roduction

hm for
em

atching

n graph is

Backtracking-5

acktracking-6

-7

cktracking-3

n N-Queen

n for Knight's

lem |

Rat in a Maze | Backtracking-2

Count number of ways to reach destination in a Maze

Count all possible paths from top left to bottom right of a mXn matrix

Print all possible paths from top left to bottom right of a mXn matrix

Unique paths in a Grid with Obstacles

Unique paths covering every non-obstacle block exactly once in a grid

Depth First Search or DFS for a Graph

Breadth First Search or BFS for a Graph

Level Order Tree Traversal

Tree Traversals (Inorder, Preorder and Postorder)

Inorder Tree Traversal without Recursion

Inorder Tree Traversal without recursion and without stack!

Print Postorder traversal from given Inorder and Preorder traversals

Construct Tree from given Inorder and Preorder traversals

Real-time application of Data Structures

Breadth First Search without using Queue

How Coronavirus outbreak can end | Visualize using Data structures

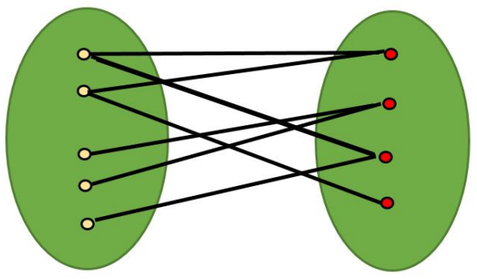
Shortest path in a directed graph by Dijkstra's algorithm

Flood Fill Algorithm

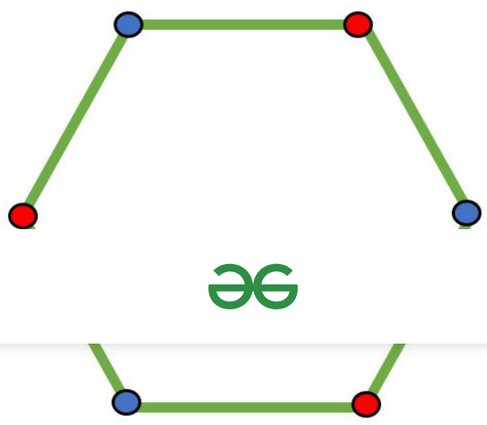
Check whether a given graph is Bipartite or not

Last Updated: 07-01-2020

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

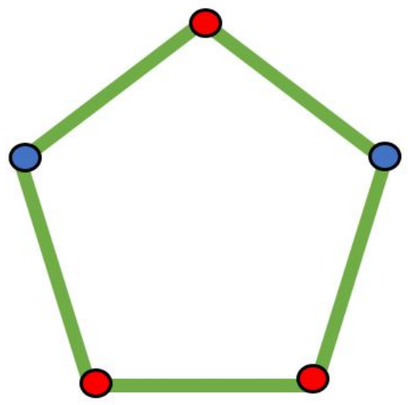


A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



Cycle graph of length 6

It is not possible to color a cycle graph with odd cycle using two colors.



Cycle graph of length 5

Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using backtracking algorithm m coloring problem.

Following is a simple algorithm to find out whether a given graph is Birpartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

```cpp
// C++ program to find out whether a
// given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4

using namespace std;

// This function returns true if graph
// G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors
    // assigned to all veritces. Vertex
    // number is used as index in this array.
    // The value '-1' of colorArr[i]
    // is used to indicate that no color
    // is assigned to vertex 'i'. The value 1
    // is used to indicate first color
    // is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex
    // numbers and enqueue source vertex
    // for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices
    // in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();

        // Return false if there is a self-loop
        if (G[u][u] == 1)
        return false;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and destination
            // v is colored with same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
            }
        }

        // If we reach here, then all adjacent
        // vertices can be colored with alternate color
        return true;
    }

    // Driver program to test above function
    int main()
    {
        int G[][V] = {{0, 1, 0, 1},
            {1, 0, 1, 0},
            {0, 1, 0, 1},
            {1, 0, 1, 0}
        };

        isBipartite(G, 0) ? cout << "Yes" : cout << "No";
        return 0;
    }
```

