**To Find A Cycle In A Graph:**

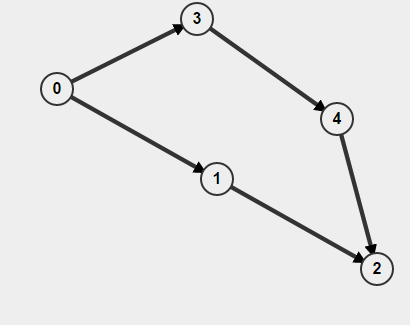
For finding a cycle in a **undirected** graph, either breadth first search or depth first search can be used.

However, for finding a cycle in **directed graph**, only depth first search can be used.

1. **Why can we not use breadth first search for finding cycle?**

Depth first search is more memory efficient than breadth first search as you can backtrack sooner. It is also easier to implement if you use the call stack but this relies on the longest path not overflowing the stack.

Also if your graph is directed then you have to not just remember if you have visited a node or not, but also how you got there. Otherwise you might think you have found a cycle but in reality all you have is two separate paths A->B but that doesn't mean there is a path B->A. For example,



If you do BFS starting from 0, it will detect as cycle is present but actually there is no cycle.

With a depth first search you can mark nodes as visited as you descend and unmark them as you backtrack.

**Finding Cycle In An Undirected Graph Using Recursive DFS:**

we can use DFS to detect cycle in an undirected graph in O(V+E) time. We do a DFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

#include<bits/stdc++.h>

using namespace std;

// Class for an undirected graph

class Graph

{

private:

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

bool is\_cyclic\_util(int v, bool visited[], int parent);

public:

Graph(int V); // Constructor

void add\_edge(int v, int w); // to add an edge to graph

bool is\_cyclic(); // returns true if there is a cycle

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::add\_edge(int v, int w)

{

adj[v].push\_back(w); // Add w to vâ€™s list.

adj[w].push\_back(v); // Add v to wâ€™s list.

}

// A recursive function that uses visited[] and parent to detect

// cycle in subgraph reachable from vertex v.

bool Graph::is\_cyclic\_util(int v, bool visited[], int parent)

{

// Mark the current node as visited

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator it;

for (it = adj[v].begin(); it != adj[v].end();it++)

{

// If an adjacent is not visited, then recur for that adjacent

if (!visited[\*it])

{

//if the adjacent is not visited, but visiting it would eventually lead to find a cycle

if (is\_cyclic\_util(\*it, visited, v))

{

return true;

}

}

// If an adjacent is visited and not parent of current vertex,

// then there is a cycle.

else if (\*it != parent)

{

return true;

}

}

return false;

}

// Returns true if the graph contains a cycle, else false.

bool Graph::is\_cyclic()

{

// Mark all the vertices as not visited and not part of recursion stack

bool \*visited = new bool[V];

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

{

visited[i] = false;

}

// Call the recursive helper function to detect cycle in different DFS trees

//it is necessary when graph is not strongly connected

//since, there is not any mention of the graph being strongly connected, we must do that

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

{

if (!visited[u]) // Don't recur for u if it is already visited

{

//initially for every disjoint n ary tree or for every disjoint component of a graph, the parent is set as -1

if (is\_cyclic\_util(u, visited, -1))

{

return true;

}

}

}

//if graph is strongly connected, we can skip that

return false;

}

// Driver program to test above functions

int main()

{

Graph g1(5);

g1.add\_edge(1, 0);

g1.add\_edge(0, 2);

g1.add\_edge(2, 0);

g1.add\_edge(0, 3);

g1.add\_edge(3, 4);

g1.is\_cyclic()? cout << "Graph contains cycle\n": cout << "Graph doesn't contain cycle\n";

Graph g2(3);

g2.add\_edge(0, 1);

g2.add\_edge(1, 2);

g2.is\_cyclic()? cout << "Graph contains cycle\n":cout << "Graph doesn't contain cycle\n";

return 0;

}

Now, you can convert the recursive solution into iterative DFS using a stack and having a parent array.

We can solve it using BFS, too.

Now, there is another approach of discovering a cycle in a undirected graph. That is using Union Find Algorithm.

