**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

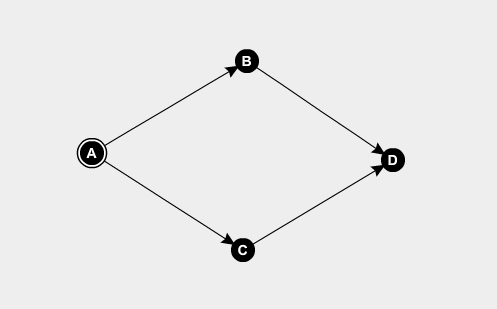
**Why should we use a recursion stack even if already have a visited boolean array when we determine cycle in a directed graph ?**

**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**

**Refer to Deepseek**

**R1 logic:**

Okay, let's see. The user is asking why we need a recursion stack in addition to a visited array when detecting cycles in a directed graph. The previous answer tried to explain it, but the user found another explanation and wants a better one.

Hmm, the example given in the Quora link is about a directed graph where using just a visited array would falsely detect a cycle. Let me parse that example again. The graph is A -> B -> D, and A -> C -> D. So there's no cycle here. If you traverse A->B->D, then backtrack and go A->C->D, D is already marked as visited. If the algorithm checks only the visited array, when it reaches D again from C, it would think there's a cycle because D was visited before. But in reality, D is part of a different path, not a cycle. So the problem is that the visited array alone can't distinguish between nodes visited in the current path versus other paths.

So the key point here is that in directed graphs, a node can be part of multiple paths, and a cycle exists only if a node is revisited in the same path. The recursion stack keeps track of the current traversal path. When backtracking, nodes are removed from the recursion stack, indicating they're no longer part of the current path. This way, when D is visited again via a different path (A->C->D), the recursion stack for the current path (A->C->D) doesn't have D until it's added again. Since D was already in the visited array from the previous path (A->B->D), but not in the current recursion stack, it's allowed. Thus, the recursion stack helps differentiate between revisiting a node in the same path (cycle) versus a different path (no cycle).

In contrast, undirected graphs don't have this issue because each edge is bidirectional. If you visit a node twice, there's definitely a cycle since you can go back along the same edge. But in directed graphs, edges are one-way, so a node can be reachable via multiple distinct paths without forming a cycle. Hence, the recursion stack is necessary to track the current path and avoid false cycle detection.

So the user's example illustrates that without the recursion stack, the visited array would incorrectly flag a cycle when a node is visited again via a different path. By using the recursion stack (which is dynamically updated during the DFS traversal and backtracking), we ensure that only the current path is checked for cycles. When backtracking, nodes are removed from the recursion stack, so subsequent visits to the same node via different paths don't trigger a cycle detection unless they're part of the same recursive path.

Therefore, the recursion stack is essential for directed graphs to track the current traversal path, while the visited array prevents redundant processing of nodes across different paths. Combining both ensures accurate cycle detection without false positives.

You're absolutely correct. The key distinction lies in how directed graphs require tracking the current traversal path to avoid false cycle detection, which isn't necessary for undirected graphs. Let’s break this down with clarity:

**Why the Visited Array Alone Fails for Directed Graphs**

In undirected graphs, a node being visited twice always implies a cycle (since edges are bidirectional). However, in directed graphs, a node can be visited multiple times through different paths without forming a cycle. The visited array alone cannot distinguish between these two scenarios:

**Example:**

Consider the directed graph:

A → B → D

↓ ↓

C → D

Path 1: A → B → D

Path 2: A → C → D

If we use only a visited array:

After exploring Path 1, D is marked as visited.

When exploring Path 2, D is already visited.

A naive algorithm would falsely conclude a cycle exists because D was visited twice.

This is incorrect, as the graph has no cycle (all edges flow forward).

**The Role of the Recursion Stack**

The recursion stack (recStack[]) tracks the current DFS traversal path dynamically. When backtracking, nodes are removed from the stack, ensuring that only nodes in the current path are checked for cycles.

**How It Fixes the Problem:**

During Path 1 (A → B → D):

recStack holds [A, B, D].

