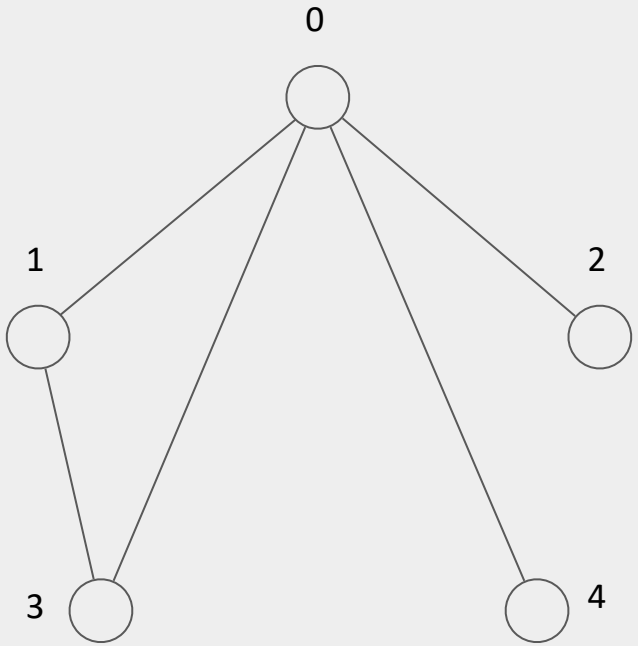
Graphs


0
1
2
3
4

Definition

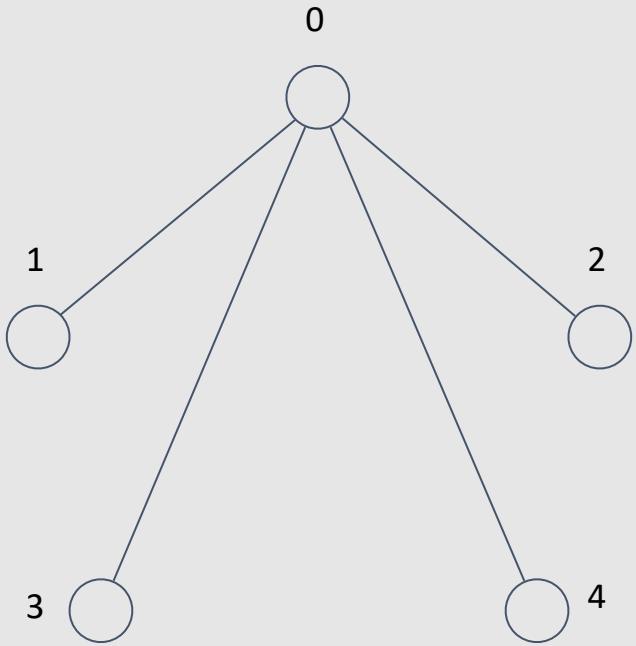
- G = <V, E>
- V = {i}, where i $\subseteq$ N
- E $\subseteq$ V X V
- |E| (size of E) = m
- |V| (size of V) = n

Example

- $V = \{0, 1, 2, 3, 4\}$
- $E = \{(0, 1), (0, 2), (0, 3), (0, 4), (1, 3)\}$

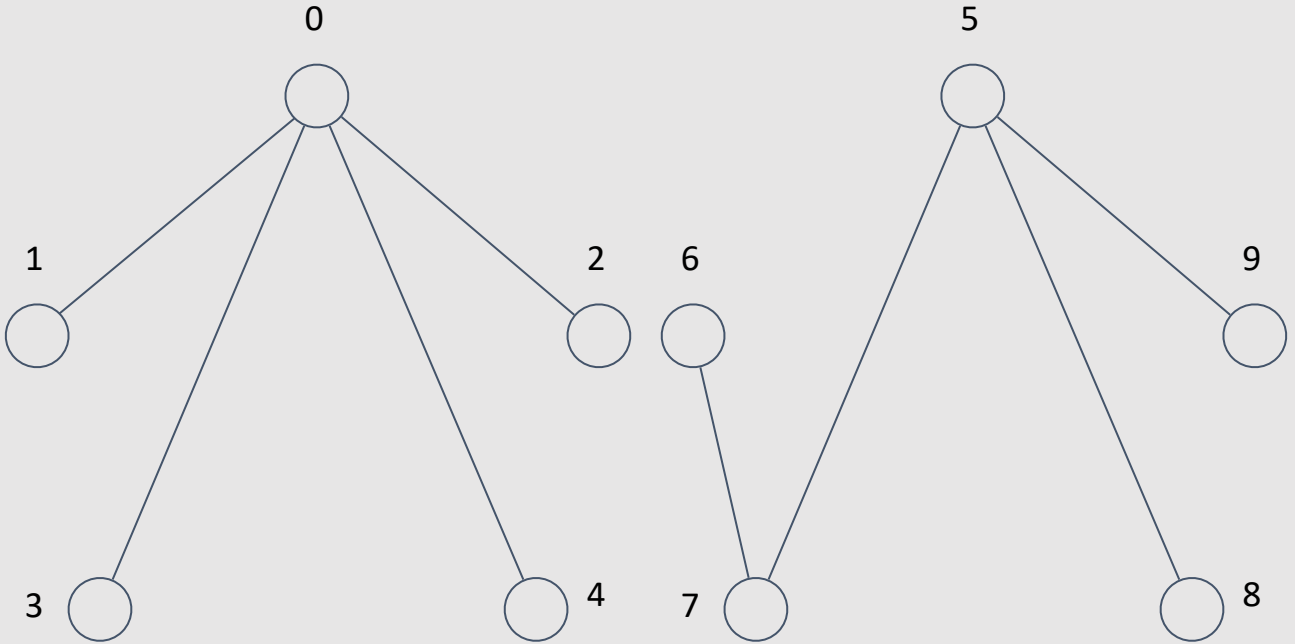
Tree

- A tree is an undirected graph in which any two vertices are connected by exactly one path.
- A tree is a connected acyclic undirected graph.


0
1
2
3
4

Forest

- Any amount(maybe 0) of trees

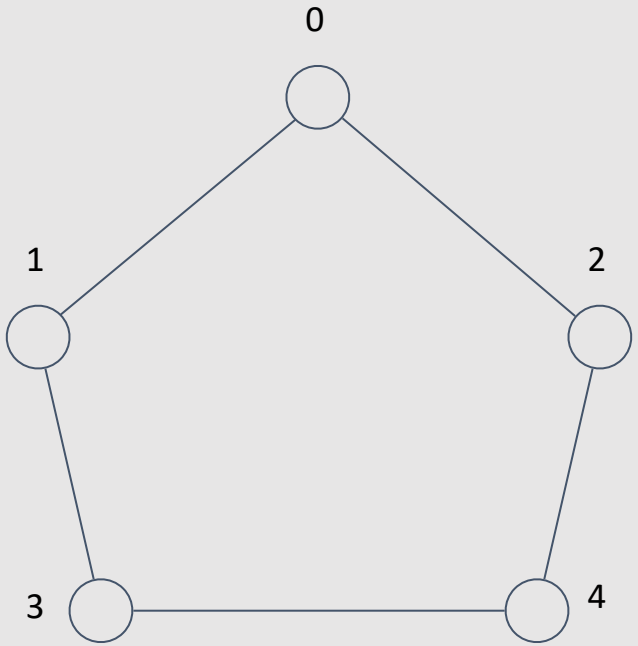

0
1
2
3
4
5
6
7
8
9

Cycle

- A cycle graph is a graph in which the vertices can be listed in an order v1, v2, …, vn such that the edges are the {vi, vi+1} where i = 1, 2, …, n – 1, plus the edge {vn, v1}.
- Cycle graphs is a connected graph in which the degree of all vertices is 2.

Facts

- Amount of trees in forest = n - m
- In 1 tree you will have n vertices and n - 1 edges
- In 2 trees n + n vertices and (n - 1) + (n - 1) edges, so n*2 vertices n * 2 - 2 edges


0
1
2
3
4

How to implement?

- list of edges
- matrix
- list of lists

list of edges

- list of edges - just save all edges
- e.g. [(0, 1), (1, 3), (3, 4), (4, 2), (2, 0)]

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Edge{
    int from, to;
};

int main() {
    int n, m;
    cin >> n >> m;
    vector<Edge> edges;

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        edges.push_back({from, to});
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5360KB

stdin

```
3 4
1 2
2 3
3 1
1 3
```

```python
class Edge:
    def __init__(self, from_, to):
        self.from_ = from_
        self.to = to

n, m = map(int, input().split(' '))

edges = []

for i in range(m):
    from_, to = map(int, input().split(' '))
    edges.append(Edge(from_ - 1, to - 1))

for edge in edges:
    print(edge.from_, edge.to)
```

Success #stdin #stdout 0.04s 9836KB

stdin

```
3 4
1 2
2 3
3 1
1 3
```

stdout

```
0 1
1 2
2 0
0 2
```

Matrix

- 1 iff edge exists, else 0
- e.g. [[0, 1, 1, 0, 0], [1, 0, 0, 1, 0], [1, 0, 0, 0, 1], [0, 1, 0, 0, 1], [0, 0, 1, 1, 0]]

```cpp
#include <bits/stdc++.h>
using namespace std;
array<array<bool, 100>, 100> Matrix;

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        Matrix[from][to] = Matrix[to][from] = true;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << Matrix[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5316KB

stdin

```
3 4
1 2
2 3
3 1
1 3
```

stdout

```
0 1 1
1 0 1
1 1 0
```

```python
n, m = map(int, input().split(' '))

Matrix = []
for _ in range(n):
    Matrix.append([False] * n)

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1][to - 1] = Matrix[to - 1][from_ - 1] = True

print(Matrix)
```

Success #stdin #stdout 0.03s 9816KB

comments

stdin

cop

```
3 4
1 2
2 3
3 1
1 3
```

stdout

cop

```
[[False, True, True], [True, False, True], [True, True, False]]
```

List of lists

- for each vertex we save list of vertices with edges.
- e.g. [(1, 2), (0, 3), (0, 4), (1, 4), (2, 3)]

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    for (int i = 0; i < n; i++) {
        cout << i << " : ";
        for (auto vertex: graph[i]) {
            cout << vertex << ", ";
        }
        cout << "\n";
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5536KB

stdin

```
3 4
1 2
2 3
3 1
1 3
```

stdout

```
0 : 1, 2,
1 : 2,
2 : 0,
```

```python
n, m = map(int, input().split(' '))