**Finding Cycle In An Undirected Graph Using Union Find Algorithm:**

**(Note: This solution is taken from geeksforgeeks. Though, in geeksforgeeks it is mentioned as a solution for undirected graph, I do not find the graph to be undirected)**

**Basics About Union Find Algorithm:**

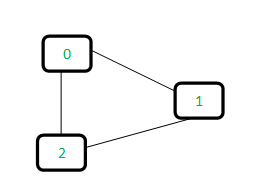
**Find:** Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

**Union:** Join two subsets into a single subset.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an algorithm to detect cycle. This is another method based on Union-Find. This method assumes that graph doesn’t contain any self-loops.

We can keep track of the subsets in a 1D array, let’s call it parent[].

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0 1 2

-1 -1 -1

Now process all edges one by one.

**Edge 0-1:** Find the subsets in which vertices 0 and 1 are. Since they are in different subsets (initially, all vertices’ parent are set as -1, so, initially any two vertices will be disjoint) , we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa. (Note: whatever order you did choose, you have to choose it every time)

0 1 2 <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

1 -1 -1

**Edge 1-2:** 1 is in subset 1 and 2 is in subset 2. So, take union.

0 1 2 <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})

1 2 -1

**Edge 0-2:**

0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

1. >1->2 // 1 is parent of 0 and 2 is parent of 1

**Code:**

// A union-find algorithm to detect cycle in a graph

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// a structure to represent an edge in the graph

struct Edge

{

int src, dest;

};

// a structure to represent a graph

struct Graph

{

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph =

(struct Graph\*) malloc( sizeof(struct Graph) );

graph->V = V;

graph->E = E;

graph->edge =

(struct Edge\*) malloc( graph->E \* sizeof( struct Edge ) );

return graph;

}

// A utility function to find the subset of an element i

int find(int parent[], int i)

{

if (parent[i] == -1)

return i;

return find(parent, parent[i]);

}

// A utility function to do union of two subsets

void Union(int parent[], int x, int y)

{

int xset = find(parent, x);

int yset = find(parent, y);

parent[xset] = yset;

}

// The main function to check whether a given graph contains

// cycle or not

int isCycle( struct Graph\* graph )

{

// Allocate memory for creating V subsets

int \*parent = (int\*) malloc( graph->V \* sizeof(int) );

// Initialize all subsets as single element sets

memset(parent, -1, sizeof(int) \* graph->V);

// Iterate through all edges of graph, find subset of both

// vertices of every edge, if both subsets are same, then

// there is cycle in graph.

for(int i = 0; i < graph->E; ++i)

{

int x = find(parent, graph->edge[i].src);

int y = find(parent, graph->edge[i].dest);

if (x == y)

return 1;

Union(parent, x, y);

}

return 0;

}

// Driver program to test above functions

int main()

{

/\* Let us create the following graph

0

| \

| \

1-----2 \*/

int V = 3, E = 3;

struct Graph\* graph = createGraph(V, E);

// add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

// add edge 1-2

graph->edge[1].src = 1;

graph->edge[1].dest = 2;

// add edge 0-2

graph->edge[2].src = 0;

graph->edge[2].dest = 2;

if (isCycle(graph))

printf( "graph contains cycle" );

else

printf( "graph doesn't contain cycle" );

return 0;

}

**Is the solution better than BFS or DFS Based Solution for cycle detection?**

The time complexity of the union-find algorithm is O(ELogV). Whereas, we could solve it using DFS in O(V+E) time.

Now, this solution is taken from geeksforgeeks. I will modify the solution later to make it work like the solution with DFS approach.

However, though, in Geeksforgeeks, it is given with the heading **Disjoint Set (Or Union-Find) | Set 1 (Detect Cycle in an Undirected Graph),** but I don’t think it is actually undirected graph.

# Detect Cycle in a Directed Graph

Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS.

Now, this is the solution taken form geeksforgeeks:

// A C++ Program to detect cycle in a graph

#include<iostream>

#include <list>

#include <limits.h>

using namespace std;

class Graph

{

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

bool isCyclicUtil(int v, bool visited[], bool \*rs); // used by isCyclic()

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // to add an edge to graph

bool isCyclic(); // returns true if there is a cycle in this graph

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)

{

if(visited[v] == false)

{

// Mark the current node as visited and part of recursion stack

visited[v] = true;

recStack[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )

return true;

else if (recStack[\*i])

return true;

}

}

//if the vertex is already visited or if the vertex is not visited but does not lead to a cycle

//that means if the current path does not lead to a cycle we will remove vertices one by one from recStack

recStack[v] = false; // remove the vertex from recursion stack

return false;

}

// Returns true if the graph contains a cycle, else false.

