ("a", "b"), ("c", "b"), ("f", "g"), ("c", "e")]
connected = is_connected(edges, "e", "a")
print(connected)

```

Console Shell

```
False
Traceback (most recent call last):
  File "main.py", line 48, in <module>
    queue.push(1)
AttributeError: 'list' object has no attribute 'push'
```