**Why cycle detection algorithm differs in directed and undirected graphs ?**

<https://codeforces.com/blog/entry/91377>

Because having a cycle in a directed and an undirected graph isn't the same thing.

Suppose you have a 2-vertex undirected graph represented with the following adjacency list:

neighbors[0] = {1}

neighbors[1] = {0}

This graph doesn't have a cycle. It's just an edge.

Now interpret the same adjacency list as a directed graph. Now there is a cycle 0→1→0

**1. Detect Cycle in a Directed Graph using BFS [Topological Sort]**

/\*\*

Refer to

https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/

The idea is to simply use Kahn’s algorithm for Topological Sorting

Steps involved in detecting cycle in a directed graph using BFS.

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the

        graph and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

        Increment count of visited nodes by 1.

        Decrease in-degree by 1 for all its neighboring nodes.

        If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If count of visited nodes is not equal to the number of nodes in the graph has cycle, otherwise not.

How to find in-degree of each node?

There are 2 ways to calculate in-degree of every vertex:

Take an in-degree array which will keep track of

1) Traverse the array of edges and simply increase the counter of the destination node by 1.

    for each node in Nodes

        indegree[node] = 0;

    for each edge(src,dest) in Edges

        indegree[dest]++

  Time Complexity: O(V+E)

2) Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

    for each node in Nodes

        If (list[node].size()!=0) then

        for each dest in list

            indegree[dest]++;

  Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be

  executed E number of times, Thus overall time complexity is O(V+E).

  The overall time complexity of the algorithm is O(V+E)

\*/

  // Java program to check if there is a cycle in directed graph using BFS.

  class GFG {

      // Class to represent a graph

      static class Graph {

          int V; // No. of vertices'

          // Pointer to an array containing adjacency list

          Vector < Integer > [] adj;

          @SuppressWarnings("unchecked")

          Graph(int V) {

              // Constructor

              this.V = V;

              this.adj = new Vector[V];

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

                  adj[i] = new Vector < > ();

          }

          // function to add an edge to graph

          void addEdge(int u, int v) {

              adj[u].add(v);

          }

          // Returns true if there is a cycle in the graph

          // else false.

          // This function returns true if there is a cycle

          // in directed graph, else returns false.

          boolean isCycle() {

              // Create a vector to store indegrees of all

              // vertices. Initialize all indegrees as 0.

              int[] in\_degree = new int[this.V];

              Arrays.fill(in\_degree, 0);

              // Traverse adjacency lists to fill indegrees of

              // vertices. This step takes O(V+E) time

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

                  for (int v: adj[u])

                      in\_degree[v]++;

              }

              // Create an queue and enqueue all vertices with

              // indegree 0

              Queue < Integer > q = new LinkedList < Integer > ();

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

                  if (in\_degree[i] == 0)

                      q.add(i);

              // Initialize count of visited vertices

              int cnt = 0;

              // Create a vector to store result (A topological

              // ordering of the vertices)

              Vector < Integer > top\_order = new Vector < > ();

              // One by one dequeue vertices from queue and enqueue

              // adjacents if indegree of adjacent becomes 0

              while (!q.isEmpty()) {

                  // Extract front of queue (or perform dequeue)

                  // and add it to topological order

                  int u = q.poll();

                  top\_order.add(u);

                  // Iterate through all its neighbouring nodes

                  // of dequeued node u and decrease their in-degree

                  // by 1

                  for (int itr: adj[u])

                      if (--in\_degree[itr] == 0)

                          q.add(itr);

                  cnt++;

              }

              // Check if there was a cycle

              if (cnt != this.V)

                  return true;

              else

                  return false;

          }

      }

      // Driver Code

      public static void main(String[] args) {

          // Create a graph given in the above diagram

          Graph g = new Graph(6);

          g.addEdge(0, 1);

          g.addEdge(1, 2);

          g.addEdge(2, 0);

          g.addEdge(3, 4);

          g.addEdge(4, 5);

          if (g.isCycle())

              System.out.println("Yes");

          else

              System.out.println("No");

      }

  }

// Actually BFS implement on this way is Toplogical Sort, examples on below problems:

// CourseSchedule.java

**2. Detect cycle in an undirected graph using BFS**

<https://www.geeksforgeeks.org/detect-cycle-in-an-undirected-graph-using-bfs/>

// Java program to detect cycle in an undirected graph using BFS.

/\*\*

\* In this article, BFS based solution is discussed. We do a BFS 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.

\*/

class Cycle {

    public static void main(String arg[]) {

        int V = 4;

        ArrayList < Integer > adj[] = new ArrayList[V];

        for (int i = 0; i < 4; i++)

            adj[i] = new ArrayList < Integer > ();

        addEdge(adj, 0, 1);

        addEdge(adj, 1, 2);

        addEdge(adj, 2, 0);

        addEdge(adj, 2, 3);

        if (isCyclicDisconntected(adj, V))

            System.out.println("Yes");

        else

            System.out.println("No");

    }

    static void addEdge(ArrayList < Integer > adj[], int u, int v) {

        adj[u].add(v);

        adj[v].add(u);

    }

    static boolean isCyclicConntected(ArrayList < Integer > adj[], int s, int V, boolean visited[]) {

        // Set parent vertex for every vertex as -1.

        int parent[] = new int[V];

        Arrays.fill(parent, -1);

        // Create a queue for BFS

        Queue < Integer > q = new LinkedList < > ();

        // Mark the current node as

        // visited and enqueue it

        visited[s] = true;

        q.add(s);

        while (!q.isEmpty()) {

            // Dequeue a vertex from

            // queue and print it

            int u = q.poll();

            // Get all adjacent vertices

            // of the dequeued vertex u.