Matrix = []
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

print(Matrix)
```

Success #stdin #stdout 0.03s 9876KB

stdin

```
3 4
1 2
2 3
3 1
1 3
```

stdout

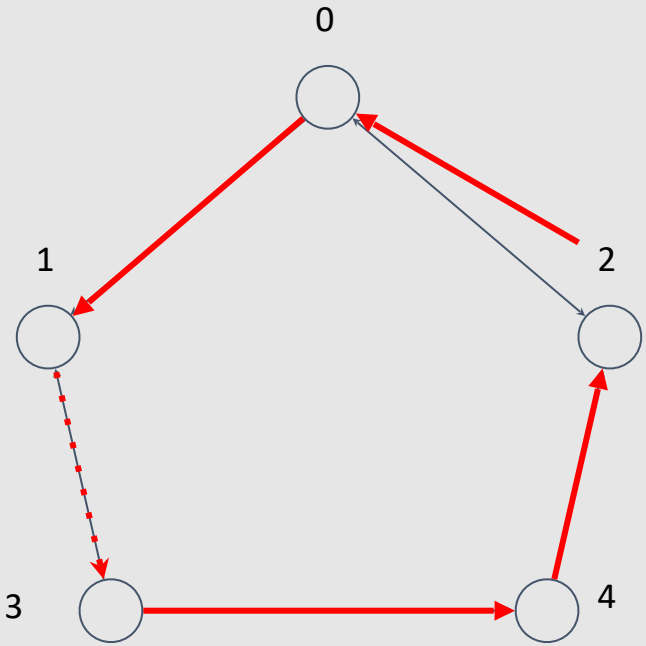
```
[[1, 2], [2], [0]]
```

What to choose?

- List of edges is a very rare solution(dsu, mo, euler)
- If the task is matrix multiplication or Floyd - matrix
- If the graph is pretty sparse - list of lists
- Otherwise - matrix

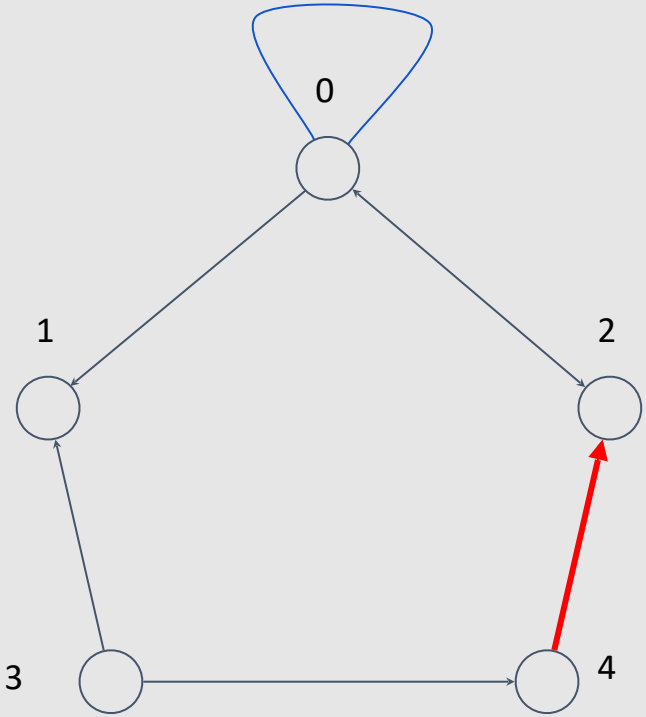
Directed graph

- Each road has a direction. For example, you can imagine a single-lane road.


0
1
2
3
4

Loop/multiedge

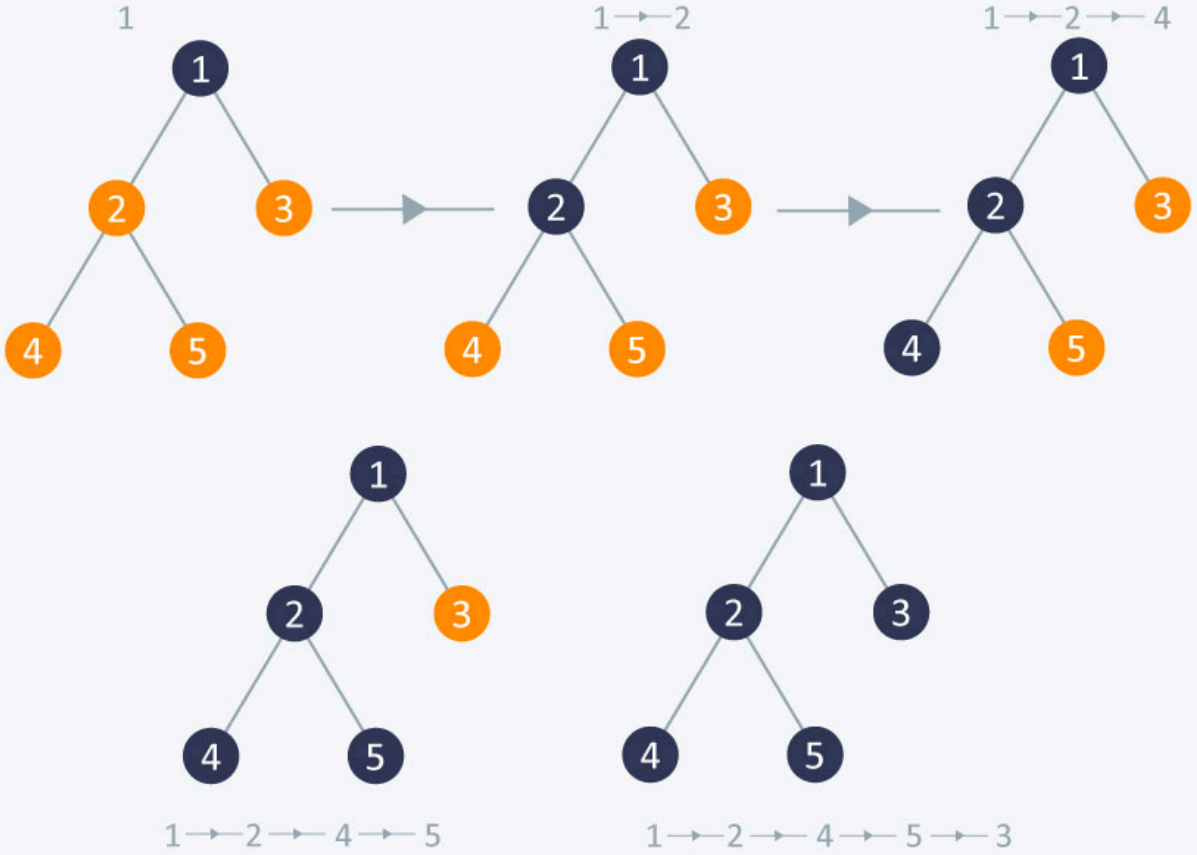
- Loop - (v, v)
- Multiedge - (u, v), (u, v)


0
1
2
3
4

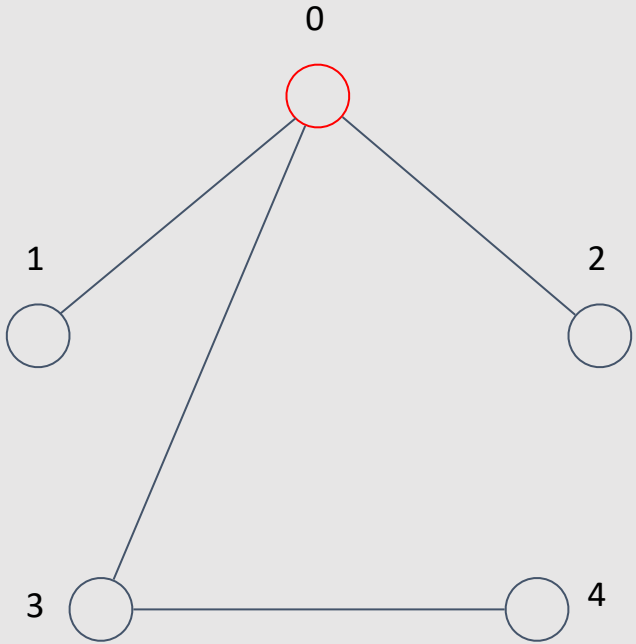
DFS

- We have graph <V, E>. We need to check all vertices.
- The easiest way is just to check recursively every vertex.
- Firstly let’s do with tree

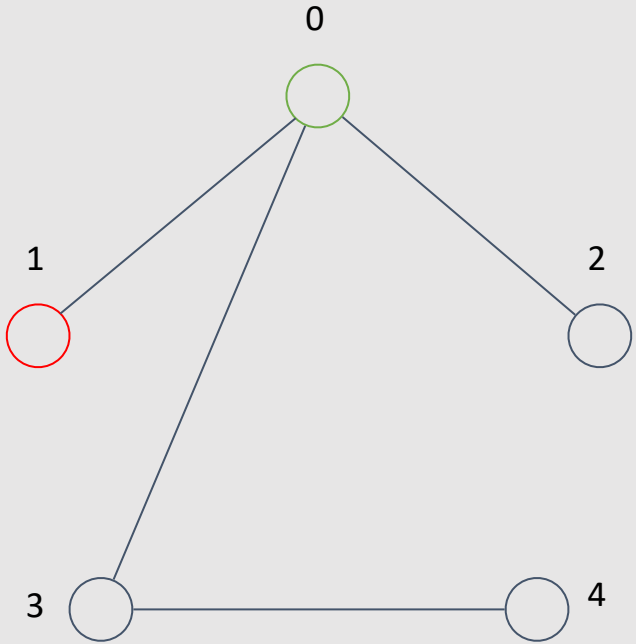
DFS



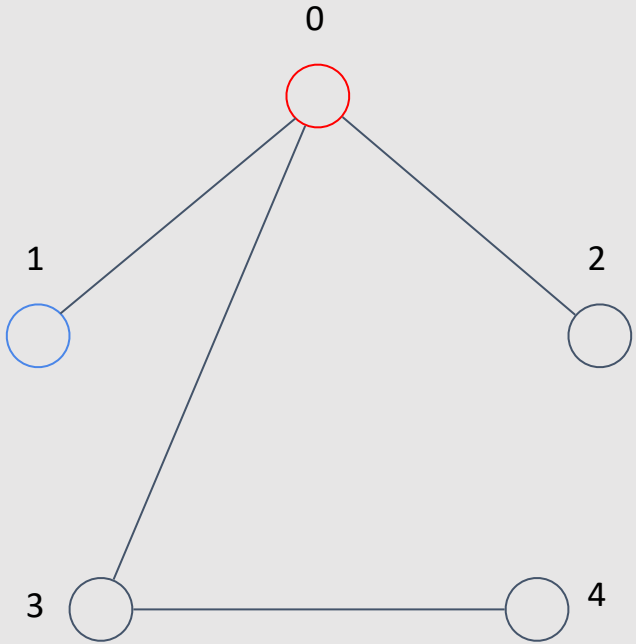
red - current
blue - already visited
black - not yet
green - parent



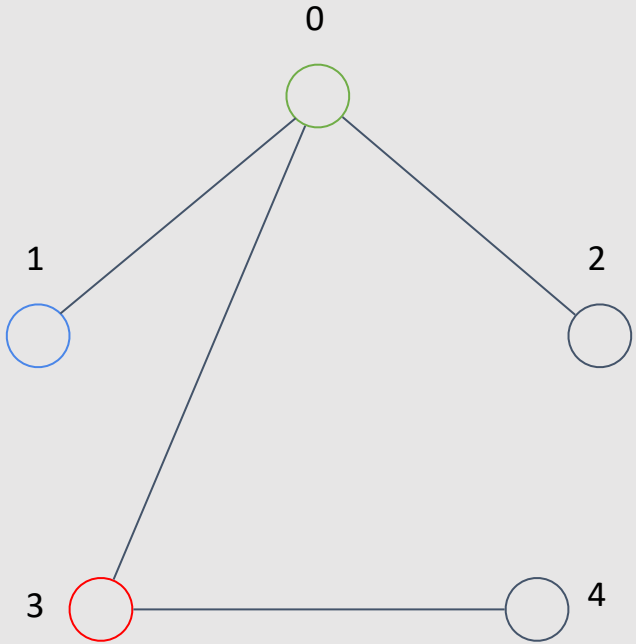
red - current
blue - already visited
black - not yet
green - parent



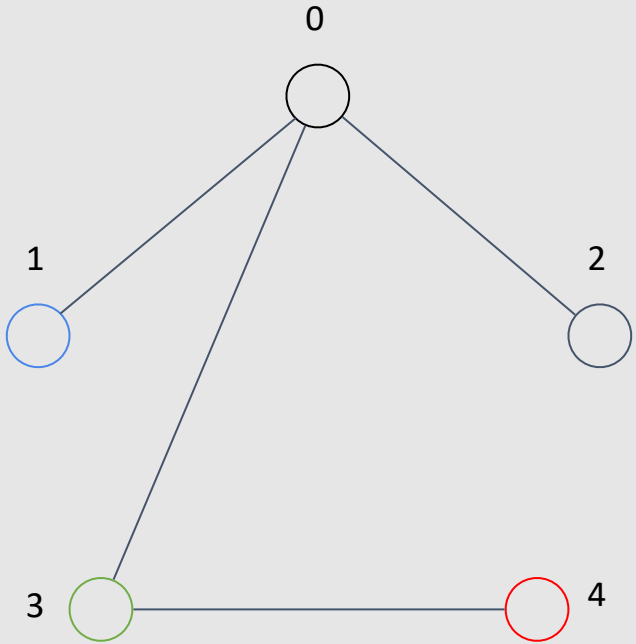
red - current
blue - already visited
black - not yet
green - parent



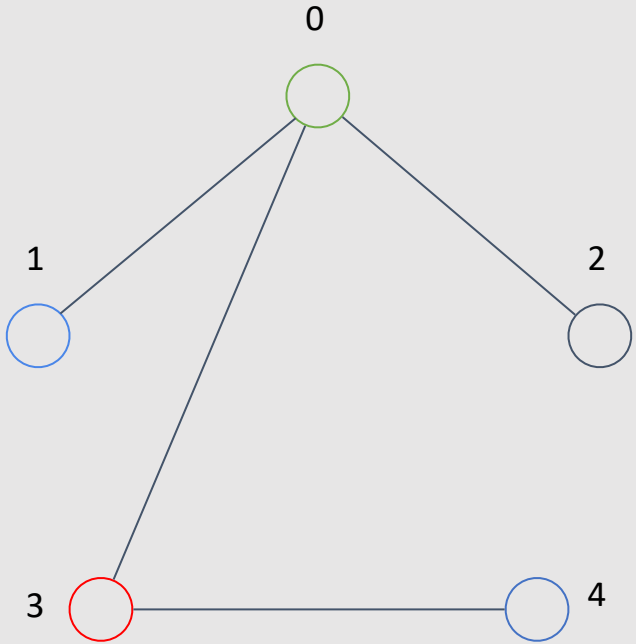
red - current
blue - already visited
black - not yet
green - parent



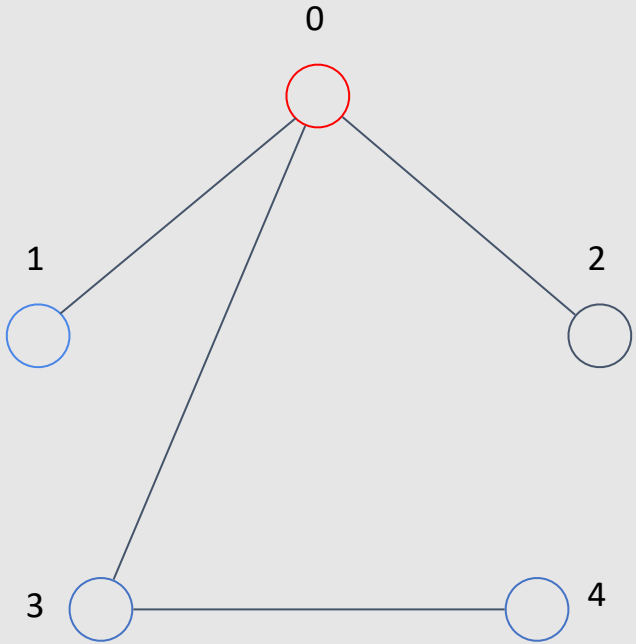
red - current
blue - already visited
black - not yet
green - parent



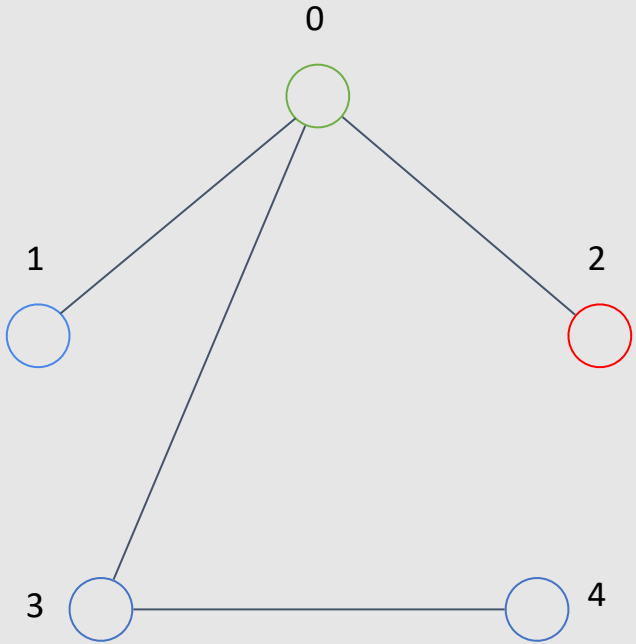
red - current
blue - already visited
black - not yet
green - parent



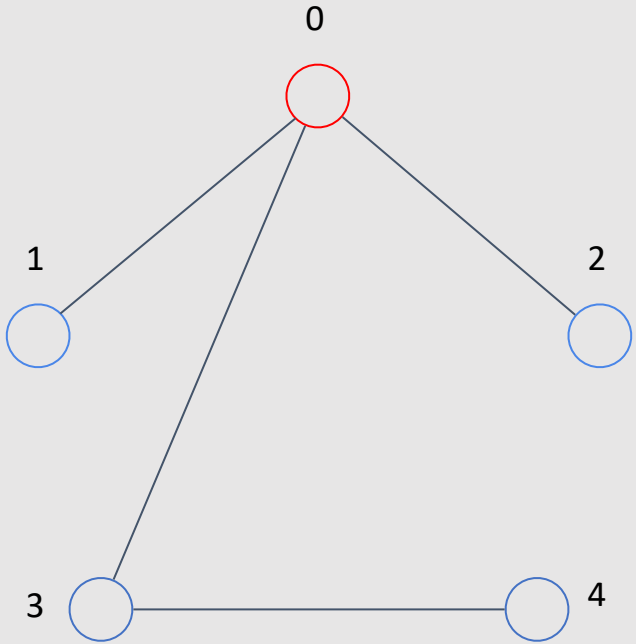
red - current
blue - already visited
black - not yet
green - parent



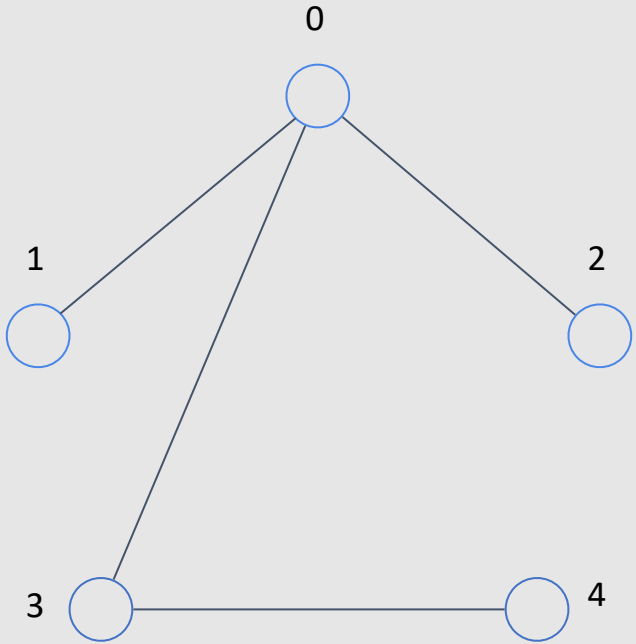
red - current
blue - already visited
black - not yet
green - parent



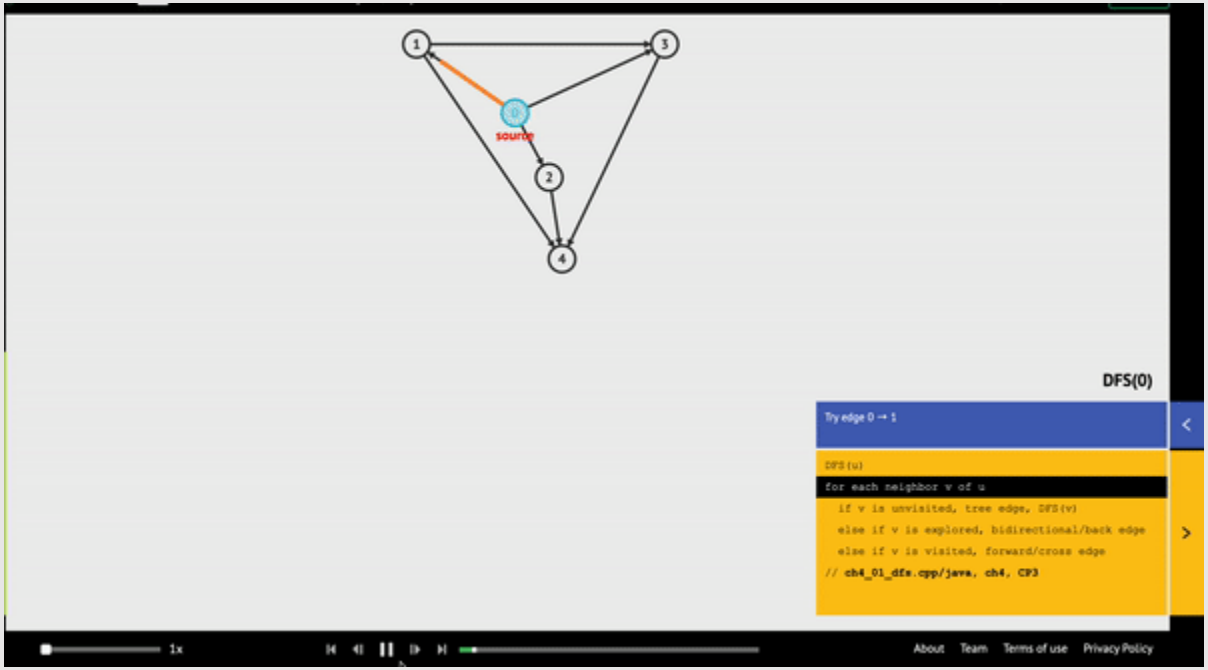
red - current
blue - already visited
black - not yet
green - parent



red - current
blue - already visited
black - not yet
green - parent



Visualization



```cpp
void dfs(int u, int p, const vector<vector<int> > &g) {
    for (auto v: g[u]) {
        if (v != p) {
            dfs(v, u, g);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    dfs(0, 0, graph);

    return 0;
}
```

Success #stdin #stdout 0.01s 5392KB

comments (0)

stdin

copy

```
4 3
1 2
2 3
1 4
```

stdout

copy

```
Standard output is empty
```

```python
def DFS(g, u, p):
    for v in g[u]:
        if v != p:
            DFS(g, v, u)

n, m = map(int, input().split(' '))

Matrix = []
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, 0)

```

Success #stdin #stdout 0.03s 9628KB

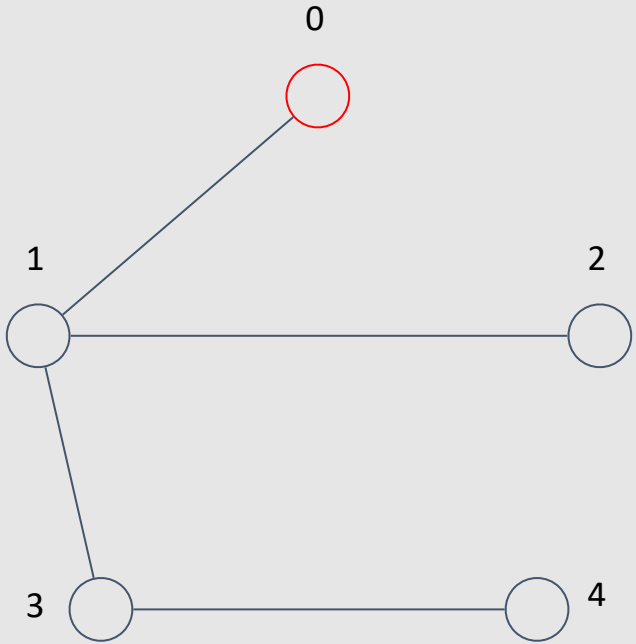
stdin

```
3 2
1 2
2 3
```

stdout

```
Standard output is empty
```

red - current
blue - already visited
black - not yet
green - parent



DFS

- With graph better not to keep parent and to keep array of visited vertices.