When backtracking, D, B, and A are removed from recStack.

During Path 2 (A → C → D):

recStack holds [A, C, D].

Even though D is marked as visited, it’s not in the current recStack, so no cycle is detected.

This ensures that visiting D again through a different path does not trigger a false cycle.

**Key Differences: Undirected vs. Directed Graphs**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Undirected Graph** | **Directed Graph** |
| Node visited twice | Cycle (always) | Not necessarily a cycle |
| Edge direction | Bidirectional | Unidirectional |
| Required Tracking | visited[] | visited[]  +  recStack[] |

**Why Undirected Graphs Don’t Need recStack[]**

In undirected graphs:

If you visit a node u from v, you must also traverse back from u to v (due to bidirectionality).

A visited node encountered again must be a parent node (trivial) or a cycle.

Thus, visited[] suffices.

**Summary**

Visited Array (visited[]):

Recursion Stack (recStack[]):

By combining both, we ensure correct cycle detection in directed graphs. The recursion stack is critical because it enforces the rule:

A cycle exists only if a node is revisited \*within the same traversal path\*.

**Based on above theory, please evaluate below example solution, if it clearly explain the situation ?**

Okay, let me try to understand the problem here. The user is asking to evaluate whether the provided Java code correctly explains the situation of detecting cycles in a directed graph using both a visited array and a recursion stack. Based on the previous conversation, the key point is that in directed graphs, you need the recursion stack to track the current path, unlike undirected graphs where a visited array alone suffices.

First, I'll look at the code structure. The Graph class has an adjacency list, and the isCyclic method uses two boolean arrays: visited and recStack. The isCyclicUtil method is a recursive helper that performs DFS.

Let me walk through the code step by step. When isCyclic is called, it initializes both arrays. Then, for each vertex, it calls isCyclicUtil if the vertex hasn't been visited yet.

In isCyclicUtil, the first check is if the current node is already in the recursion stack (recStack[i]). If yes, it returns true, indicating a cycle. Then it checks if the node is already visited; if so, returns false. Then it marks the node as visited and adds it to recStack. It then recursively checks all children. After processing children, it removes the node from recStack (by setting it to false) before backtracking.

Wait, in the example from the user's previous question, where there's a graph with A->B->D and A->C->D, the code should not detect a cycle. Let me see how this code would handle that. Suppose the nodes are 0 (A), 1 (B), 2 (C), 3 (D). The edges are 0→1, 0→2, 1→2, 2→0, 2→3, 3→3. Wait, in this specific code's main method, the graph has an edge from 2 to 0, which creates a cycle (0→1→2→0). Also, node 3 has a self-loop. So in this case, the code should correctly detect a cycle. But the user's example earlier was about a graph without a cycle, where a naive visited array would falsely detect a cycle. So maybe the code's example isn't the best, but the code itself is correct.

Wait, in the code's main method, the graph does have cycles. The edge 2→0 creates a cycle between 0,1,2. Also, node 3 has a self-loop. So when the code runs, it should print that a cycle exists, which is correct. But the user's previous example was about a graph without a cycle where using only visited would fail. Let's check if the code would handle that correctly.

Take the example where the graph is A→B→D, A→C→D. So nodes 0 (A), 1 (B), 2 (C), 3 (D). Edges: 0→1, 1→3, 0→2, 2→3. This graph is a DAG with no cycles. Let's simulate how the code would process this.

Starting with node 0 (A). visited is all false. recStack is all false. isCyclicUtil(0, visited, recStack) is called. recStack[0] is false, visited[0] is false. Mark visited[0] as true, recStack[0] as true. Then check children 1 and 2.

First child 1: isCyclicUtil(1, ...). recStack[1] is false, visited[1] is false. Mark visited[1], recStack[1]. Its child is 3. Call isCyclicUtil(3). Mark visited[3], recStack[3]. 3 has no children. Return false. Backtrack, recStack[3] is set to false. Then return from 1's loop, recStack[1] is false. Back to node 0, process next child 2. recStack[2] is false, visited[2] is false. Mark it. Its child is 3. Now, visited[3] is already true. So the check in isCyclicUtil for 3: since recStack[3] is false (because after backtracking from the first path, recStack[3] was reset), and visited[3] is true, so returns false. So no cycle detected. Thus, the code correctly doesn't detect a cycle here. So the code works in that case.