Java

```java
// Java program to find out whether
// a given graph is Bipartite or not
import java.util.*;
import java.lang.*;
import java.io.*;

class Bipartite
{
    final static int V = 4; // No. of Vertices

    // This function returns true if
    // graph G[V][V] is Bipartite, else false
    boolean isBipartite(int G[][],int src)
    {
        // Create a color array to store
        // colors assigned to all veritces.
        // Vertex number is used as index
        // in this array. The value '-1'
        // of colorArr[i] is used to indicate
        // that no color is assigned
        // to vertex 'i'. The value 1 is
        // used to indicate first color
        // is assigned and value 0 indicates
        // second color is assigned.
        int colorArr[] = new int[V];
        for (int i=0; i<V; ++i)
            colorArr[i] = -1;

        // Assign first color to source
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers
        // and enqueue source vertex for BFS traversal
        LinkedList<Integer>q = new LinkedList<Integer>();
```

```
        while (q.size() != 0)
        {
            // Dequeue a vertex from queue
            int u = q.poll();

            // Return false if there is a self-loop
            if (G[u][u] == 1)
                return false;

            // Find all non-colored adjacent vertices
            for (int v=0; v<V; ++v)
            {
                // An edge from u to v exists
                // and destination v is not colored
                if (G[u][v]==1 && colorArr[v]==-1)
                {
                    // Assign alternate color to this adjacent v of u
                    colorArr[v] = 1-colorArr[u];
                    q.add(v);
                }

                // An edge from u to v exists and destination
                //  v is colored with same color as u
                else if (G[u][v]==1 && colorArr[v]==colorArr[u])
                    return false;
            }
        }
        // If we reach here, then all adjacent vertices can
        // be colored with alternate color
        return true;
    }
```

```
    // Driver program to test above function
    public static void main (String[] args)
    {
        int G[][] = {{0, 1, 0, 1},
            {1, 0, 1, 0},
            {0, 1, 0, 1},
            {1, 0, 1, 0}
        };
        Bipartite b = new Bipartite();
        if (b.isBipartite(G, 0))
        System.out.println("Yes");
        else
        System.out.println("No");
    }
}

// Contributed by Aakash Hasija
```

Python

```
# Python program to find out whether a
# given graph is Bipartite or not

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = [[0 for column in range(V)] \
                      for row in range(V)]

# This function returns true if graph G[V][V]
# is Bipartite, else false
def isBipartite(self, src):

    # Create a color array to store colors
    # assigned to all veritces. Vertex
    # number is used as index in this array.
    # The value '-1' of  colorArr[i] is used to
    # indicate that no color is assigned to
    # vertex 'i'. The value 1 is used to indicate
    # first color is assigned and value 0
    # indicates second color is assigned.
    colorArr = [-1] * self.V

    # Assign first color to source
    colorArr[src] = 1

    # Create a queue (FIFO) of vertex numbers and
    # enqueue source vertex for BFS traversal
    queue = []
    queue.append(src)

    # Run while there are vertices in queue
    # (Similar to BFS)
    while queue:

        u = queue.pop()

        # Return false if there is a self-loop
        if self.graph[u][u] == 1:
            return False;

        for v in range(self.V):

            # An edge from u to v exists and destination
            # v is not colored
            if self.graph[u][v] == 1 and colorArr[v] == -1:

                # Assign alternate color to this
                # adjacent v of u
                colorArr[v] = 1 - colorArr[u]
                queue.append(v)

            # An edge from u to v exists and destination
            # v is colored with same color as u
            elif self.graph[u][v] == 1 and colorArr[v] == colorArr[u]:
                return False

    # If we reach here, then all adjacent
    # vertices can be colored with alternate
    # color
    return True

# Driver program to test above function
g = Graph(4)
g.graph = [[0, 1, 0, 1],
           [1, 0, 1, 0],
```

```
print "Yes" if g.isBipartite(0) else "No"

# This code is contributed by Divyanshu Mehta
```