```cpp
void dfs(int u, const vector<vector<int> > &g, vector<bool> &visited) {
    if (visited[u]) {
        return;
    }
    visited[u] = true;
    for (auto v: g[u]) {
        dfs(v, g, visited);
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    dfs(0, graph, visited);

    return 0;
}
```

Success #stdin #stdout 0.01s 5460KB

comments (0)

stdin

copy

```
4 4
1 2
2 3
1 4
4 3
```

```python
def DFS(g, u, visited):
    visited[u] = True

    for v in g[u]:
        if not visited[v]:
            DFS(g, v, visited)

n, m = map(int, input().split(' '))

Matrix = []
visited = [False] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, visited)

print(visited)

```

Success #stdin #stdout 0.03s 9624KB

stdin

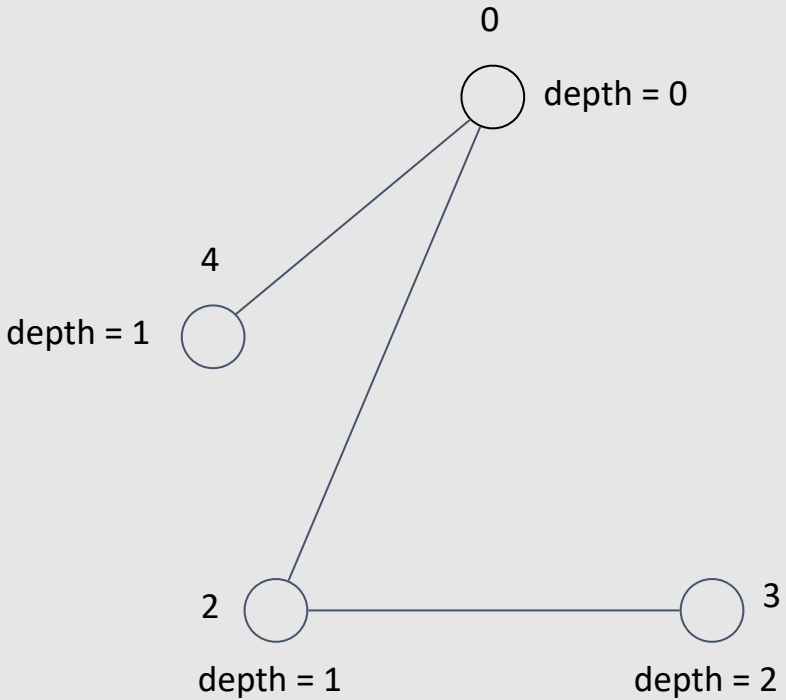
```
3 4
1 2
2 3
3 1
1 3
```

stdout

```
[True, True, True]
```

DFS

- We can find depth(only trees) with DFS for example


0
depth = 0
4
depth = 1
2
depth = 1
3
depth = 2

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(int u, int p, int cur_d, const vector<vector<int> > &g, vector<int> &d) {
    d[u] = cur_d;
    for (auto v: g[u]) {
        if (v != p) {
            dfs(v, u, cur_d + 1, g, d);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<int> depth(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }
```

```
    dfs(0, 0, 0, graph, depth);

    for (int i = 0; i < n; i++) {
        cout << depth[i] << " ";
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5508KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
0 1 2 1
```

```python
def DFS(g, u, p, depth):
    for v in g[u]:
        if p != v:
            depth[v] = depth[u] + 1
            DFS(g, v, u, depth)

n, m = map(int, input().split(' '))

Matrix = []
depth = [0] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, 0, depth)

print(depth)

```

Success #stdin #stdout 0.03s 9572KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

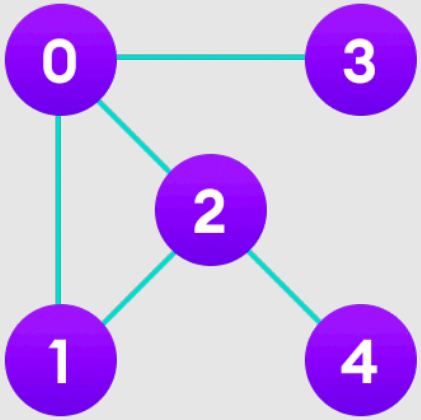
```
[0, 1, 2, 1]
```

Asymptotic

- O(m) - maximum of actions in cycle
- O(n) - maximum of recursion calls

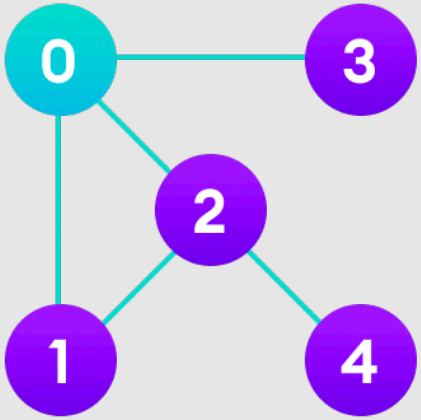
non-recursive DFS

- Now, instead of recursion, we will use a stack.
- Using a stack, we can emulate which vertex to take next



Visited

Stack

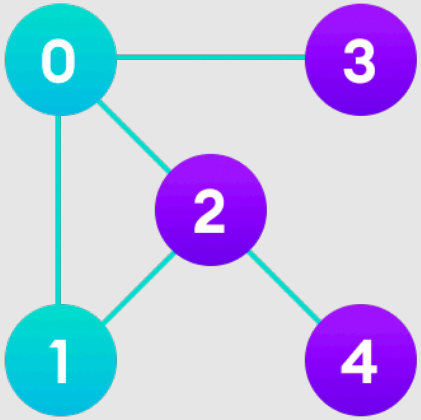


0				

Visited

1	2	3		

Stack

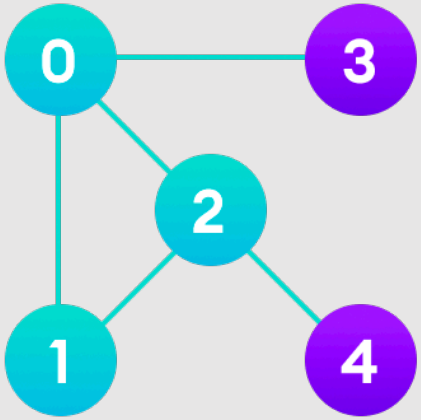


0	1			

Visited

2	3			

Stack

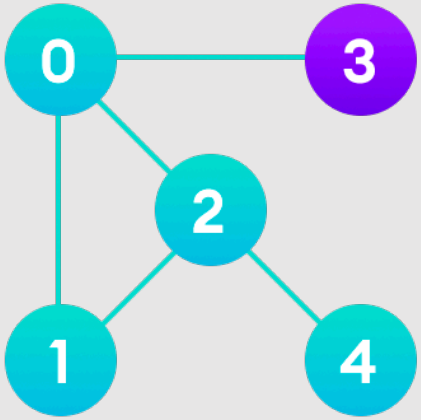


0	1	2		

Visited

4	3			

Stack

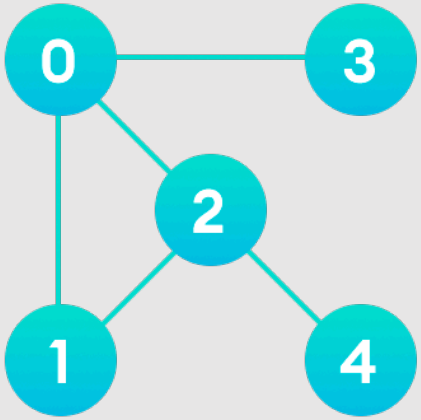


0	1	2	4	

Visited

3				

Stack



0	1	2	4	3

Visited

Stack

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(int root, const vector<vector<int> > &g, vector<bool> &visited) {
    stack<int> vertices;
    vertices.push(root);
    while (!vertices.empty()) {
        auto u = vertices.top();
        if (!visited[u]) {
            visited[u] = true;
            vertices.pop();
            for (auto v: g[u]) {
                if (!visited[v]) {
                    vertices.push(v);
                }
            }
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }
```

```
32.          graph[from].push_back(to);
33.      }
34.
35.      dfs(0, graph, visited);
36.
37.      for (int i = 0; i < n; i++) {
38.          cout << visited[i] << " ";
39.      }
40.
41.      return 0;
42.  }
```

Success #stdin #stdout 0.01s 5548KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
1 1 1 1
```

```python
def DFS(g, root, p, visited):
    stack = [root]

    while stack:
        u = stack.pop()
        if not visited[u]:
            visited[u] = True

            for v in g[u]:
                if not visited[v]:
                    stack.append(v)

n, m = map(int, input().split(' '))

Matrix = []
visited = [False for _ in range(n)]
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, 0, visited)

print(visited)

```

Success #stdin #stdout 0.04s 9552KB

comments (0)

stdin

copy

```
4 3
1 2
2 3
1 4
```

stdout

copy

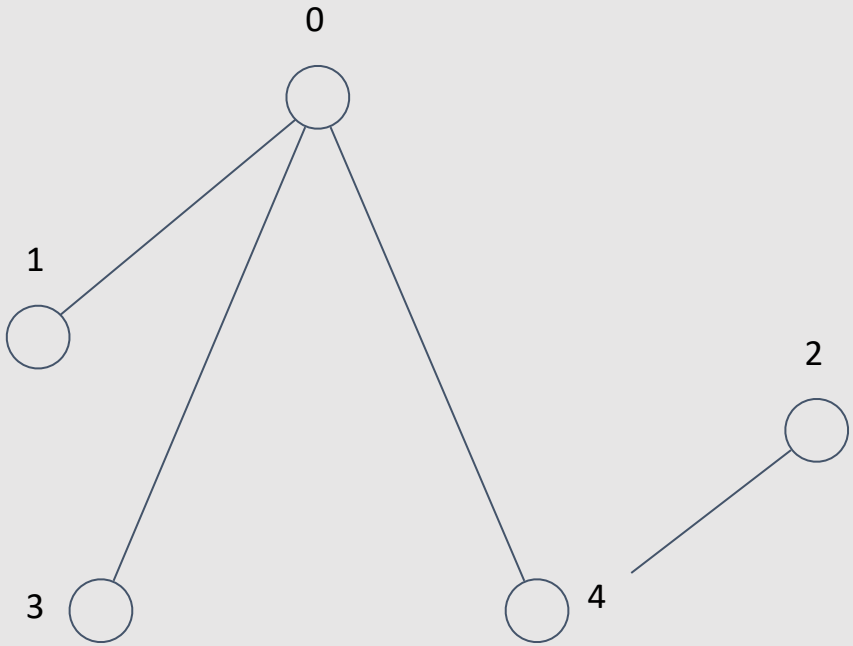
```
[True, True, True, True]
```

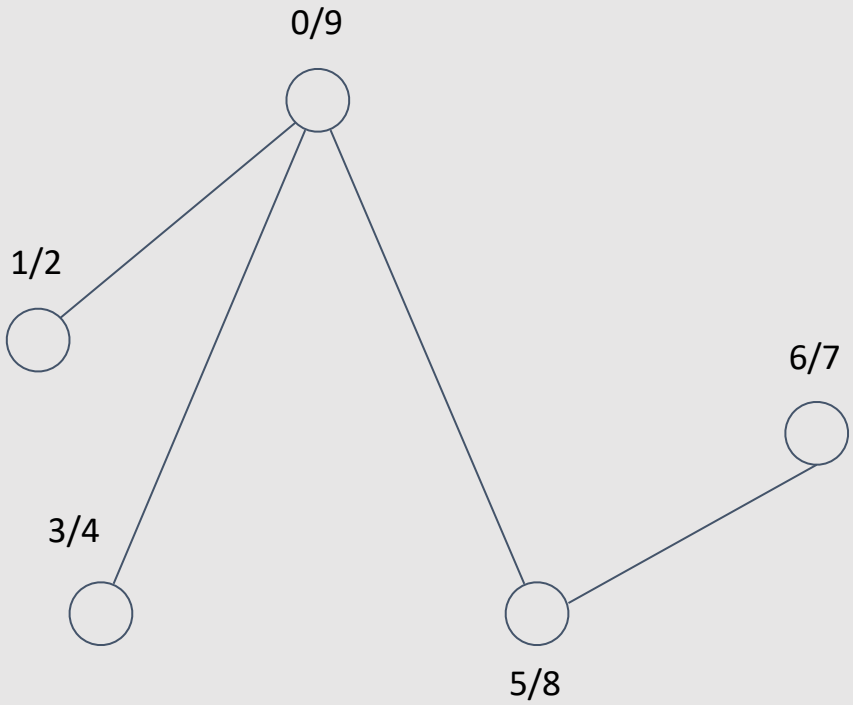
Asymptotic

- O(m) - maximum of actions in cycle
- O(m) - maximum of elements in stack

Timers

- Let’s create timer
- Every time we go in or from vertex -> timer++
- tin = time in
- tout = time out


0
1
2
3
4


0/9
1/2
6/7
3/4
5/8

Euler tour

- Let’s put the vertex in list, every time we change her timer
- Example - [0, 1, 1, 3, 3, 4, 2, 2, 4, 0]

```
4.  void dfs(int u, const vector<vector<int> > &g, vector<bool> &visited, int &timer, vector<int> &eul
    er, vector<int> &tin, vector<int> &tout) {
5.      if (visited[u]) {
6.          return;
7.      }
8.      tin[u] = timer++;
9.      euler.push_back(u);
10.     visited[u] = true;
11.     for (auto v: g[u]) {
12.         dfs(v, g, visited, timer, euler, tin, tout);
13.     }
14.     tout[u] = timer++;
15.     euler.push_back(u);
16. }
17.
18. int main() {
19.     int n, m, timer = 0;
20.     cin >> n >> m;
21.     vector<vector<int> > graph(n);
22.     vector<bool> visited(n);
23.     vector<int> euler, tin(n), tout(n);
24.
```

```
    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    dfs(0, graph, visited, timer, euler, tin, tout);

    for (auto elem: euler) {
        cout << elem << " ";
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5532KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
0 1 2 2 1 3 3 0
```

```python
def DFS(g, u, visited, euler, tin, tout):
    if visited[u]:
        return
    visited[u] = True
    tin[u] = len(euler)
    euler.append(u)

    for v in g[u]:
        DFS(g, v, visited, euler, tin, tout)

    tout[u] = len(euler)
    euler.append(u)

n, m = map(int, input().split(' '))

Matrix, euler = [], []
visited, tin, tout = [False] * n, [0] * n, [0] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, visited, euler, tin, tout)

print(tin, tout, euler)