But why? Because when processing node 3 through the second path (A→C→D), even though visited[3] is true, since recStack[3] was already reset when backtracking from the first path (A→B→D), the code doesn't mistake it for a cycle. The recStack ensures that only nodes in the current path are considered. So the code correctly uses both arrays.

Another test case: the user's initial example where A→B→D and A→C→D, no cycle. The code would process node 0, mark it as visited and in recStack. Process child 1 (B), mark visited and recStack. Then child 3 (D), mark visited and recStack. D has no children, backtrack, recStack[3] reset. Then backtrack from B, recStack[1] reset. Back to A, process next child 2 (C). Mark visited and recStack. Child is 3 (D), which is visited but recStack[3] is false. So returns false. So no cycle detected. Correct.

But in the code provided, the example graph does have cycles, so the code correctly detects them. The code's logic seems solid. The key part is that after backtracking, recStack is reset, allowing nodes to be part of different paths without causing false positives.

Potential issues: The code's recStack is a boolean array, not an actual stack structure, but it works because during recursion, the order of setting and unsetting recStack[i] mimics the stack behavior. Each recursive call sets recStack[i] to true before processing children and sets it back to false after. This correctly tracks the current path.

Another thing to check: what if a node is part of multiple cycles? For example, two separate cycles. The code should still detect them as long as the DFS explores those paths.

So, in conclusion, the code correctly implements cycle detection for directed graphs by using both the visited array to avoid reprocessing nodes and the recStack to track the current path. The example in the code's main method does contain cycles, and the code would correctly identify them. The logic aligns with the theory discussed earlier.

The provided code correctly implements cycle detection in a directed graph using both a visited array and a recStack (recursion stack). Here's the breakdown:

**Key Strengths of the Code**

Correct Use of visited and recStack:

visited[]: Tracks nodes processed in \*any\* path to avoid redundant work.

recStack[]: Tracks nodes in the \*current DFS traversal path\* to detect back edges (cycles).

Backtracking with recStack:

When backtracking, recStack[i] is reset to false, ensuring nodes from previous paths don’t falsely trigger cycles in subsequent paths.

Example: In the graph A → B → D and A → C → D, after backtracking from A → B → D, recStack for B and D is cleared. When exploring A → C → D, D is marked as visited but not in the current recStack, so no cycle is detected.

Handling Self-Loops and Complex Cycles:

The code detects self-loops (e.g., 3 → 3) and cycles like 0 → 1 → 2 → 0 in the example graph.

**Example Walkthrough (Code’s Main Method)**

The graph in main has edges:

0 → 1, 0 → 2

1 → 2

2 → 0 (cycle!), 2 → 3

3 → 3 (self-loop)

Cycle 1: 0 → 1 → 2 → 0 (detected when DFS reaches 2 → 0, where 0 is in recStack).

Cycle 2: 3 → 3 (self-loop).

The code correctly prints "Graph contains cycle".

**Why It Works for Directed Graphs**

Scenario 1: Node Revisited in a Different Path

D is marked visited after the first path, but recStack[D] is reset.

In the second path, D is visited but not in recStack, so no false cycle is detected.

Scenario 2: Node Revisited in the Same Path

When 2 → 0 is processed, 0 is already in recStack, triggering a cycle.

**Code Limitations**

Efficiency: The recStack array is a valid approach, but an explicit stack data structure could improve clarity.

Thread Safety: Not thread-safe due to shared visited and recStack arrays (but acceptable for single-threaded use).

**Conclusion**

The code correctly explains and implements cycle detection in directed graphs by:

Using visited[] to skip already processed nodes.

Using recStack[] to track the current DFS path and detect back edges.

It avoids false positives (e.g., revisiting nodes in different paths) and correctly identifies true cycles (e.g., loops in the same path or self-loops). The example in main demonstrates this effectively.

**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>