C#

```
// C# program to find out whether
// a given graph is Bipartite or not
using System;
using System.Collections.Generic;

class GFG
{
    readonly static int V = 4; // No. of Vertices

    // This function returns true if
    // graph G[V,V] is Bipartite, else false
    bool isBipartite(int [,]G, int src)
    {
        // Create a color array to store
        // colors assigned to all veritces.
        // Vertex number is used as index
        // in this array. The value '-1'
        // of colorArr[i] is used to indicate
        // that no color is assigned
        // to vertex 'i'. The value 1 is
        // used to indicate first color
        // is assigned and value 0 indicates
// second color is assigned.
int []colorArr = new int[V];
for (int i = 0; i < V; ++i)
    colorArr[i] = -1;

// Assign first color to source
colorArr[src] = 1;

// Create a queue (FIFO) of vertex numbers
// and enqueue source vertex for BFS traversal
List<int>q = new List<int>();
q.Add(src);

// Run while there are vertices
// in queue (Similar to BFS)
while (q.Count != 0)
{
    // Dequeue a vertex from queue
    int u = q[0];
    q.RemoveAt(0);

    // Return false if there is a self-loop
    if (G[u, u] == 1)
        return false;

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists
        // and destination v is not colored
        if (G[u, v] == 1 && colorArr[v] == -1)
        {
            // Assign alternate color
            // to this adjacent v of u
            colorArr[v] = 1 - colorArr[u];
            q.Add(v);
        }

        // An edge from u to v exists and
        // destination v is colored with
        // same color as u
        else if (G[u, v] == 1 &&
                 colorArr[v] == colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices
// can be colored with alternate color
return true;
}

// Driver Code
public static void Main(String[] args)
{
    int [,]G = {{0, 1, 0, 1},
                {1, 0, 1, 0},
                {0, 1, 0, 1},
                {1, 0, 1, 0}};
    GFG b = new GFG();
    if (b.isBipartite(G, 0))
        Console.WriteLine("Yes");
    else
        Console.WriteLine("No");
}
}

// This code is contributed by Rajput-Ji
```

Output:

```
Yes
```

The above algorithm works only if the graph is connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipiartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly call above method for all not yet visited vertices.

C++

```
// C++ program to find out whether
// a given graph is Bipartite or not.
// It works for disconnected graph also.
#include <bits/stdc++.h>
```

```
// This function returns true if
// graph G[V][V] is Bipartite, else false
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers a
    // nd enqueue source vertex for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();

        // Return false if there is a self-loop
        if (G[u][u] == 1)
        return false;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this
                // adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and destination
            // v is colored with same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent vertices can
    // be colored with alternate color
    return true;
}

// Returns true if G[][] is Bipartite, else false
bool isBipartite(int G[][V])
{
    // Create a color array to store colors assigned to all
    // veritces. Vertex/ number is used as index in this
    // array. The value '-1' of colorArr[i] is used to
    // ndicate that no color is assigned to vertex 'i'.
    // The value 1 is used to indicate first color is
    // assigned and value 0 indicates second color is
    // assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // This code is to handle disconnected graoh
    for (int i = 0; i < V; i++)
    if (colorArr[i] == -1)
        if (isBipartiteUtil(G, i, colorArr) == false)
        return false;

    return true;
}

// Driver program to test above function
int main()
{
    int G[][V] = {{0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0}
    };

    isBipartite(G) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Java

```
// JAVA Code to check whether a given
// graph is Bipartite or not
import java.util.*;

class Bipartite {

public static int V = 4;

// This function returns true if graph
// G[V][V] is Bipartite, else false
public static boolean isBipartiteUtil(int G[][], int src,
                                      int colorArr[])
{
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and
    // enqueue source vertex for BFS traversal
    LinkedList<Integer> q = new LinkedList<Integer>();
    q.add(src);

    // Run while there are vertices in queue
    // (Similar to BFS)
    while (!q.isEmpty())
    {
        // Dequeue a vertex from queue
        // ( Refer http://goo.gl/35oz8 )
        int u = q.getFirst();
        q.pop();

        // Return false if there is a self-loop
        if (G[u][u] == 1)
           return false;
```

```
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] ==1 && colorArr[v] == -1)
            {
                // Assign alternate color to this
                // adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and
            // destination v is colored with same
            // color as u
            else if (G[u][v] ==1 && colorArr[v] ==
                                    colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent vertices
    // can be colored with alternate color
    return true;
}