```

Success #stdin #stdout 0.03s 9664KB comm

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
[0, 1, 2, 5] [7, 4, 3, 6] [0, 1, 2, 2, 1, 3, 3, 0]
```

How to check for ancestor?

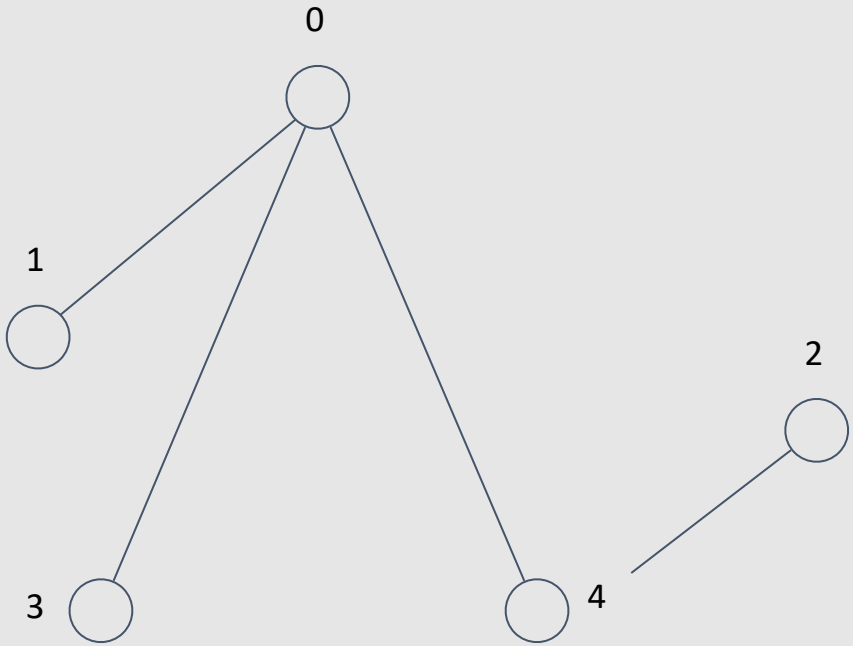
- tin[u] <= tin[v] <= tout[v] <= tout[u] <-> v - son of u

Why?

- tin[u] <= tin[v] <= tout[u] - v is in subtree of u

Another euler tour

- Every time we go in or return to vertex -> add it to euler tour


0
1
2
3
4

Another euler tour

- Example - [0, 1, 0, 3, 0, 4, 2, 4, 0]

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(int u, const vector<vector<int> > &g, vector<bool> &visited, int &timer, vector<int> &eul
er) {
    if (visited[u]) {
        return;
    }
    euler.push_back(u);
    visited[u] = true;
    for (auto v: g[u]) {
        dfs(v, g, visited, timer, euler);
        euler.push_back(u);
    }
}

int main() {
    int n, m, timer = 0;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);
    vector<int> euler;

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }
```

```
30.
31.     dfs(0, graph, visited, timer, euler);
32.
33.     for (auto elem: euler) {
34.         cout << elem << " ";
35.     }
36.     return 0;
37. }
```

Success #stdin #stdout 0.01s 5408KB

comments (0)

stdin

copy

```
5 4
1 2
1 4
1 5
5 3
```

stdout

copy

```
0 1 0 3 0 4 2 4 0
```

```python
def DFS(g, u, visited, euler, tin, tout):
    if visited[u]:
        return
    visited[u] = True
    euler.append(u)

    for v in g[u]:
        DFS(g, v, visited, euler, tin, tout)
        euler.append(u)

n, m = map(int, input().split(' '))

Matrix, euler = [], []
visited, tin, tout = [False] * n, [0] * n, [0] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

DFS(Matrix, 0, visited, euler, tin, tout)

print(euler)

```

Success #stdin #stdout 0.03s 9700KB

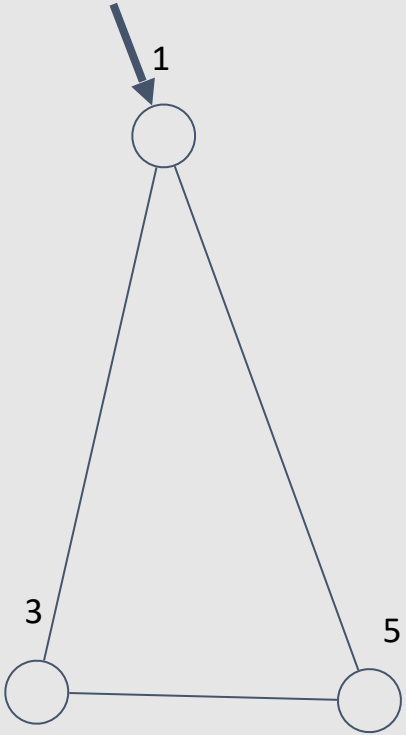
stdin

```
5 4
1 2
1 4
1 5
5 3
```

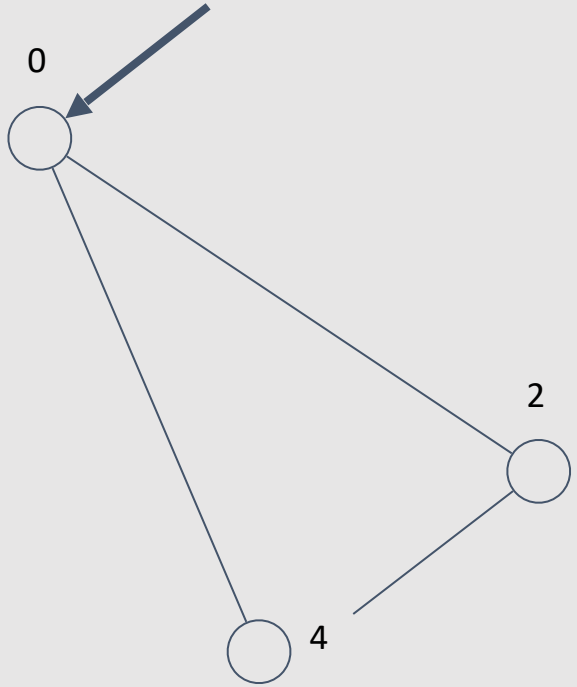
stdout

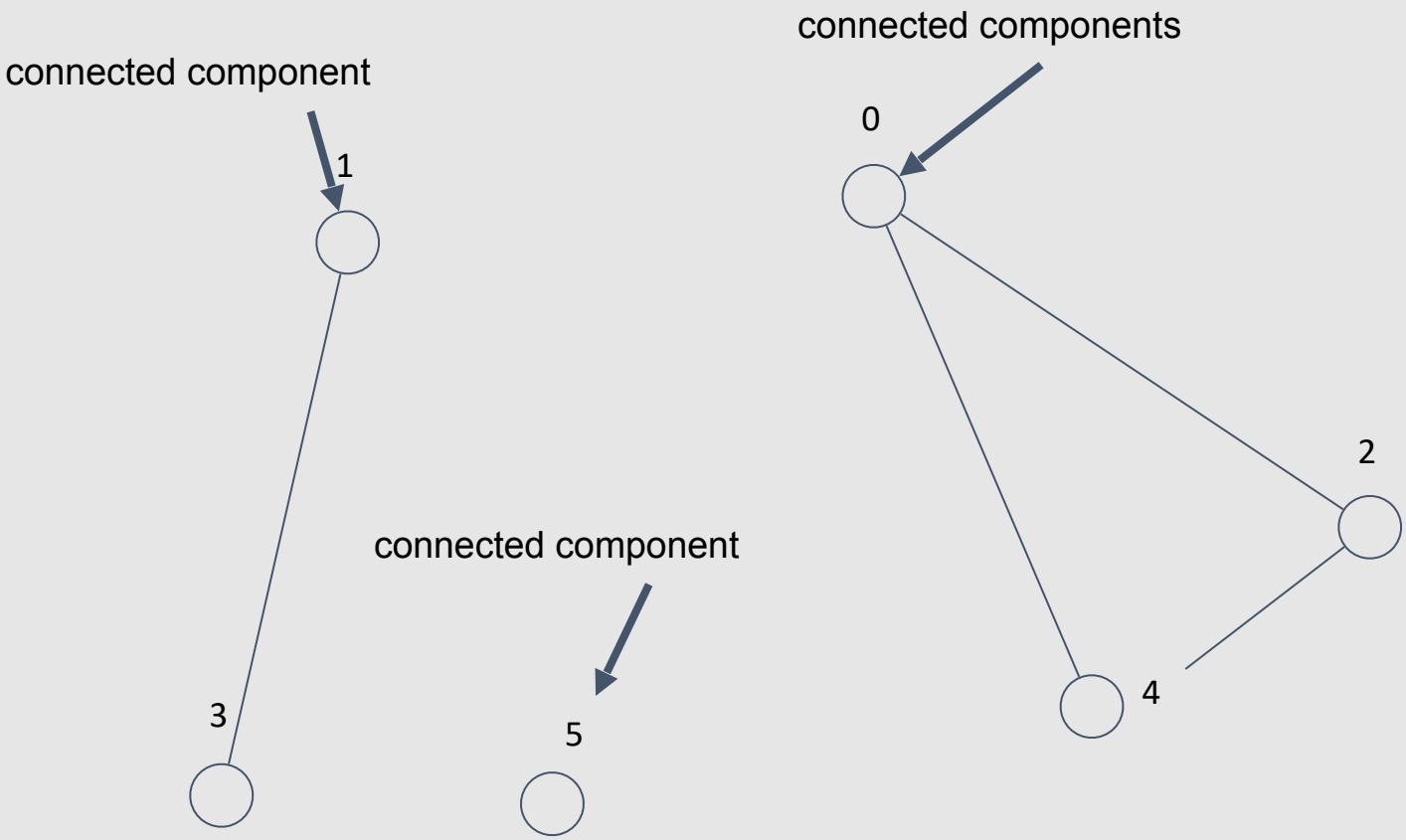
```
[0, 1, 0, 3, 0, 4, 2, 4, 0]
```

connected component



connected components

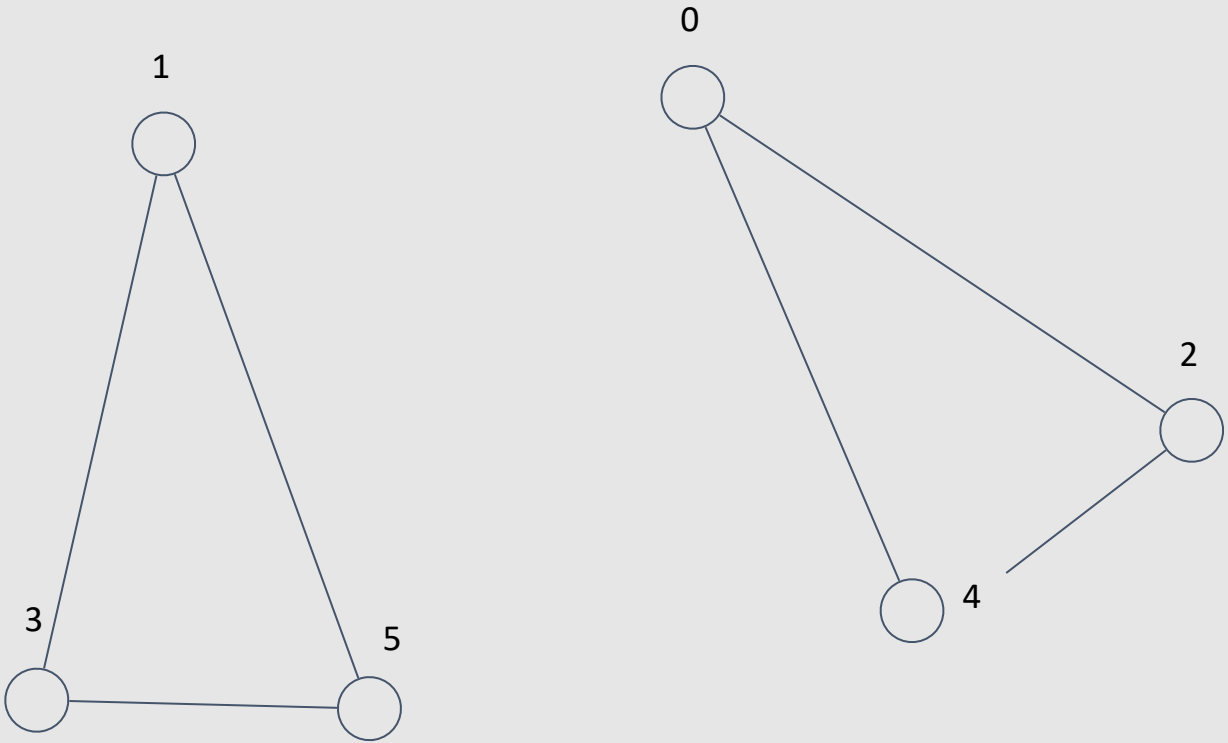



connected component
1
3
connected component
5
connected components
0
2
4

Amount of components

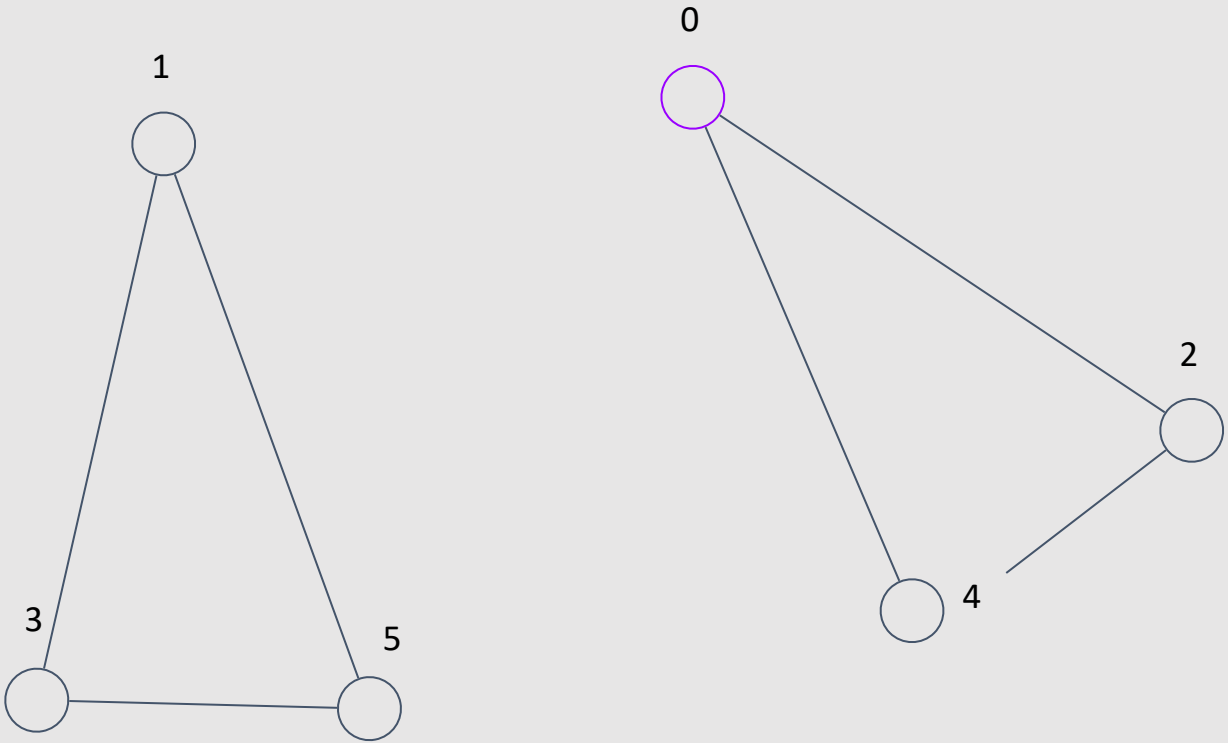
- Component of an undirected graph is a connected subgraph that is not part of any larger connected subgraph
- We want to find amount of components in graph

