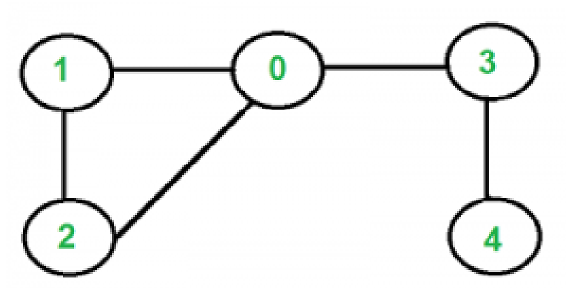
Detect cycle in an undirected graph

.

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.



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

We have discussed cycle detection for directed graph. We have also discussed a union-find algorithm for cycle detection in undirected graphs. The time complexity of the union-find algorithm is O(ELogV). Like directed graphs, 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.

C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;    // No. of vertices
```

```
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
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
};

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.
    adj[w].push_back(v); // Add v to w’s list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(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 i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
           if (isCyclicUtil(*i, visited, v))
              return true;
        }

        // If an adjacent is visited and not parent of current vertex,
        // then there is a cycle.
        else if (*i != parent)
           return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // 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
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if it is already visited
          if (isCyclicUtil(u, visited, -1))
             return true;

    return false;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
```

```
    g1.addEdge(3, 4);
    g1.isCyclic()? cout << "Graph contains cycle\n":
                   cout << "Graph doesn't contain cycle\n";

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.isCyclic()? cout << "Graph contains cycle\n":
                   cout << "Graph doesn't contain cycle\n";

    return 0;
}
```

Run on IDE Copy Code

Java

```
// A Java Program to detect cycle in an undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List Represntation

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for(int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

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

    // A recursive function that uses visited[] and parent to detect
    // cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

            // If an adjacent is not visited, then recur for that
            // adjacent
            if (!visited[i])
            {
                if (isCyclicUtil(i, visited, v))
                    return true;
            }

            // If an adjacent is visited and not parent of current
            // vertex, then there is a cycle.
            else if (i != parent)
                return true;
        }
        return false;
    }
```

```
    // Returns true if the graph contains a cycle, else false.
    Boolean isCyclic()
    {
        // Mark all the vertices as not visited and not part of
        // recursion stack
        Boolean visited[] = new Boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper function to detect cycle in
        // different DFS trees
        for (int u = 0; u < V; u++)
            if (!visited[u]) // Don't recur for u if already visited
                if (isCyclicUtil(u, visited, -1))
                    return true;

        return false;
    }


    // Driver method to test above methods
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 0);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        if (g1.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");

        Graph g2 = new Graph(3);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        if (g2.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");
    }
}
// This code is contributed by Aakash Hasija
```

Run on IDE Copy Code

Python

```
# Python Program to detect cycle in an undirected graph

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph


    # function to add an edge to graph
    def addEdge(self,v,w):
        self.graph[v].append(w) #Add w to v_s list
        self.graph[w].append(v) #Add v to w_s list

    # A recursive function that uses visited[] and parent to detect
    # cycle in subgraph reachable from vertex v.
    def isCyclicUtil(self,v,visited,parent):
```

```
        #Mark the current node as visited
        visited[v]= True

        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            # If the node is not visited then recurse on it
            if  visited[i]==False :
                if(self.isCyclicUtil(i,visited,v)):
                    return True
            # If an adjacent vertex is visited and not parent of current vertex,
            # then there is a cycle
            elif  parent!=i:
                return True

        return False


    #Returns true if the graph contains a cycle, else false.
    def isCyclic(self):
        # Mark all the vertices as not visited
        visited =[False]*(self.V)
        # Call the recursive helper function to detect cycle in different
        #DFS trees
        for i in range(self.V):
            if visited[i] ==False: #Don't recur for u if it is already visited
                if(self.isCyclicUtil(i,visited,-1))== True:
                    return True

        return False

# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "
g1 = Graph(3)
g1.addEdge(0,1)
g1.addEdge(1,2)


if g1.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Run on IDE Copy Code

Output:

```
Graph contains cycle
Graph doesn't contain cycle
```

Time Complexity: The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is O(V+E)

Detect cycle in an undirected graph | GeeksforGeeks

Exercise: Can we use BFS to detect cycle in an undirected graph in O(V+E) time? What about directed graphs?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recommended Posts:

- Detect cycle in an undirected graph using BFS
- Detect Cycle in a directed graph using colors
- Assign directions to edges so that the directed graph remains acyclic
- Applications of Breadth First Traversal
- Longest Path in a Directed Acyclic Graph
- Topological Sorting
- Check whether a given graph is Bipartite or not
- Dijkstra's shortest path algorithm | Greedy Algo-7
- Prim's Minimum Spanning Tree (MST) | Greedy Algo-5
- Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2
- Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)
- Disjoint Set (Or Union-Find) | Set 1 (Detect Cycle in an Undirected Graph)
- Detect Cycle in a Directed Graph
- Breadth First Search or BFS for a Graph
- Depth First Search or DFS for a Graph

Article Tags : Graph Adobe Amazon BFS DFS Flipkart graph-cycle MakeMyTrip Oracle union-find

Practice Tags : Amazon Flipkart Oracle Adobe MakeMyTrip DFS Graph BFS union-find

4

2.9

To-do Done

Based on **126** vote(s)

Feedback Add Notes Improve Article

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 Share this post!

A computer science portal for geeks

710-B, Advant Navis Business Park,
Sector-142, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org