// Returns true if G[][] is Bipartite, else false
public static boolean isBipartite(int G[][])
{
    // Create a color array to store colors assigned
    // to all veritces. Vertex/ number is used as
    // index in this array. The value '-1' of
    // colorArr[i] is used to indicate that no color
    // is assigned to vertex 'i'. The value 1 is used
    // to indicate first color is assigned and value
    // 0 indicates second color is assigned.
    int colorArr[] = new int[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // This code is to handle disconnected graoh
    for (int i = 0; i < V; i++)
      if (colorArr[i] == -1)
        if (isBipartiteUtil(G, i, colorArr) == false)
           return false;

     return true;
}

/* Driver program to test above function */
public static void main(String[] args)
{
     int G[][] = {{0, 1, 0, 1},
                  {1, 0, 1, 0},
                  {0, 1, 0, 1},
                  {1, 0, 1, 0}};

            if (isBipartite(G))
               System.out.println("Yes");
            else
               System.out.println("No");
    }
}

// This code is contributed by Arnav Kr. Mandal.
```

Python

```
# Python3 program to find out whether a
# given graph is Bipartite or not

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = [[0 for column in range(V)]
                      for row in range(V)]

        self.colorArr = [-1 for i in range(self.V)]

    # This function returns true if graph G[V][V]
    # is Bipartite, else false
    def isBipartiteUtil(self, src):

        # Create a color array to store colors
        # assigned to all veritces. Vertex
        # number is used as index in this array.
        # The value '-1' of self.colorArr[i] is used
        # to indicate that no color is assigned to
        # vertex 'i'. The value 1 is used to indicate
        # first color is assigned and value 0
        # indicates second color is assigned.

# Assign first color to source

# Create a queue (FIFO) of vertex numbers and
# enqueue source vertex for BFS traversal
queue = []
queue.append(src)

# Run while there are vertices in queue
# (Similar to BFS)
while queue:

    u = queue.pop()

    # Return false if there is a self-loop
    if self.graph[u][u] == 1:
        return False;

    for v in range(self.V):

        # An edge from u to v exists and
        # destination v is not colored
        if (self.graph[u][v] == 1 and
            self.colorArr[v] == -1):

            # Assign alternate color to
            # this adjacent v of u
            self.colorArr[v] = 1 - self.colorArr[u]
            queue.append(v)
```

```
                elif (self.graph[u][v] == 1 and
                      self.colorArr[v] == self.colorArr[u]):
                    return False

        # If we reach here, then all adjacent
        # vertices can be colored with alternate
        # color
        return True

    def isBipartite(self):
        self.colorArr = [-1 for i in range(self.V)]
        for i in range(self.V):
            if self.colorArr[i] == -1:
                if not self.isBipartiteUtil(i):
                    return False
        return True

# Driver Code
g = Graph(4)
g.graph = [[0, 1, 0, 1],
           [1, 0, 1, 0],
           [0, 1, 0, 1],
           [1, 0, 1, 0]]

print "Yes" if g.isBipartite() else "No"

# This code is contributed by Anshuman Sharma
```

C#

```
// C# Code to check whether a given
// graph is Bipartite or not
using System;
using System.Collections.Generic;

class GFG
{
    public static int V = 4;

    // This function returns true if graph
    // G[V,V] is Bipartite, else false
    public static bool isBipartiteUtil(int [,]G,
                            int src, int []colorArr)
    {
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers and
        // enqueue source vertex for BFS traversal
        Queue<int> q = new Queue<int>();
        q.Enqueue(src);

        // Run while there are vertices in queue
        // (Similar to BFS)
        while (q.Count != 0)
        {
            // Dequeue a vertex from queue
            // ( Refer http://goo.gl/35oz8 )
            int u = q.Peek();
            q.Dequeue();

            // Return false if there is a self-loop
            if (G[u, u] == 1)
                return false;

            // Find all non-colored adjacent vertices
            for (int v = 0; v < V; ++v)
            {

                // An edge from u to v exists and
                // destination v is not colored
                if (G[u, v] == 1 && colorArr[v] == -1)
                {