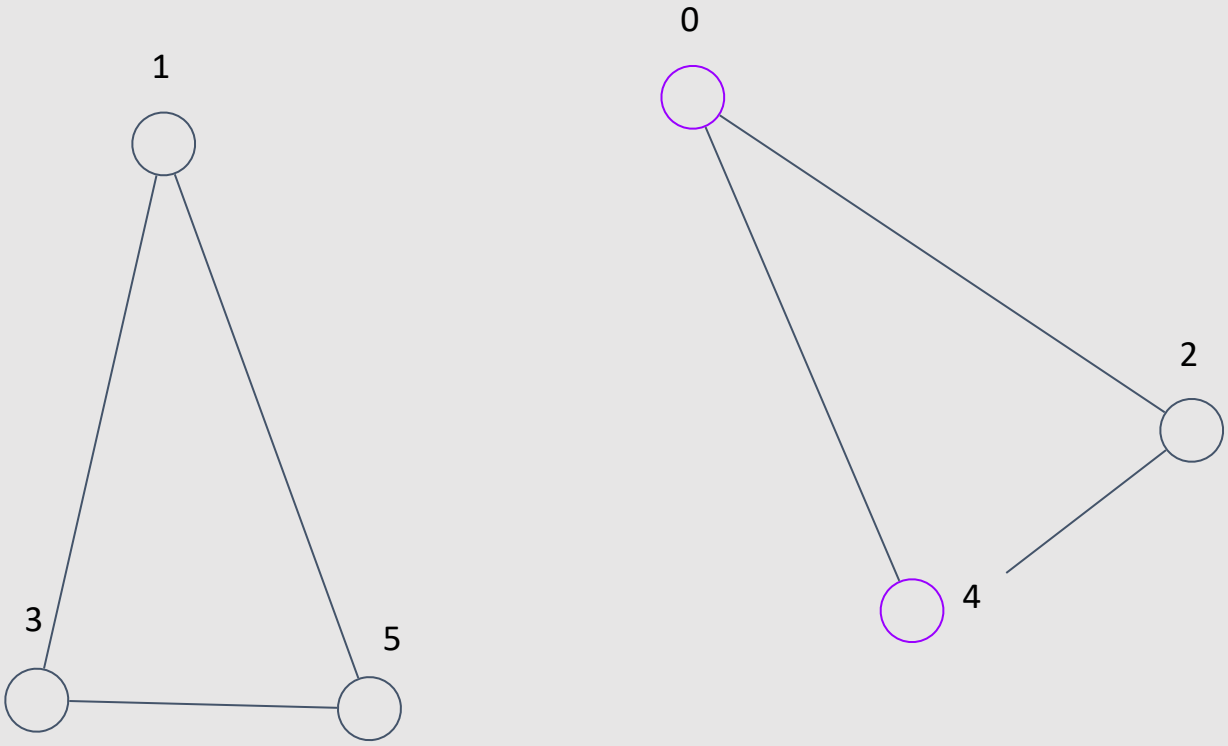
2 connected components

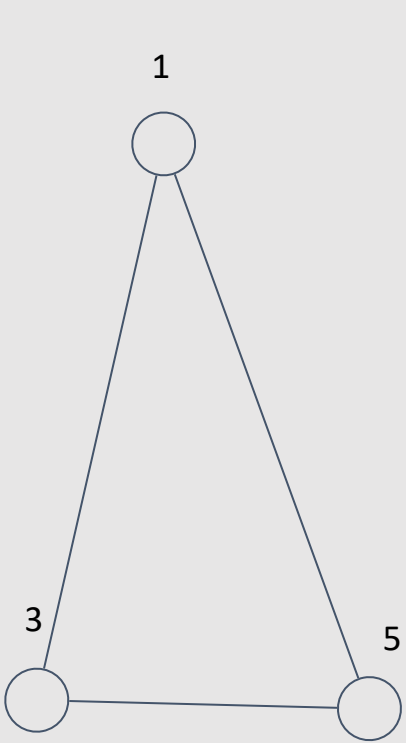


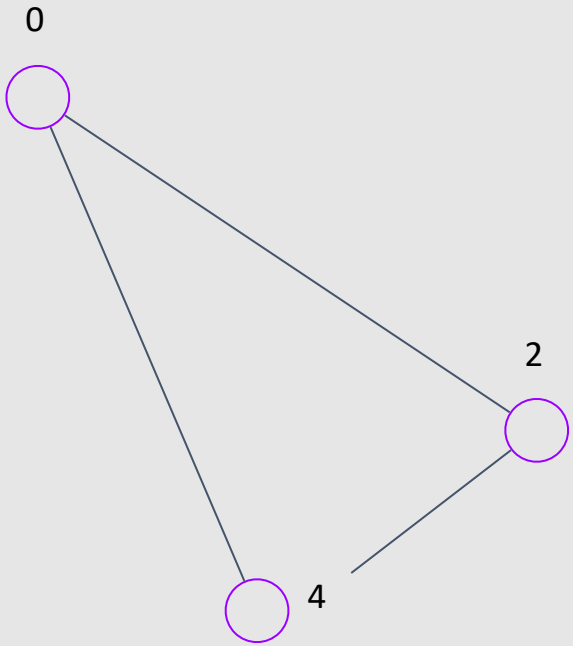
Idea

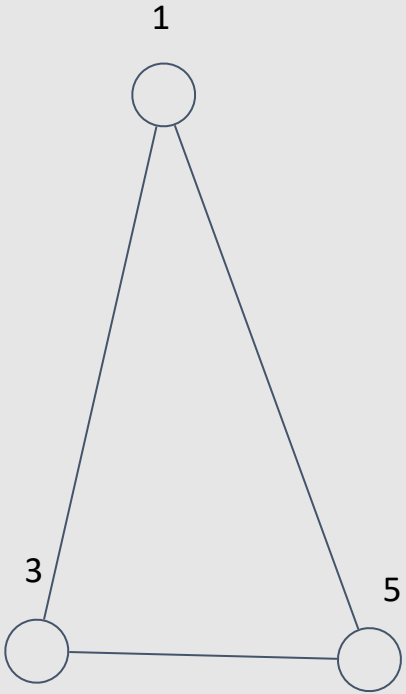
- In dfs we mark all vertices in component
- Amount of components = amount of dfs functions from not marked components.

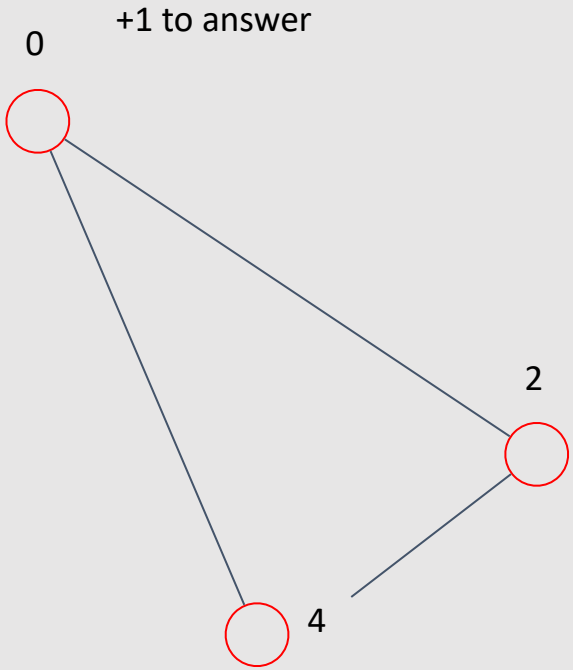

1
3
5
0
2
4

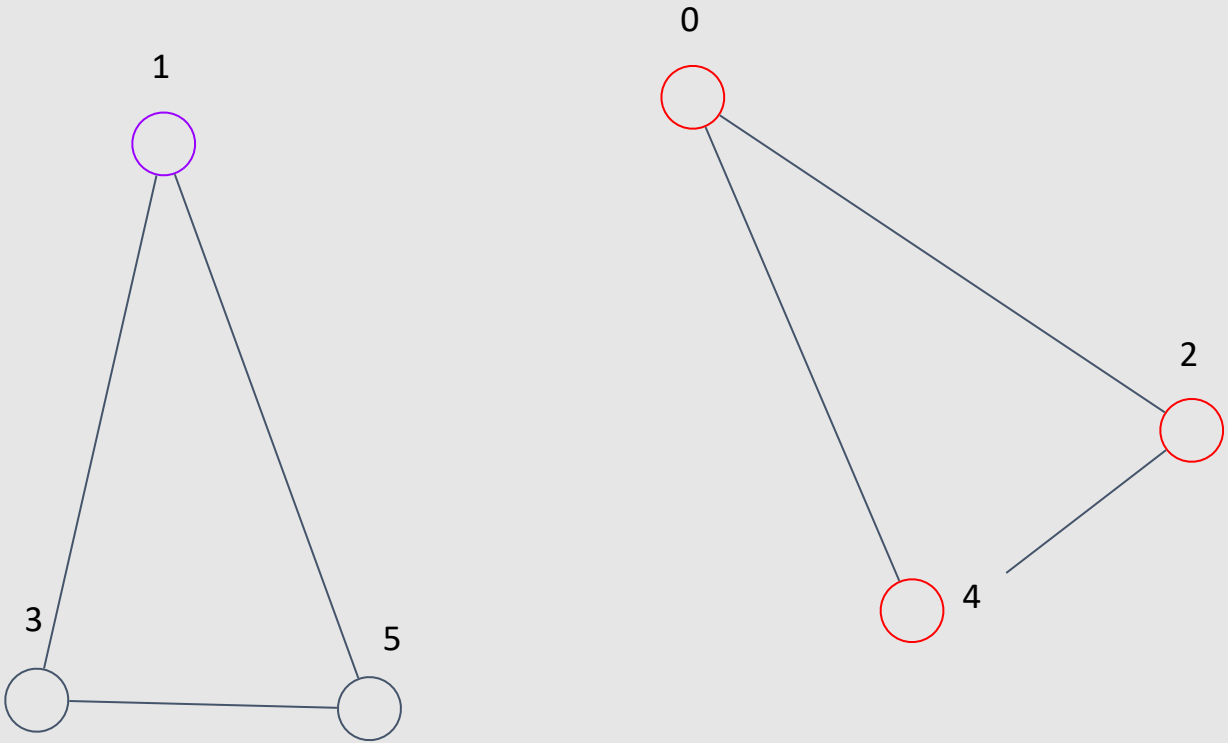

1
3
5
0
2
4

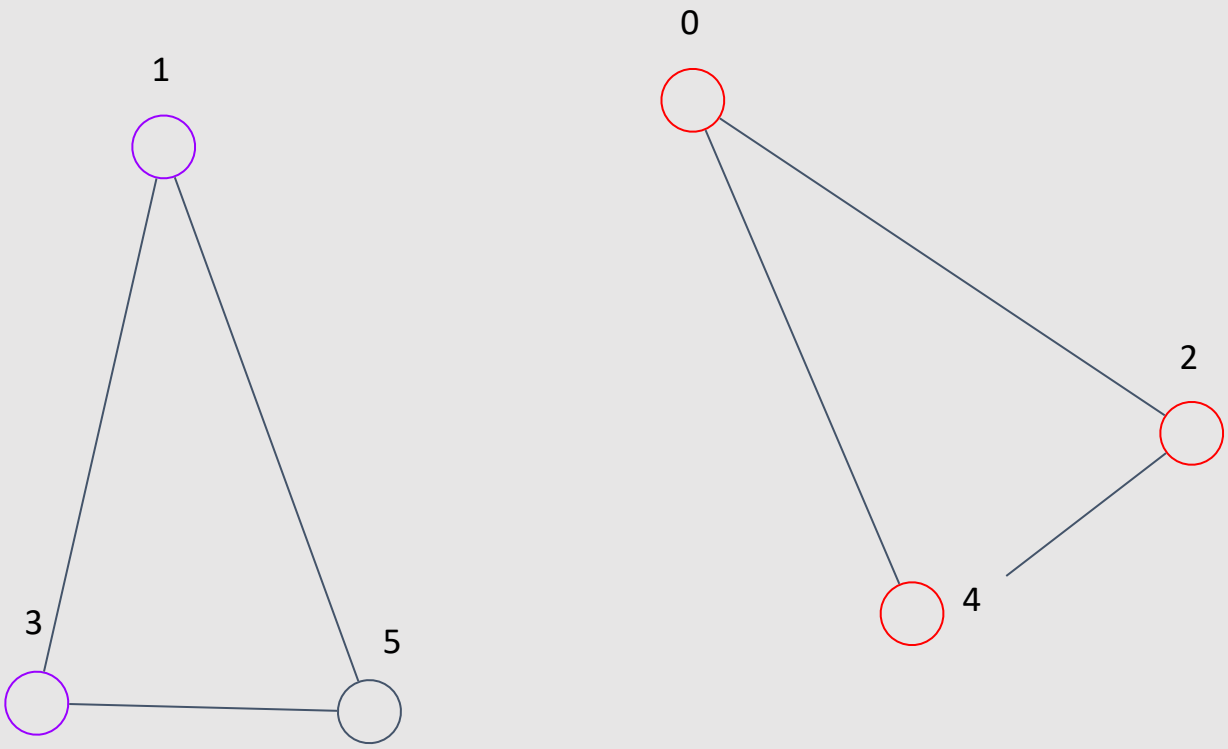

1
3
5

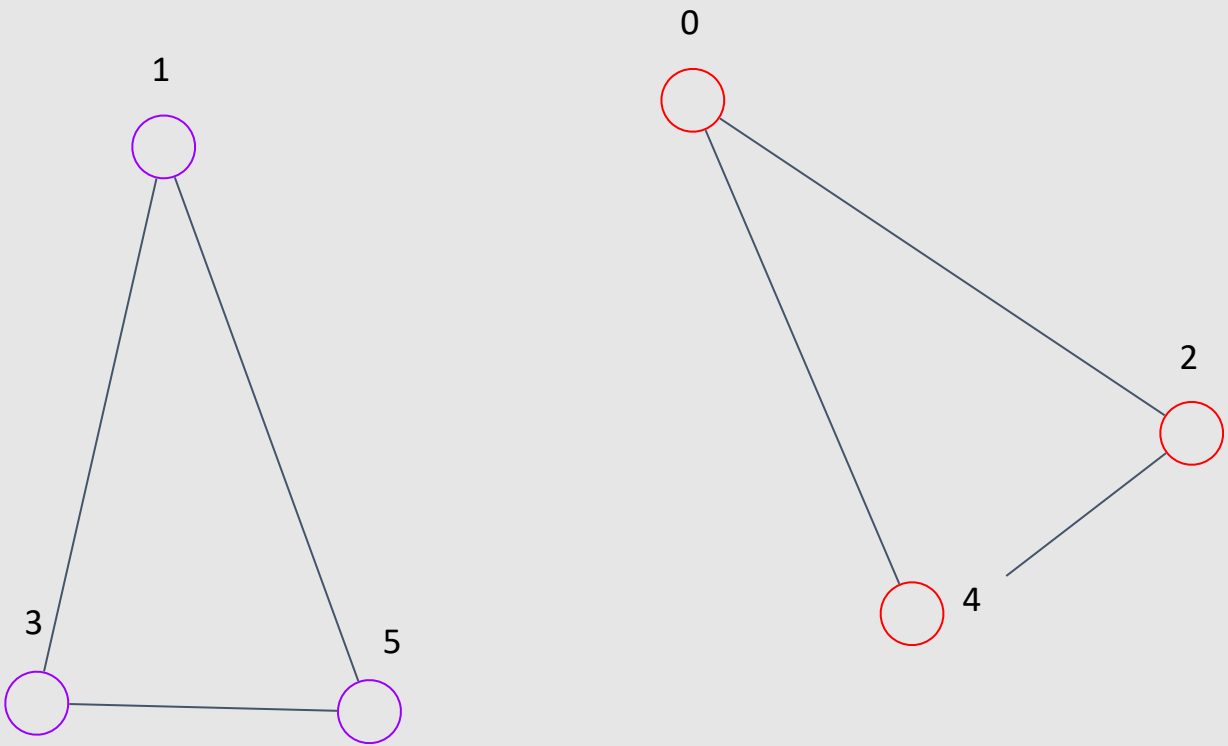

0
2
4


1
3
5

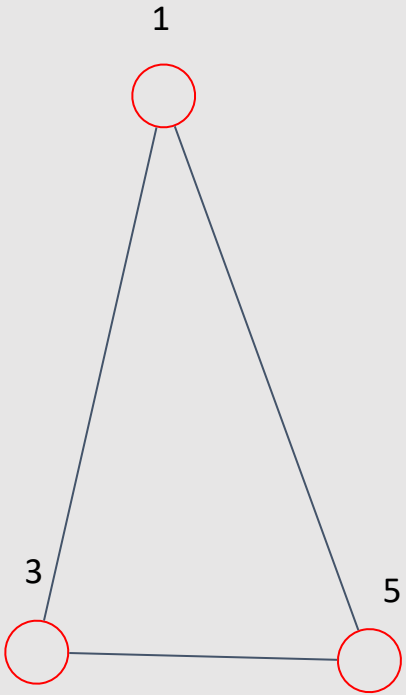

+1 to answer
0
2
4

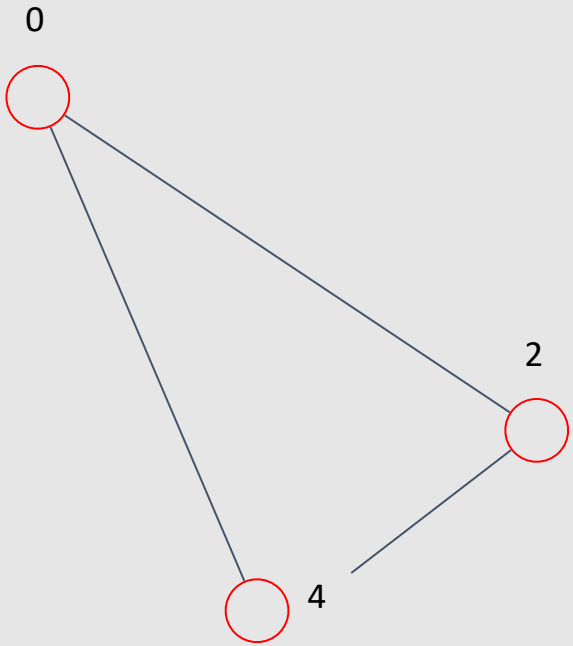

1
3
5
0
2
4


1
3
5
0
2
4


1
3
5
0
2
4

+1 to answer





```
28.     int answer = 0;
29.     for (int i = 0; i < n; i++) {
30.         if (!visited[i]) {
31.             dfs(i, graph, visited);
32.             answer += 1;
33.         }
34.     }
35.
36.     cout << answer;
37.     return 0;
38. }
```

Success #stdin #stdout 0.01s 5472KB

comments (0)

stdin

copy

```
6 6
2 4
2 6
6 4
1 5
1 3
5 3
```

stdout

copy

```
2
```

```
answer = 0
for i in range(n):
    if not visited[i]:
        answer += 1
        DFS(Matrix, i, visited)

print(answer)
```

Success #stdin #stdout 0.03s 9648KB

comments (?)

stdin

copy

```
6 6
2 4
2 6
6 4
1 5
1 3
5 3
```

stdout

copy

```
2
```

Cycle

- We need to check if there are cycles in the graph. How can we do this using DFS?