// This function is a variation of DFS() in

bool Graph::isCyclic()

{

// Mark all the vertices as not visited and not part of recursion

// stack

bool \*visited = new bool[V];

bool \*recStack = new bool[V];

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

{

visited[i] = false;

recStack[i] = false;

}

// Call the recursive helper function to detect cycle in different

// DFS trees

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

if (isCyclicUtil(i, visited, recStack))

return true;

return false;

}

int main()

{

// Create a graph given in the above diagram

Graph g(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);

if(g.isCyclic())

cout << "Graph contains cycle";

else

cout << "Graph doesn't contain cycle";

return 0;

}

recStack is different than visited. (If the vertex is already visited or if the vertex is not visited but does not lead to a cycle.) That means if the current path does not lead to a cycle we will remove vertices one by one from recStack.

**Can it be solved with the same DFS Approach used for Undirected Graph?**

#include<bits/stdc++.h>

using namespace std;

// Class for an Directed graph

class Graph

{

private:

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

bool is\_cyclic\_util(int v, bool visited[], int parent);

public:

Graph(int V); // Constructor

void add\_edge(int v, int w); // to add an edge to graph

bool is\_cyclic(); // returns true if there is a cycle

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::add\_edge(int v, int w)

{

adj[v].push\_back(w); // Add w to vâ€™s list.

//make it directed graph

}

// A recursive function that uses visited[] and parent to detect

// cycle in subgraph reachable from vertex v.

bool Graph::is\_cyclic\_util(int v, bool visited[], int parent)

{

// Mark the current node as visited

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator it;

for (it = adj[v].begin(); it != adj[v].end();it++)

{

// If an adjacent is not visited, then recur for that adjacent

if (!visited[\*it])

{

//if the adjacent is not visited, but visiting it would eventually lead to find a cycle

if (is\_cyclic\_util(\*it, visited, v))

{

return true;

}

}

// If an adjacent is visited and not parent of current vertex,

// then there is a cycle.

else if (\*it != parent)

{

return true;

}

}

return false;

}

// Returns true if the graph contains a cycle, else false.

bool Graph::is\_cyclic()

{

// Mark all the vertices as not visited and not part of recursion stack

bool \*visited = new bool[V];

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

{

visited[i] = false;

}

// Call the recursive helper function to detect cycle in different DFS trees

//it is necessary when graph is not strongly connected

//since, there is not any mention of the graph being strongly connected, we must do that

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

{

if (!visited[u]) // Don't recur for u if it is already visited

{

//initially for every disjoint n ary tree or for every disjoint component of a graph, the parent is set as -1

if (is\_cyclic\_util(u, visited, -1))

{

return true;

}

}

}

//if graph is strongly connected, we can skip that

return false;

}

// Driver program to test above functions

int main()

{

Graph g(4);

g.add\_edge(0, 1);

g.add\_edge(0, 2);

g.add\_edge(1, 2);

g.add\_edge(2, 0);

g.add\_edge(2, 3);

g.add\_edge(3, 3);

if(g.is\_cyclic())

cout << "Graph contains cycle";

else

cout << "Graph doesn't contain cycle";

return 0;

}

**It worked with the same graph in the previous example.**

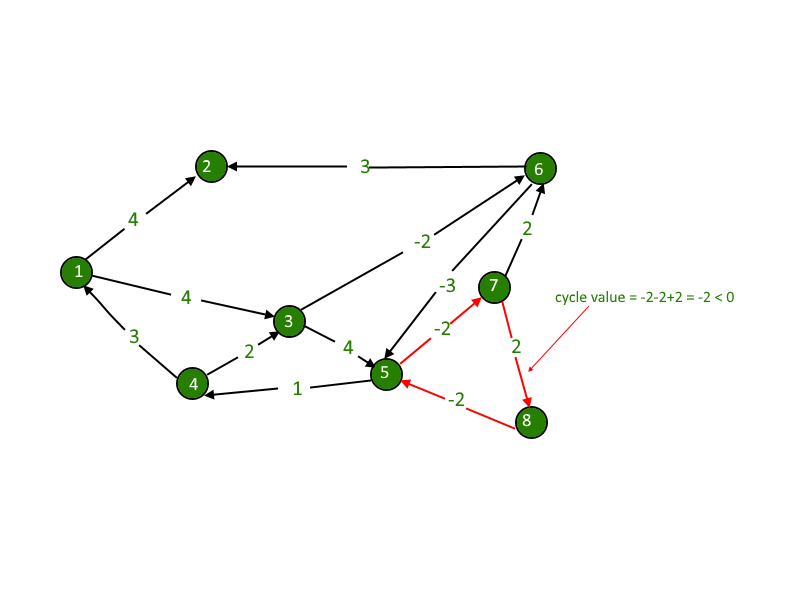
**Whether there was a performance flaw or not, I have to check. Or, whether this solution will generate wrong result in terminal cases or not, I have to check. My initial guess, it will not.**