                    // Assign alternate color to this
                    // adjacent v of u
                    colorArr[v] = 1 - colorArr[u];
                    q.Enqueue(v);
                }

                // An edge from u to v exists and
                // destination v is colored with same
                // color as u
                else if (G[u, v] == 1 &&
                         colorArr[v] == colorArr[u])
                    return false;
            }
        }

        // If we reach here, then all
        // adjacent vertices can be colored
        // with alternate color
        return true;
    }

    // Returns true if G[,] is Bipartite,
    // else false
    public static bool isBipartite(int [,]G)
    {
        // Create a color array to store
        // colors assigned to all veritces.
        // Vertex/ number is used as
        // index in this array. The value '-1'
        // of colorArr[i] is used to indicate
        // that no color is assigned to vertex 'i'.
        // The value 1 is used to indicate
        // first color is assigned and value
        // 0 indicates second color is assigned.
        int []colorArr = new int[V];
        for (int i = 0; i < V; ++i)
            colorArr[i] = -1;

        // This code is to handle disconnected graoh
        for (int i = 0; i < V; i++)
            if (colorArr[i] == -1)
                if (isBipartiteUtil(G, i,
                                    colorArr) == false)
                    return false;

        return true;
    }
`
```

```
    public static void Main(String[] args)
    {
        int [,]G = { { 0, 1, 0, 1 }, { 1, 0, 1, 0 },
                     { 0, 1, 0, 1 }, { 1, 0, 1, 0 } };

        if (isBipartite(G))
            Console.WriteLine("Yes");
        else
            Console.WriteLine("No");
    }
}

// This code is contributed by Rajput-Ji
```

Output:

```
Yes
```

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is O(V^2) where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes O(V+E).

Exercise:

1. Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

Solution :

C++

```
// C++ program to find out whether a given graph is Bipartite or not.
// Using recursion.
#include <iostream>

using namespace std;
#define V 4


bool colorGraph(int G[][V],int color[],int pos, int c){

    if(color[pos] != -1 && color[pos] !=c)
        return false;

    // color this pos as c and all its neighbours and 1-c
    color[pos] = c;
    bool ans = true;
    for(int i=0;i<V;i++){
        if(G[pos][i]){
            if(color[i] == -1)
                ans &= colorGraph(G,color,i,1-c);

            if(color[i] !=-1 && color[i] != 1-c)
                return false;
        }
        if (!ans)
            return false;
    }

    return true;
}

bool isBipartite(int G[][V]){
    int color[V];
    for(int i=0;i<V;i++)
        color[i] = -1;

    //start is vertex 0;
    int pos = 0;
    // two colors 1 and 0
    return colorGraph(G,color,pos,1);

}

int main()
{
    int G[][V] = {{0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0}
    };

    isBipartite(G) ? cout<< "Yes" : cout << "No";
    return 0;
}
// This code is contributed By Mudit Verma
```

Java

```
// Java program to find out whether
// a given graph is Bipartite or not.
// Using recursion.
class GFG
{
    static final int V = 4;

    static boolean colorGraph(int G[][],
                              int color[],
                              int pos, int c)
    {
        if (color[pos] != -1 &&
            color[pos] != c)
            return false;

        // color this pos as c and
        // all its neighbours as 1-c
        color[pos] = c;
        boolean ans = true;
        for (int i = 0; i < V; i++)
        {
            if (G[pos][i] == 1)
            {
                if (color[i] == -1)
                    ans &= colorGraph(G, color, i, 1 - c);

                if (color[i] != -1 && color[i] != 1 - c)
                    return false;
```

```
        }
        return true;
    }

    static boolean isBipartite(int G[][])
    {
        int[] color = new int[V];
        for (int i = 0; i < V; i++)
            color[i] = -1;

        // start is vertex 0;
        int pos = 0;

        // two colors 1 and 0
        return colorGraph(G, color, pos, 1);
    }

    // Driver Code
    public static void main(String[] args)
    {
        int G[][] = { { 0, 1, 0, 1 },
                      { 1, 0, 1, 0 },
                      { 0, 1, 0, 1 },
                      { 1, 0, 1, 0 } };

        if (isBipartite(G))
            System.out.print("Yes");
        else
            System.out.print("No");
    }
}

