**Bipartite Graph:**

A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. A bipartite graph is a special case of a k-partite graph with .

**Special Note:**

A bipartite graph cannot have a cycle of odd length.

A bipartite graph can be colored using only two color.

**Check If A Graph is bipartite or not:**

// 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;

}

**Hamiltonian Path, Hamiltonian Cycle, Hamiltonian Graph:**

A Hamiltonian path or traceable path is a path that visits each vertex of the graph exactly once.

Hamiltonian cycle is a cycle in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.

If A graph contains a hamiltonian cycle, that graph can be called as Hamiltonian path.

/\* C/C++ program for solution of Hamiltonian Cycle problem

using backtracking \*/

#include<stdio.h>

// Number of vertices in the graph

#define V 5

void printSolution(int path[]);

/\* A utility function to check if the vertex v can be added at

index 'pos' in the Hamiltonian Cycle constructed so far (stored

in 'path[]') \*/

bool isSafe(int v, bool graph[V][V], int path[], int pos)

{

/\* Check if this vertex is an adjacent vertex of the previously

added vertex. \*/

if (graph [ path[pos-1] ][ v ] == 0)

return false;

/\* Check if the vertex has already been included.

This step can be optimized by creating an array of size V \*/

for (int i = 0; i < pos; i++)

if (path[i] == v)

return false;

return true;

}

/\* A recursive utility function to solve hamiltonian cycle problem \*/

bool hamCycleUtil(bool graph[V][V], int path[], int pos)

{

/\* base case: If all vertices are included in Hamiltonian Cycle \*/

if (pos == V)

{

// And if there is an edge from the last included vertex to the

// first vertex

if ( graph[ path[pos-1] ][ path[0] ] == 1 )

return true;

else

return false;

}

// Try different vertices as a next candidate in Hamiltonian Cycle.

// We don't try for 0 as we included 0 as starting point in in hamCycle()

for (int v = 1; v < V; v++)

{

/\* Check if this vertex can be added to Hamiltonian Cycle \*/

if (isSafe(v, graph, path, pos))

{

path[pos] = v;

/\* recur to construct rest of the path \*/

if (hamCycleUtil (graph, path, pos+1) == true)

return true;

/\* If adding vertex v doesn't lead to a solution,

then remove it \*/

path[pos] = -1;

}

}

/\* If no vertex can be added to Hamiltonian Cycle constructed so far,

then return false \*/

return false;

}

/\* This function solves the Hamiltonian Cycle problem using Backtracking.

It mainly uses hamCycleUtil() to solve the problem. It returns false

if there is no Hamiltonian Cycle possible, otherwise return true and

prints the path. Please note that there may be more than one solutions,

this function prints one of the feasible solutions. \*/

bool hamCycle(bool graph[V][V])

{

int \*path = new int[V];

for (int i = 0; i < V; i++)

path[i] = -1;

/\* Let us put vertex 0 as the first vertex in the path. If there is

a Hamiltonian Cycle, then the path can be started from any point

of the cycle as the graph is undirected \*/

path[0] = 0;

if ( hamCycleUtil(graph, path, 1) == false )

{

printf("\nSolution does not exist");

return false;

}

printSolution(path);

return true;

}

/\* A utility function to print solution \*/

void printSolution(int path[])

{

printf ("Solution Exists:"

" Following is one Hamiltonian Cycle \n");

for (int i = 0; i < V; i++)

printf(" %d ", path[i]);

// Let us print the first vertex again to show the complete cycle

printf(" %d ", path[0]);

printf("\n");

}

// driver program to test above function

int main()

{

/\* Let us create the following graph

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4) \*/

bool graph1[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

};

// Print the solution

hamCycle(graph1);

/\* Let us create the following graph

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3) (4) \*/

bool graph2[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 0},

{0, 1, 1, 0, 0},

};

// Print the solution

hamCycle(graph2);

return 0;

}

**Eularian Path And Eularian Graph:**

In graph theory, an Eulerian trail (or Eulerian path) is a trail in a finite graph which visits every edge exactly once. Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian trail which starts and ends on the same vertex.

**What is the difference between Eularian Path and Eularian Graph:**

An Euler path is a path that crosses every edge exactly once without repeating, if it ends at the initial vertex then it is a Euler cycle. A Hamiltonian path passes through each vertex (note not each edge), exactly once, if it ends at the initial vertex then it is a Hamiltonian cycle

**Complete Graph:**

A complete graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with graph vertices is denoted and has (the triangular numbers) undirected edges, where is a binomial coefficient. In older literature, complete graphs are sometimes called universal graphs.

**What is the difference between Complete Graph and Connected Graph:**

A complete graph has edge between any two vertices. A connected graph has path between any two vertices.

**K Regular Graph:**

In graph theory, a regular graph is a graph where each vertex has the same number of neighbors; i.e. every vertex has the same degree or valency. A regular directed graph **must also satisfy the stronger condition that the indegree and outdegree of each vertex are equal to each other.**