**Can we follow the BFS approach of discovering cycle in a undirected graph?**

Absolutely not.

Why? It was already discussed as the first topic.

**Detecting Negative Edge Cycle Using Floyd Warshall:**

****

Now, you can see, there is a negative cycle.

Distance of any node from itself is always zero. But in some cases, as in this example, when we traverse further from 4 to 1, the distance comes out to be -2, i.e. distance of 1 from 1 will become -2. This is our catch, we just have to check the nodes distance from itself and if it comes out to be negative, we will detect the required negative cycle.

(This will work for both strongly connected graph as well as disconnected graph)

**Detecting Negative Edge Cycle Detection Using Bellman Ford:  
  
(Now, Floyd Warshall finds shortest distance among Every pair of vertices. Now, Bellman ford finds shortest distance from root/source node to any vertices)**

1) Initialize distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

…..a) Do following for each edge u-v

………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]

………………….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

**// A C++ program to check if a graph contains negative**

**// weight cycle using Bellman-Ford algorithm. This program**

**// works only if all vertices are reachable from a source**

**// vertex 0.**

**#include <bits/stdc++.h>**

**using namespace std;**

**// a structure to represent a weighted edge in graph**

**struct Edge {**

**int src, dest, weight;**

**};**

**// a structure to represent a connected, directed and**

**// weighted graph**

**struct Graph {**

**// V-> Number of vertices, E-> Number of edges**

**int V, E;**

**// graph is represented as an array of edges.**

**struct Edge\* edge;**

**};**

**// Creates a graph with V vertices and E edges**

**struct Graph\* createGraph(int V, int E)**

**{**

**struct Graph\* graph = new Graph;**

**graph->V = V;**

**graph->E = E;**

**graph->edge = new Edge[graph->E];**

**return graph;**

**}**

**// The main function that finds shortest distances**

**// from src to all other vertices using Bellman-**

**// Ford algorithm. The function also detects**

**// negative weight cycle**

**bool isNegCycleBellmanFord(struct Graph\* graph,**

**int src)**

**{**

**int V = graph->V;**

**int E = graph->E;**

**int dist[V];**

**// Step 1: Initialize distances from src**

**// to all other vertices as INFINITE**

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

**dist[i] = INT\_MAX;**

**dist[src] = 0;**

**// Step 2: Relax all edges |V| - 1 times.**

**// A simple shortest path from src to any**

**// other vertex can have at-most |V| - 1**

**// edges**

**for (int i = 1; i <= V - 1; i++) {**

**for (int j = 0; j < E; j++) {**

**int u = graph->edge[j].src;**

**int v = graph->edge[j].dest;**

**int weight = graph->edge[j].weight;**

**if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])**

**dist[v] = dist[u] + weight;**

**}**

**}**

**// Step 3: check for negative-weight cycles.**

**// The above step guarantees shortest distances**

**// if graph doesn't contain negative weight cycle.**

**// If we get a shorter path, then there**

**// is a cycle.**

**for (int i = 0; i < E; i++) {**

**int u = graph->edge[i].src;**

**int v = graph->edge[i].dest;**

**int weight = graph->edge[i].weight;**

**if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])**

**return true;**

**}**

**return false;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**/\* Let us create the graph given in above example \*/**