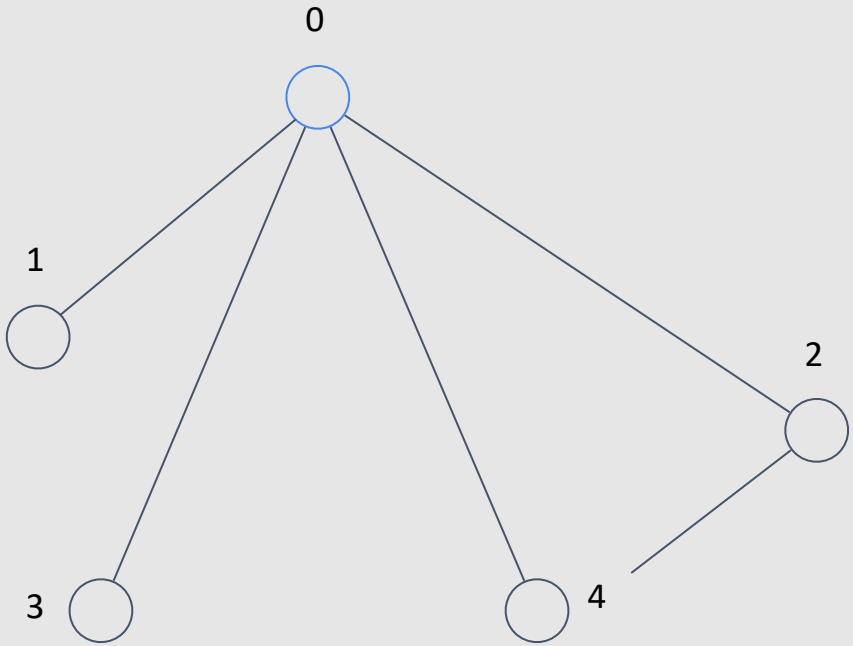
Idea

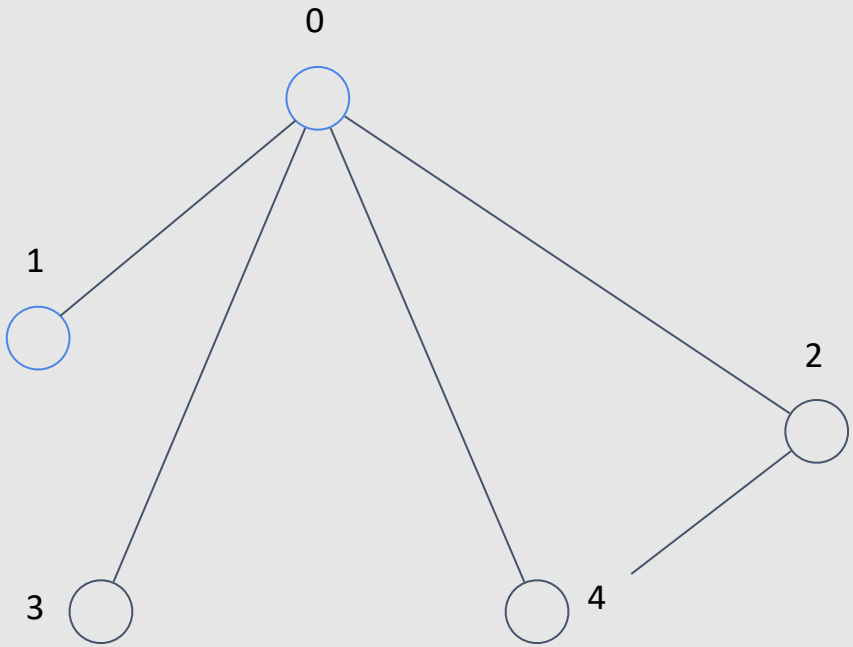
Three types:

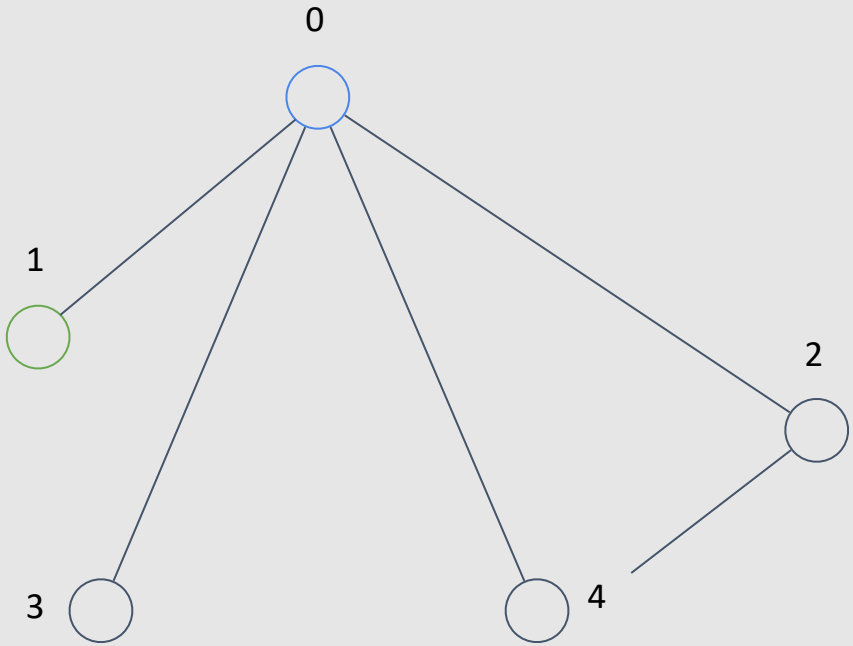
- black vertex - not visited yet
- blue vertex - visited, but opened(tout not filled)
- green vertex - visited and closed(tout filled)

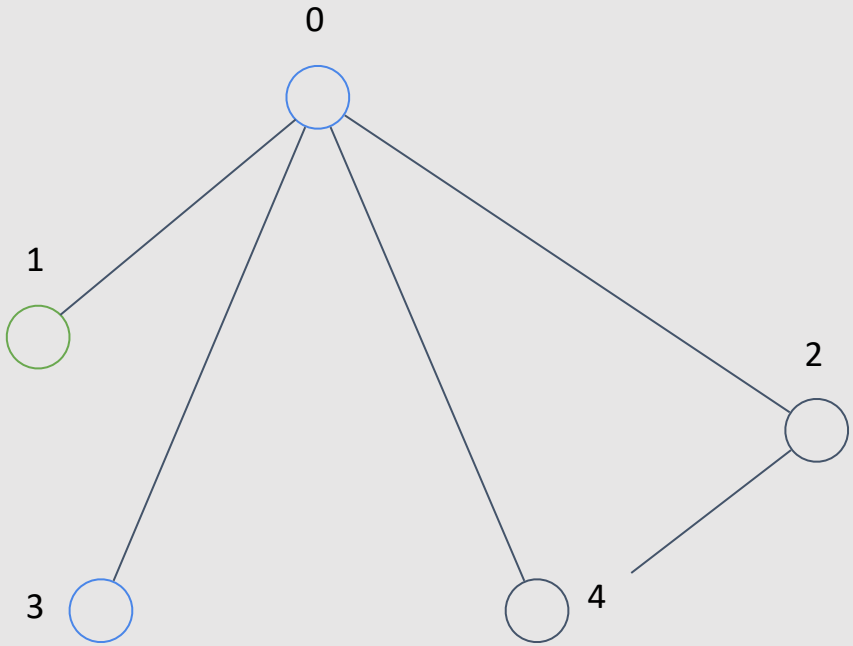
Idea

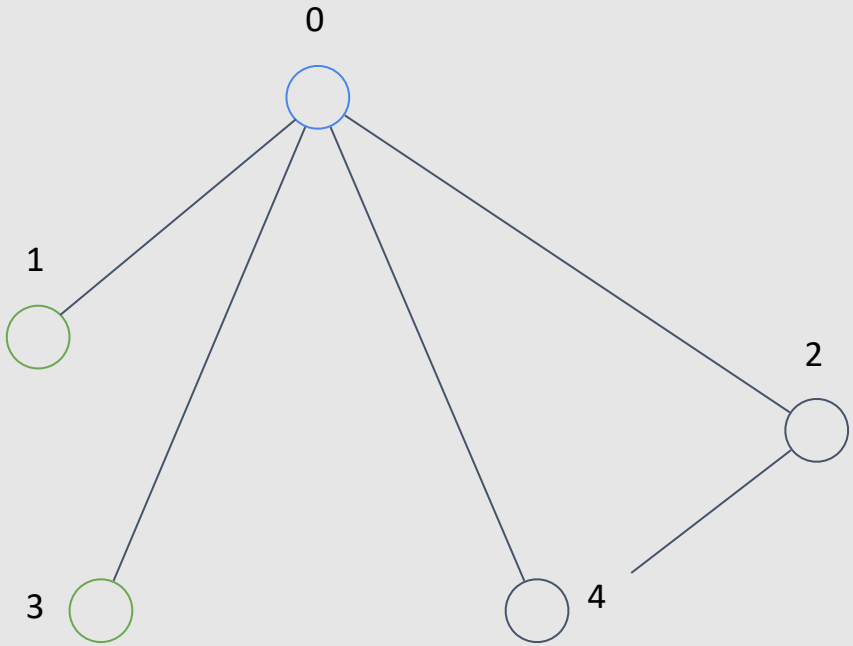
Cycle = we have an edge from blue to blue