    // This code is contributed by Rajput-Ji
```

Python3

```
# Python3 program to find out whether a given
# graph is Bipartite or not using recursion.
V = 4

def colorGraph(G, color, pos, c):

    if color[pos] != -1 and color[pos] != c:
        return False

    # color this pos as c and all its neighbours and 1-c
    color[pos] = c
    ans = True
    for i in range(0, V):
        if G[pos][i]:
            if color[i] == -1:
                ans &= colorGraph(G, color, i, 1-c)

            if color[i] !=-1 and color[i] != 1-c:
                return False

        if not ans:
            return False

    return True

def isBipartite(G):

    color = [-1] * V

    #start is vertex 0
    pos = 0
    # two colors 1 and 0
    return colorGraph(G, color, pos, 1)

if __name__ == "__main__":

    G = [[0, 1, 0, 1],
         [1, 0, 1, 0],
         [0, 1, 0, 1],
         [1, 0, 1, 0]]

    if isBipartite(G): print("Yes")
    else: print("No")

# This code is contributed by Rituraj Jain
```

C#

```
// C# program to find out whether
// a given graph is Bipartite or not.
// Using recursion.
using System;

class GFG
{
    static readonly int V = 4;

    static bool colorGraph(int [,]G,
                           int []color,
                           int pos, int c)
    {
        if (color[pos] != -1 &&
            color[pos] != c)
            return false;

        // color this pos as c and
        // all its neighbours as 1-c
        color[pos] = c;
        bool ans = true;
        for (int i = 0; i < V; i++)
        {
            if (G[pos, i] == 1)
            {
                if (color[i] == -1)
                    ans &= colorGraph(G, color, i, 1 - c);

                if (color[i] != -1 && color[i] != 1 - c)
                    return false;
            }
            if (!ans)
                return false;
```

```
    }

    static bool isBipartite(int [,]G)
    {
        int[] color = new int[V];
        for (int i = 0; i < V; i++)
            color[i] = -1;

        // start is vertex 0;
        int pos = 0;

        // two colors 1 and 0
        return colorGraph(G, color, pos, 1);
    }

    // Driver Code
    public static void Main(String[] args)
    {
        int [,]G = {{ 0, 1, 0, 1 },
                    { 1, 0, 1, 0 },
                    { 0, 1, 0, 1 },
                    { 1, 0, 1, 0 }};

        if (isBipartite(G))
            Console.Write("Yes");
        else
            Console.Write("No");
    }
}

// This code is contributed by 29AjayKumar
```

References:

http://en.wikipedia.org/wiki/Graph_coloring

http://en.wikipedia.org/wiki/Bipartite_graph

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recommended Posts:

Check if a given graph is Bipartite using DFS

Maximum number of edges to be added to a tree so that it stays a Bipartite graph

Maximum number of edges in Bipartite graph

Maximum Bipartite Matching

Minimum Bipartite Groups

Check whether given degrees of vertices represent a Graph or Tree

Check if a given tree graph is linear or not

Check if a cycle of length 3 exists or not in a graph that satisfy a given condition

Check if a given Graph is 2-edge connected or not

Check if a given graph is tree or not

Check if a directed graph is connected or not

Check if incoming edges in a vertex of directed graph is equal to vertex itself or not

Find whether it is possible to finish all tasks or not from given dependencies

Determine whether a universal sink exists in a directed graph

Graph implementation using STL for competitive programming | Set 2 (Weighted graph)

Detect cycle in the graph using degrees of nodes of graph

Convert the undirected graph into directed graph such that there is no path of length greater than 1

Maximum number of edges that N-vertex graph can have such that graph is Triangle free | Mantel's Theorem

Convert undirected connected graph to strongly connected directed graph

Check if removing a given edge disconnects a graph

Improved By : Mudit Verma, anshumansharma, rituraj_jain, danielagfavero, Rajput-Ji, more

Article Tags : Graph BFS DFS Graph Coloring Samsung

Practice Tags : Samsung DFS Graph BFS

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments