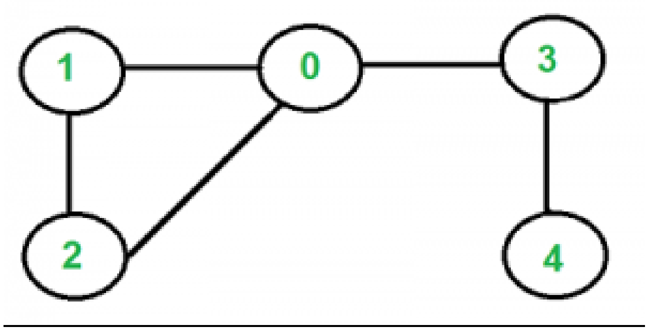
Detect cycle in an undirected graph

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.



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.

```
// 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

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)