**int V = 5; // Number of vertices in graph**

**int E = 8; // Number of edges in graph**

**struct Graph\* graph = createGraph(V, E);**

**// add edge 0-1 (or A-B in above figure)**

**graph->edge[0].src = 0;**

**graph->edge[0].dest = 1;**

**graph->edge[0].weight = -1;**

**// add edge 0-2 (or A-C in above figure)**

**graph->edge[1].src = 0;**

**graph->edge[1].dest = 2;**

**graph->edge[1].weight = 4;**

**// add edge 1-2 (or B-C in above figure)**

**graph->edge[2].src = 1;**

**graph->edge[2].dest = 2;**

**graph->edge[2].weight = 3;**

**// add edge 1-3 (or B-D in above figure)**

**graph->edge[3].src = 1;**

**graph->edge[3].dest = 3;**

**graph->edge[3].weight = 2;**

**// add edge 1-4 (or A-E in above figure)**

**graph->edge[4].src = 1;**

**graph->edge[4].dest = 4;**

**graph->edge[4].weight = 2;**

**// add edge 3-2 (or D-C in above figure)**

**graph->edge[5].src = 3;**

**graph->edge[5].dest = 2;**

**graph->edge[5].weight = 5;**

**// add edge 3-1 (or D-B in above figure)**

**graph->edge[6].src = 3;**

**graph->edge[6].dest = 1;**

**graph->edge[6].weight = 1;**

**// add edge 4-3 (or E-D in above figure)**

**graph->edge[7].src = 4;**

**graph->edge[7].dest = 3;**

**graph->edge[7].weight = -3;**

**if (isNegCycleBellmanFord(graph, 0))**

**cout << "Yes";**

**else**

**cout << "No";**

**return 0;**

**}**

**How to handle disconnected graph (If cycle is not reachable from source)?**

The above algorithm and program might not work if the given graph is disconnected. It works when all vertices are reachable from source vertex 0.

To handle disconnected graph, we can repeat the process for vertices for which distance is infinite.

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

using namespace std;

// a structure to represent a weighted edge in graph

struct Edge {

int src, dest, weight;

};

// a structure to represent a connected, directed and

// weighted graph

struct Graph {

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[graph->E];

return graph;

}

// The main function that finds shortest distances

// from src to all other vertices using Bellman-

// Ford algorithm. The function also detects

// negative weight cycle

bool isNegCycleBellmanFord(struct Graph\* graph,

int src, int dist[])

{

int V = graph->V;

int E = graph->E;

// Step 1: Initialize distances from src

// to all other vertices as INFINITE

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

dist[i] = INT\_MAX;

dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times.

// A simple shortest path from src to any

// other vertex can have at-most |V| - 1

// edges

for (int i = 1; i <= V - 1; i++) {

for (int j = 0; j < E; j++) {

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

// Step 3: check for negative-weight cycles.

// The above step guarantees shortest distances

// if graph doesn't contain negative weight cycle.

// If we get a shorter path, then there

// is a cycle.

for (int i = 0; i < E; i++) {

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

return true;

}

return false;

}

// Returns true if given graph has negative weight

// cycle.

bool isNegCycleDisconnected(struct Graph\* graph)

{

int V = graph->V;

// To keep track of visited vertices to avoid

// recomputations.

bool visited[V];

memset(visited, 0, sizeof(visited));

// This array is filled by Bellman-Ford

int dist[V];

// Call Bellman-Ford for all those vertices

// that are not visited

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

if (visited[i] == false) {

// If cycle found

if (isNegCycleBellmanFord(graph, i, dist))

return true;

// Mark all vertices that are visited

// in above call.

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

if (dist[i] != INT\_MAX)

visited[i] = true;

}

}

return false;

}

// Driver program to test above functions

int main()

{

/\* Let us create the graph given in above example \*/

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

if (isNegCycleDisconnected(graph))

cout << "Yes";

else

cout << "No";

return 0;

}

**Checks If Graph Contains Odd Length Cycle Or Not:**The idea is based on an important fact that a graph does not contain a cycle of odd length if and only if it is Bipartite, i.e., it can be colored with two colors.

It is obvious that if a graph has odd length cycle then it cannot be Bipartite. In Bipartite graph there are two sets of vertices such that no vertex in a set is connected with any other vertex of same set). For a cycle of odd length, two vertices must of same set must be connected which contradicts Bipartite definition.

**// 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] contains**

**// odd cycle, else false**

**bool containsOdd(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**

**int u = q.front();**

**q.pop();**

**// Return true if there is a self-loop**

**if (G[u][u] == 1)**

**return true;**

**// 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 true;**

**}**

**}**

**// If we reach here, then all adjacent**

**// vertices can be colored with alternate**

**// color**

**return false;**

**}**

**// 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}**

**};**

**containsOdd(G, 0) ? cout << "Yes" : cout << "No";**

**return 0;**

**}**