            // If an adjacent has not been

            // visited, then mark it visited

            // and enqueue it. We also mark parent

            // so that parent is not considered

            // for cycle.

            for (int i = 0; i < adj[u].size(); i++) {

                int v = adj[u].get(i);

                if (!visited[v]) {

                    visited[v] = true;

                    q.add(v);

                    parent[v] = u;

                } else if (parent[u] != v)

                    return true;

            }

        }

        return false;

    }

    static boolean isCyclicDisconntected(ArrayList < Integer > adj[], int V) {

        // Mark all the vertices as not visited

        boolean visited[] = new boolean[V];

        Arrays.fill(visited, false);

        for (int i = 0; i < V; i++)

            if (!visited[i] && isCyclicConntected(adj, i, V, visited))

                return true;

        return false;

    }

}

Time Complexity:

The program does a simple BFS Traversal of graph and graph is represented using adjacency list.

So the time complexity is O(V+E)

**3. Detect cycle in an undirected graph using DFS**

<https://www.geeksforgeeks.org/detect-cycle-undirected-graph/>

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.

Similar as detect cycle in undirected graph using BFS by setup a parent value (initialize as -1) which enables trace back on the graph

// 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, 1);

        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");

    }

}

**4. Detect Cycle in a Directed Graph using DFS [Backtracking]**

<https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>

class Graph {

    private final int V;

    private final List < List < Integer >> adj;

    public Graph(int V) {

        this.V = V;

        adj = new ArrayList < > (V);

        for (int i = 0; i < V; i++)

            adj.add(new LinkedList < > ());

    }

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

    // https://www.geeksforgeeks.org/archives/18212

    private boolean isCyclicUtil(int i, boolean[] visited, boolean[] recStack) {

        // Mark the current node as visited and

        // part of recursion stack

        if (recStack[i])

            return true;

        if (visited[i])

            return false;

        visited[i] = true;

        recStack[i] = true;

        List < Integer > children = adj.get(i);

        for (Integer c: children)

            if (isCyclicUtil(c, visited, recStack))

                return true;

        recStack[i] = false;

        return false;

    }

    private void addEdge(int source, int dest) {

        adj.get(source).add(dest);

    }

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

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

    // https://www.geeksforgeeks.org/archives/18212

    private boolean isCyclic() {

        // Mark all the vertices as not visited and

        // not part of recursion stack

        boolean[] visited = new boolean[V];

        boolean[] recStack = new boolean[V];

        // 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;

    }

    // Driver code

    public static void main(String[] args) {

        Graph graph = new Graph(4);

        graph.addEdge(0, 1);

        graph.addEdge(0, 2);

        graph.addEdge(1, 2);

        graph.addEdge(2, 0);

        graph.addEdge(2, 3);

        graph.addEdge(3, 3);

        if (graph.isCyclic())

            System.out.println("Graph contains cycle");

        else

            System.out.println("Graph doesn't " + "contain cycle");

    }

}

// The DFS implement examples on below problems: CourseSchedule.java

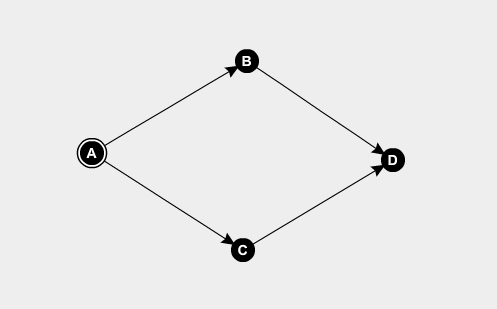
**Why should we use a recursion stack even if already have a visited boolean array when we determine cycle in a directed graph ?**

**Refer to**

<https://www.quora.com/Why-should-we-use-a-recursion-stack-in-spite-of-a-visited-boolean-array-when-we-determine-cycle-in-a-graph-directed-undirected>

**Using Boolean array for determining cycle, works only with undirected graph. This is because if we visit a node twice in an undirected graph, we can be sure that the graph is cyclic.**

**But a directed graph, requires use of stack. Take for example:**



Step 1.visited array initially: 0 0 0 0

Step 2.visit A: 1 0 0 0

Step 3.visit B: 1 1 0 0

Step 4.visit D: 1 1 0 1

**Step 5.Traceback D -> B -> A**

Step 6.visit C: 1 1 1 1

**Step 7.visit D: 1 1 1 [1] -> D visited twice**

**Now D has been visited before so the algorithm will declare the graph to be cylic, but the graph clearly is not cyclic.**

**Explain:**

**If in Step 5 we only use a visited boolean array to record the each node status (visited or not), then when go into Step 7 we will visit D again, if following same logic for undirected graph "if we visit a node twice in an undirected graph, we can be sure that the graph is cyclic.", which result as declare a cycle exists.**

**But actually not, how to fix? Introduce another boolean array besides visited boolean array, name it recursion stack to record the actual node status during recursion with backtracking technology.**

**So in Step 5 we will do backtrack on route D -> B -> A as below to clean up node status such as remove it out of recursion stack:**

**Initial status of recursion stack (top -> bottom): D -> B -> A**

**Step 1: Remove D from recursion stack, stack status: B -> A**

**Step 2: Remove B from recursion stack, stack status: A**

**Step 3: Remove A from recursion stack, stack status: empty**

**Now after backtrack, we will start as a fresh route from A -> C -> D, which is 0 0 0 0 -> 1 0 0 0 -> 1 0 1 0 -> 1 0 1 1, this time it is correct, we didn't visit D twice now, no cycle detect**

**5. Why DFS and not BFS for finding cycle in graphs**

<https://stackoverflow.com/questions/2869647/why-dfs-and-not-bfs-for-finding-cycle-in-graphs>