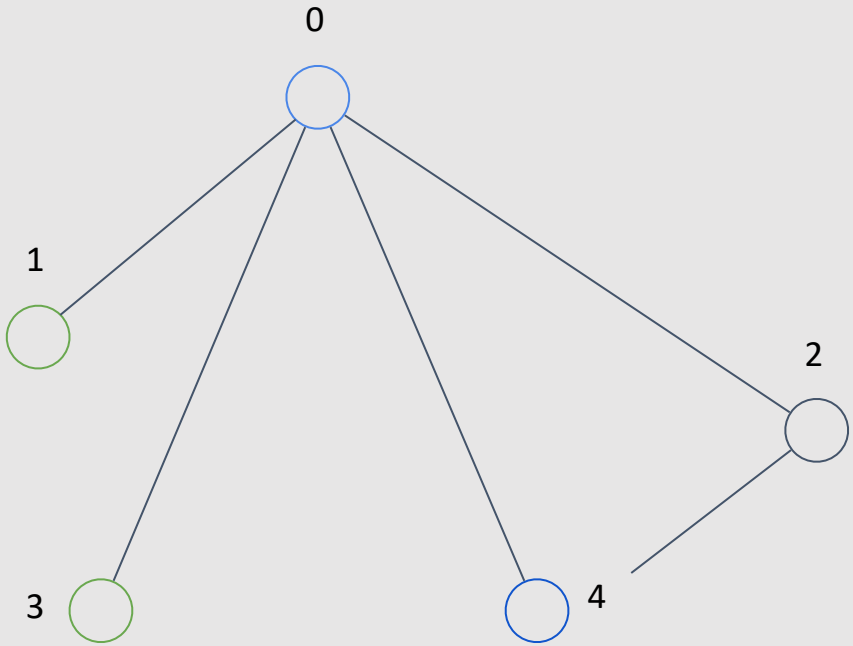

0
1
2
3
4

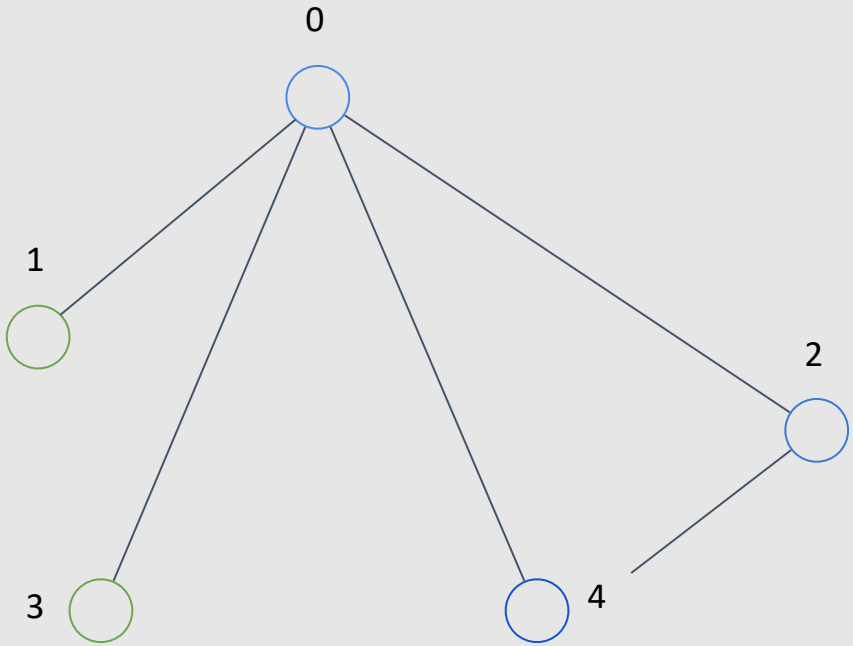

0
1
2
3
4

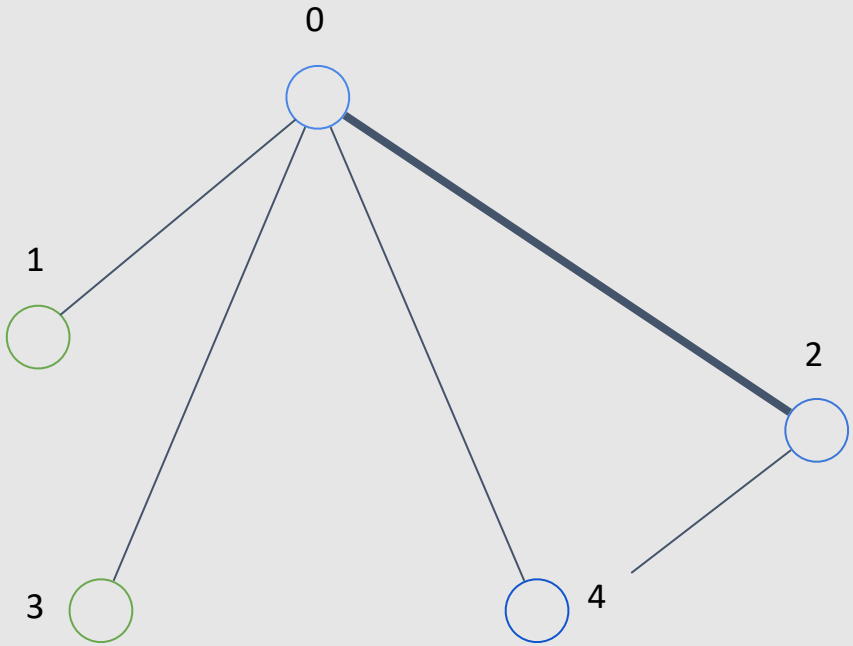

0
1
2
3
4


0
1
2
3
4


0
1
2
3
4

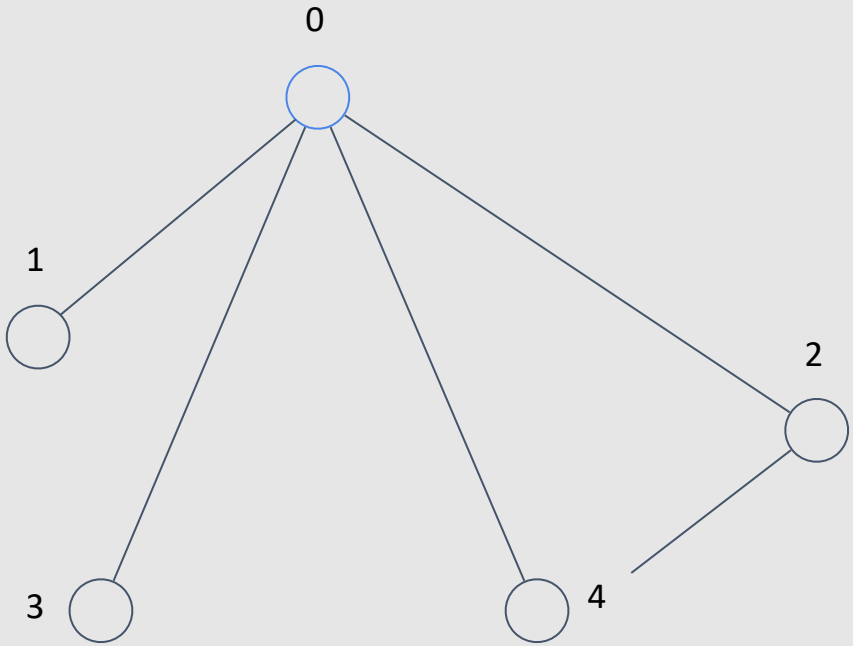

0
1
2
3
4

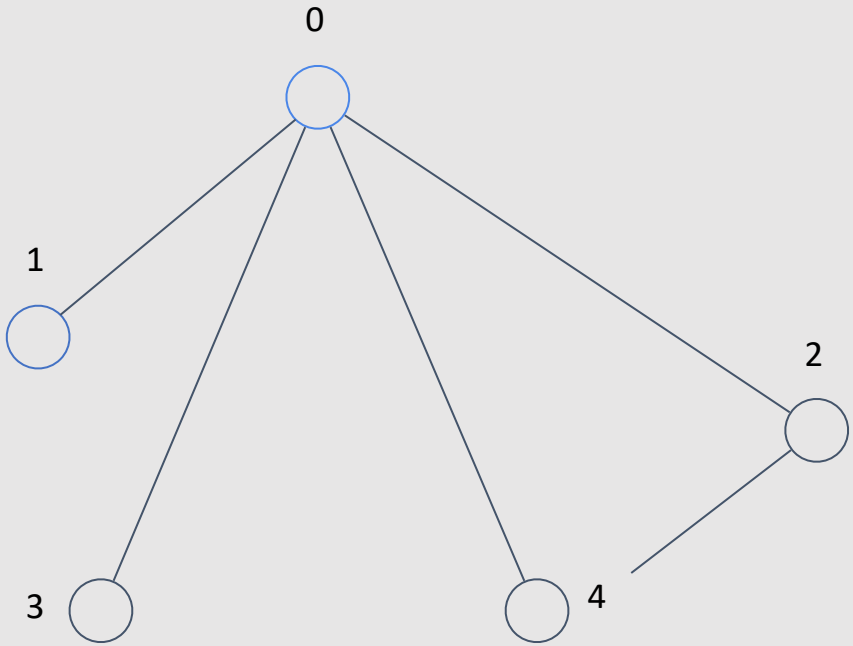

0
1
2
3
4

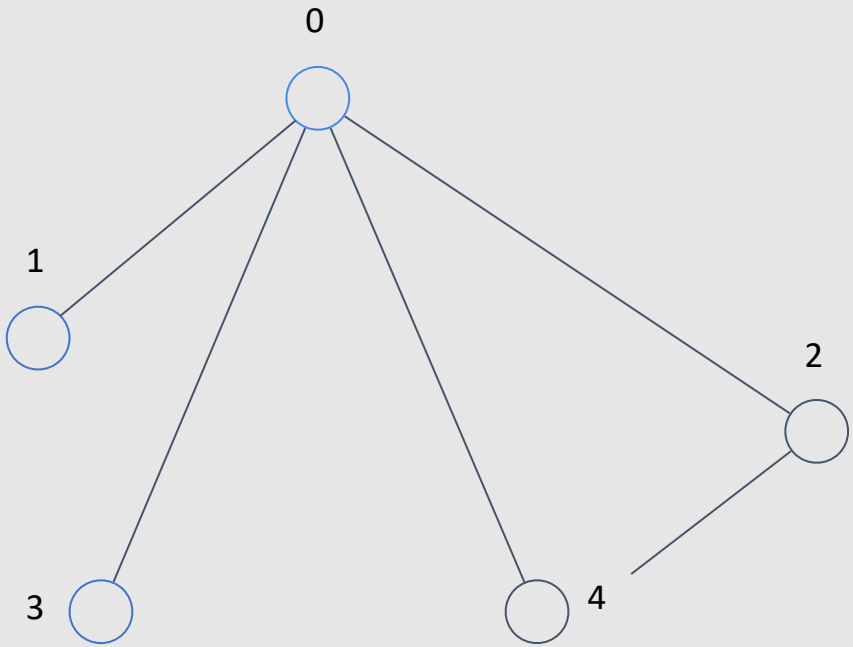

0
1
2
3
4

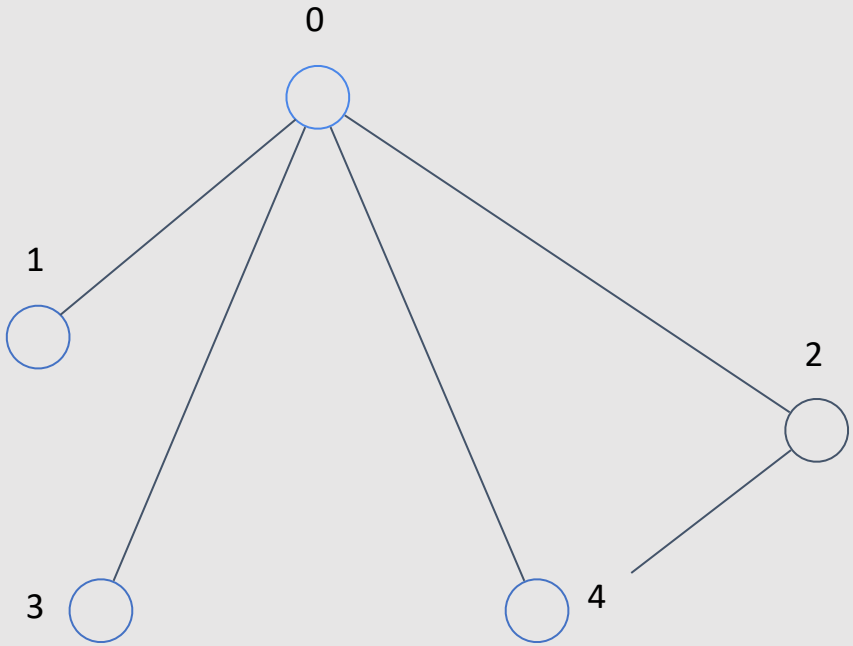
Idea

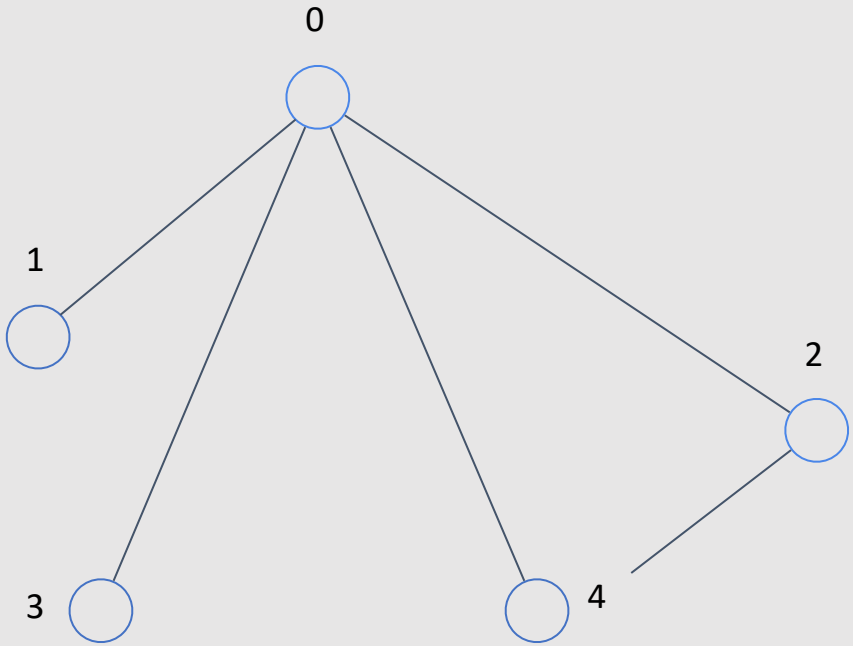
But 2 colors is also enough

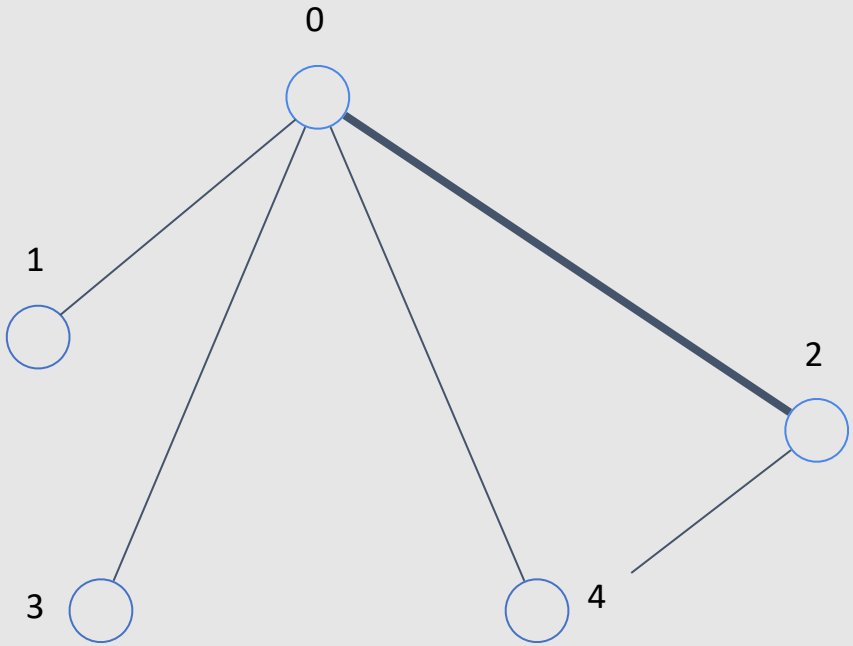

0
1
2
3
4


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4

```cpp
bool dfs(int v, const std::vector<std::vector<int>>& g, std::vector<bool>& visited, int p) {
    visited[v] = true;

    for (auto u: g[v]) {

        if (!visited[u]) {
            if (dfs(u, g, visited, v)) {
                return true;
            }
        }
        else if (u != p) {
            return true;
        }
    }

    return false;
}

bool hasCycle(const std::vector<std::vector<int>>& g) {
    std::vector<bool> visited(g.size(), false);

    for (int i = 0; i < g.size(); i++) {
        if (!visited[i]) {
            if (dfs(i, g, visited, -1)) {
                return true;
            }
        }
    }

    return false;
}
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    cout << hasCycle(graph);

    return 0;
}
```

Success #stdin #stdout 0.01s 5472KB

comments (0)

stdin

copy

```
4 4
1 2
2 3
1 4
4 3
```

stdout

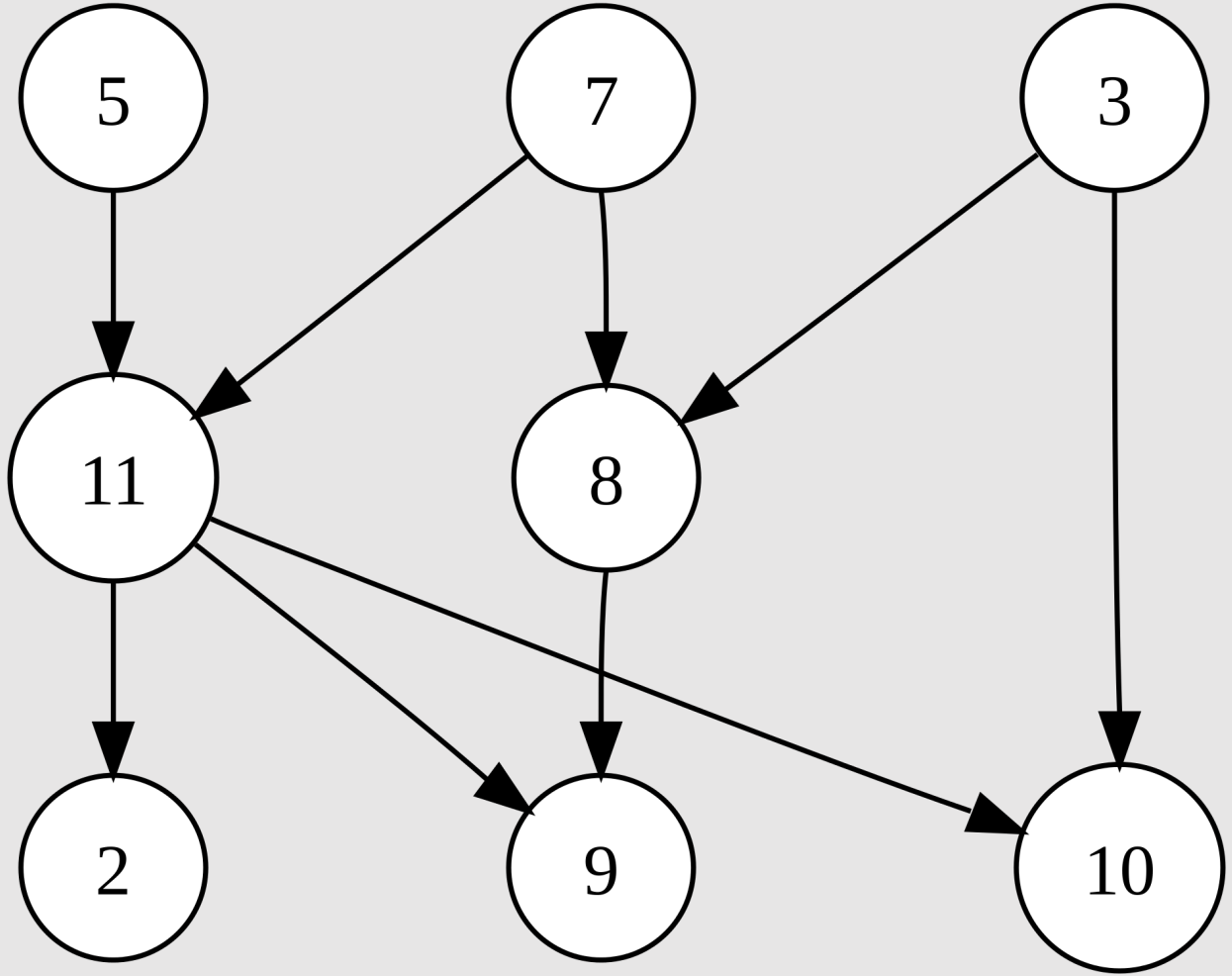
copy

```
1
```

```python
def dfs(g, u, p, visited):
    visited[u] = True

    for v in g[u]:
        if not visited[v]:
            if dfs(g, v, u,visited):
                return True
        elif u != p:
            return True
    return False

def hasCycle(g, n):
    visited = [False] * n;

    for i in range(n):
        if not visited[i]:
            if dfs(g, i, i, visited):
                return True

    return False
```

Topsort

- We have directed graph
- A topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.
- A topological ordering is possible if and only if the graph has no directed cycles(Later we will discuss it).


5
7
3
11
8
2
9
10

Example

Many topsort, for example:

- 5, 7, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 10, 9
- 5, 7, 3, 8, 11, 2, 10, 9
- 7, 5, 11, 3, 10, 8, 9, 2

Idea

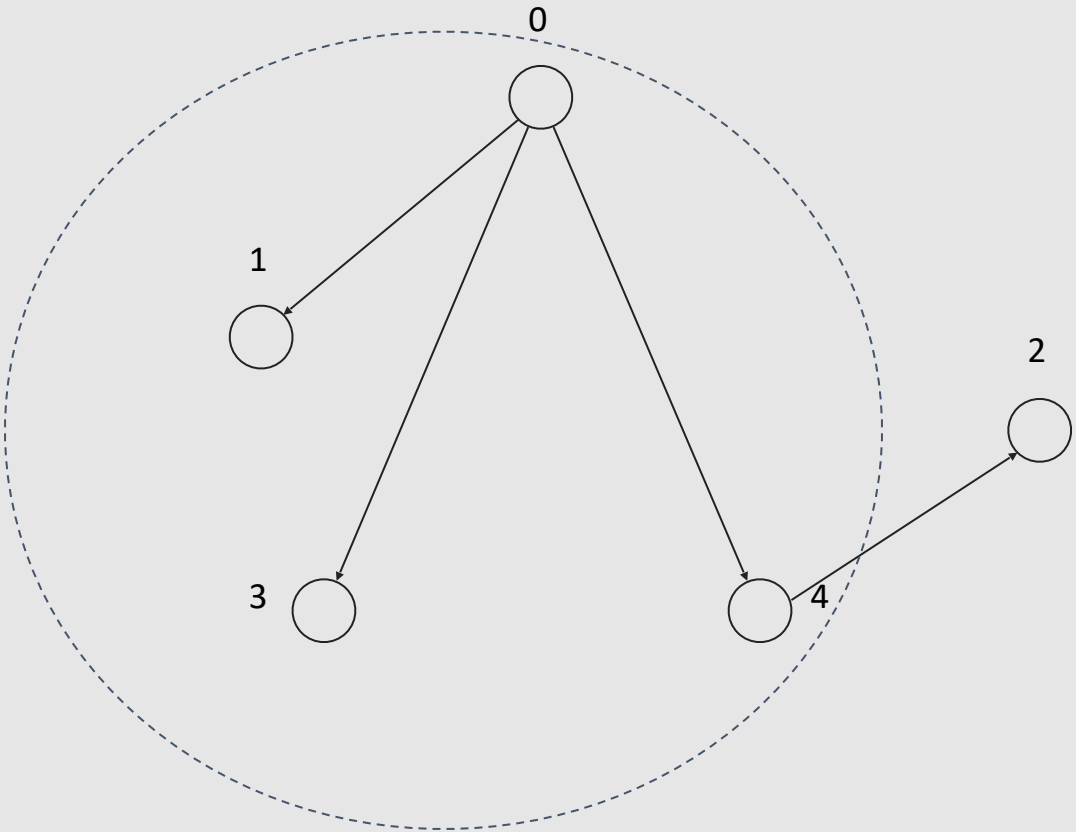
- What does depth-first search do? When launched from some vertex v, it attempts to execute along all edges from v.
- Thus, by the time we exit the dfs(v) call, all vertices reachable from v, whether by a single edge or along a path, will have been visited by the traversal.

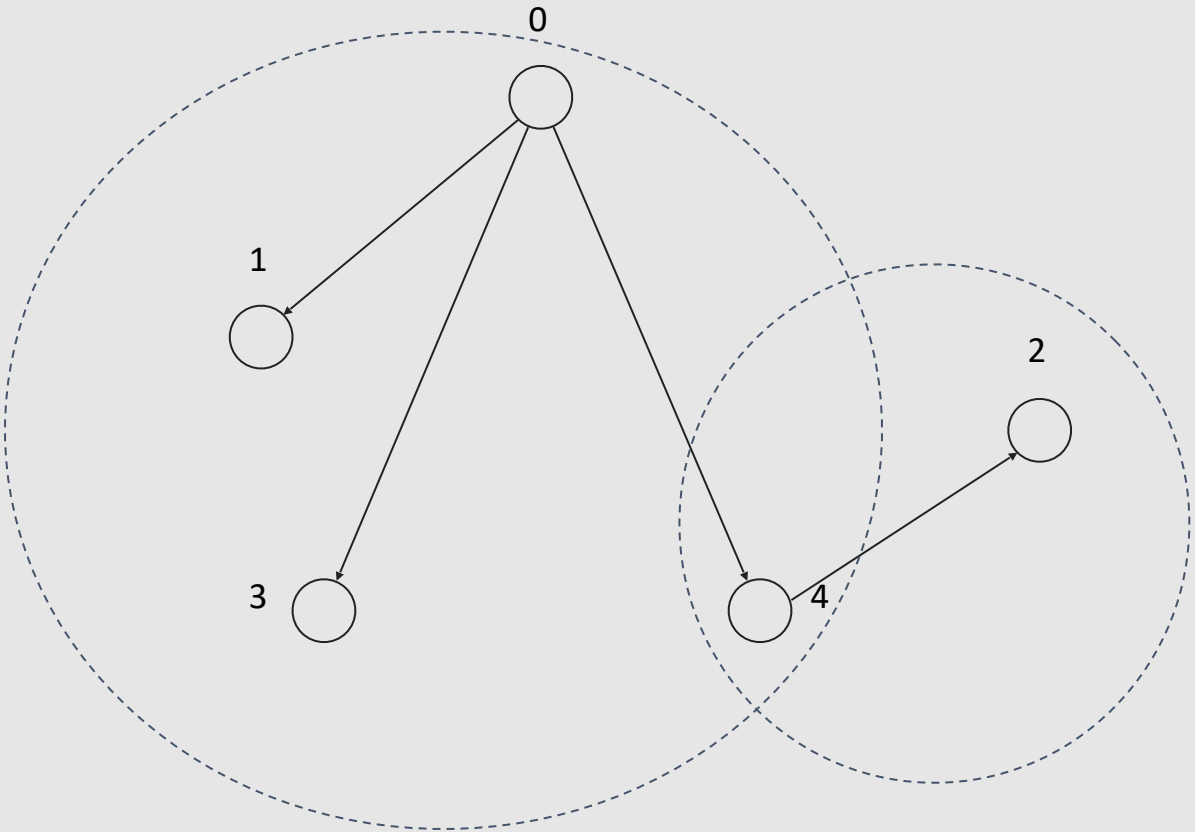
Idea

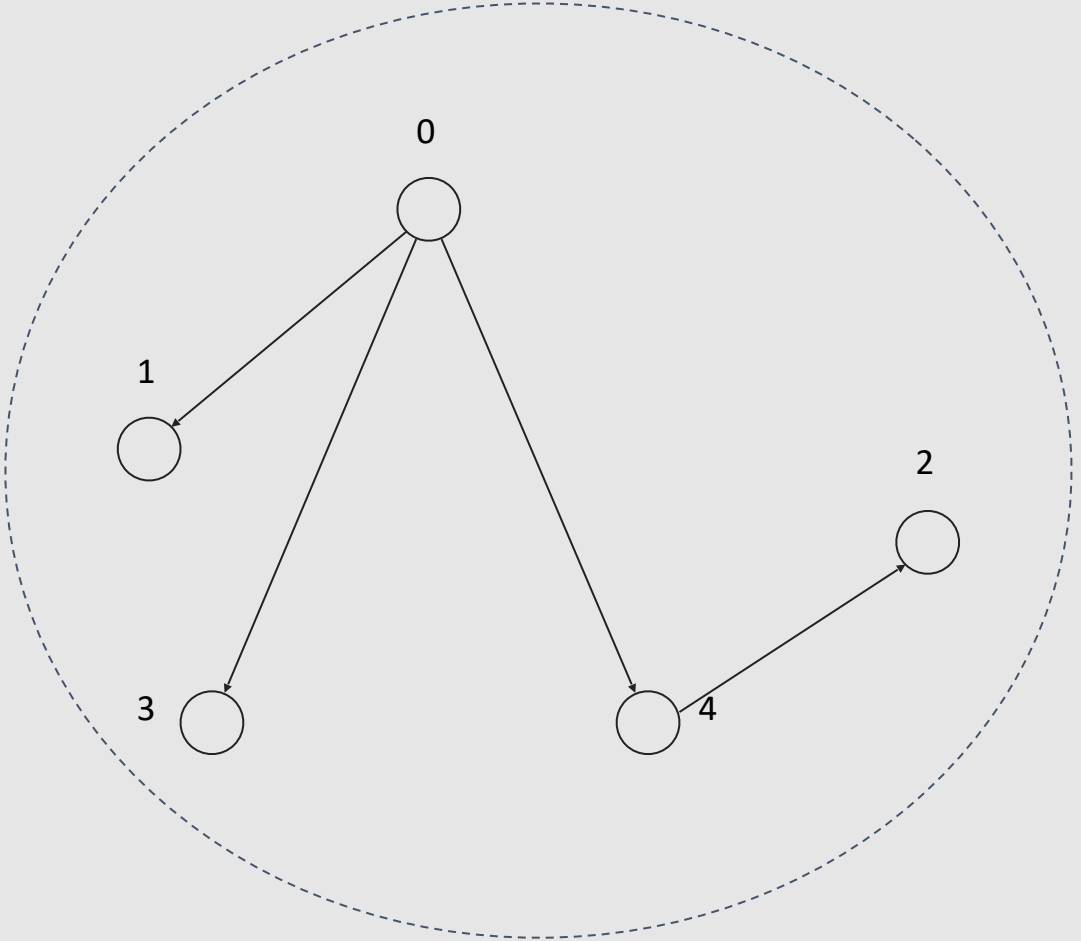
- Single edge - by the dfs algorithm. Since it checks all vertices connected to v within the loop.
- Path - induction by distance from the vertex.

For a distance of one - proven in the previous point.

For a greater distance - we have visited all vertices at distance n, but all the children of the vertices at the maximum distance have already been visited.


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4

Idea

- Therefore, if we add our vertex to the beginning of a certain list when exiting dfs(v), we will eventually get a topological sorting in this list.

```cpp
void dfs(int v, const std::vector<std::vector<int>>& g, std::vector<bool>& visited, int p, vector<int> & topsort) {
    visited[v] = true;

    for (auto u: g[v]) {

        if (!visited[u]) {
            dfs(u, g, visited, v, topsort);
        }
    }
    topsort.push_back(v);
}

void TopSort(const std::vector<std::vector<int>>& g) {
    std::vector<bool> visited(g.size(), false);
    std::vector<int> topsort;

    for (int i = 0; i < g.size(); i++) {
        if (!visited[i]) {
            dfs(i, g, visited, -1, topsort);
        }
    }

    for (auto elem: topsort) {
        cout << elem + 1 << " ";
    }
}
```

stdin copy

```
5 4
1 2
1 4
1 5
2 3
```

stdout copy

```
3 2 4 5 1
```

```python
def dfs(g, u, p, visited, topsort):
    visited[u] = True

    for v in g[u]:
        if not visited[v]:
            dfs(g, v, u, visited, topsort)

    topsort.append(u)

def TopSort(g, n):
    visited = [False] * n;
    topsort = []

    for i in range(n):
        if not visited[i]:
            dfs(g, i, i, visited, topsort)

    print(topsort)
```

```
18.     print(topsort)
19.
20. n, m = map(int, input().split(' '))
21.
22. Matrix = []
23. visited = [False] * n
24. for i in range(n):
25.     Matrix.append([])
26.
27. for i in range(m):
28.     from_, to = map(int, input().split(' '))
29.     Matrix[from_ - 1].append(to - 1)
30.
31. print(TopSort(Matrix, n))
```

Success #stdin #stdout 0.03s 9640KB

stdin

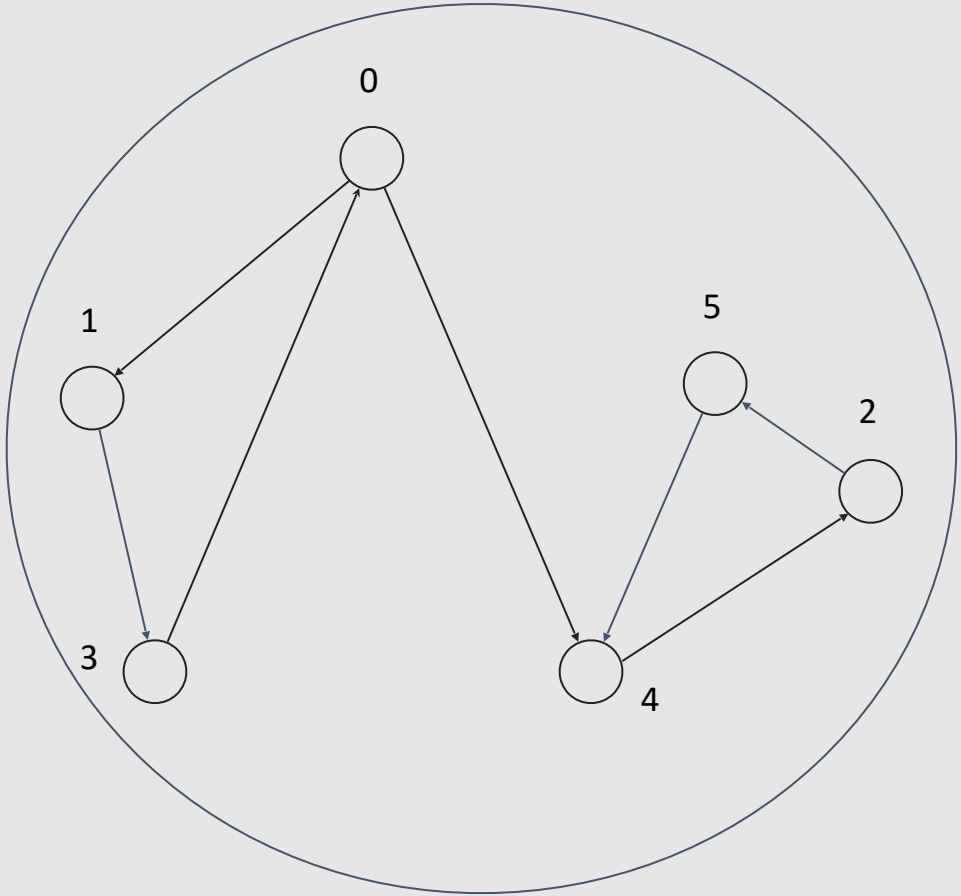
```
5 4
1 2
1 4
1 5
2 3
```

stdout

```
[2, 1, 3, 4, 0]
None
```

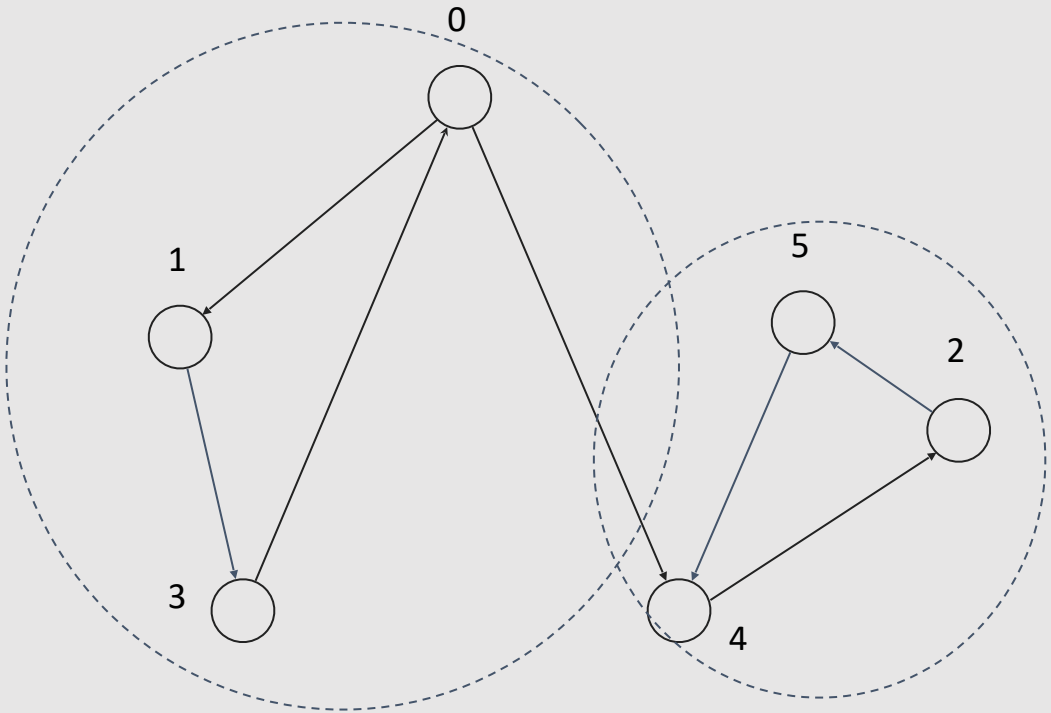
Weakly connected

- A weakly connected component is the maximum set of vertices of a directed graph, between any two of which there exists a path along the arcs regardless of direction.


0
1
5
2
3
4

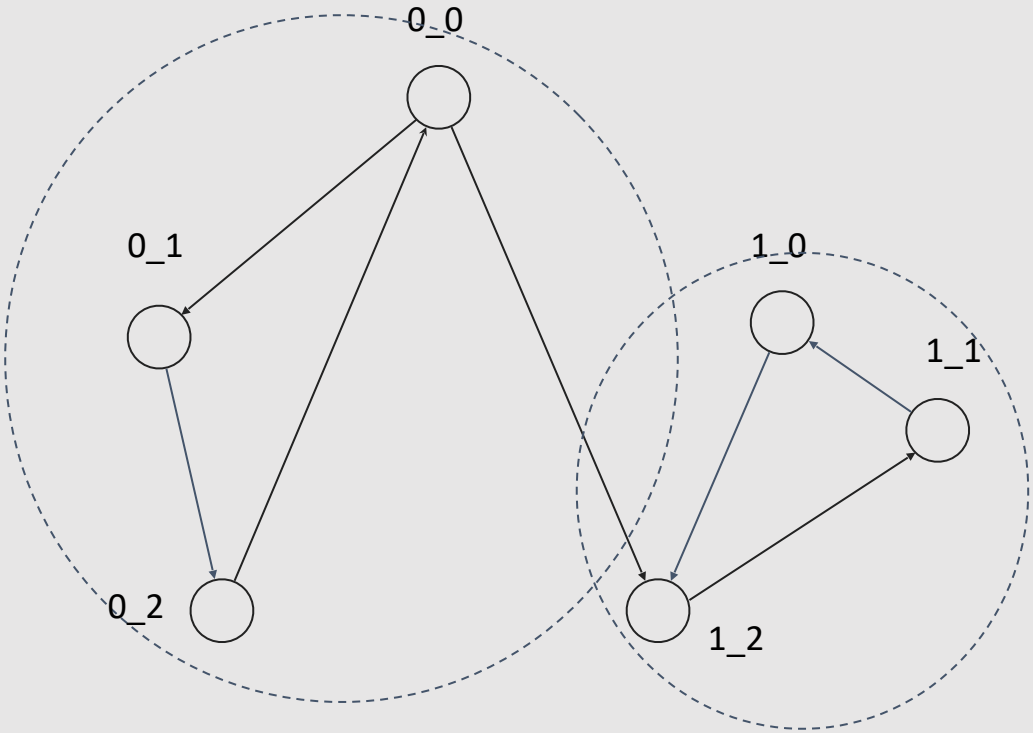
Strongly connected

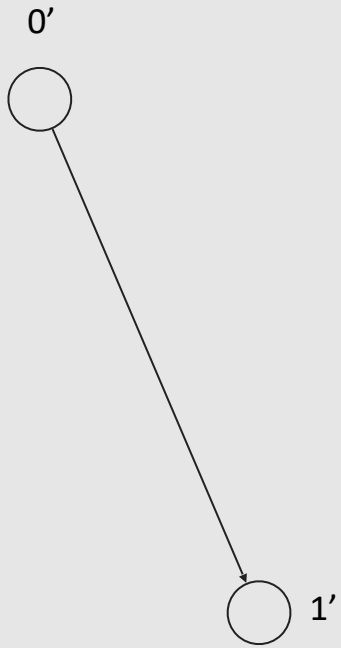
- The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.


0
1
3
5
2
4

Condensation

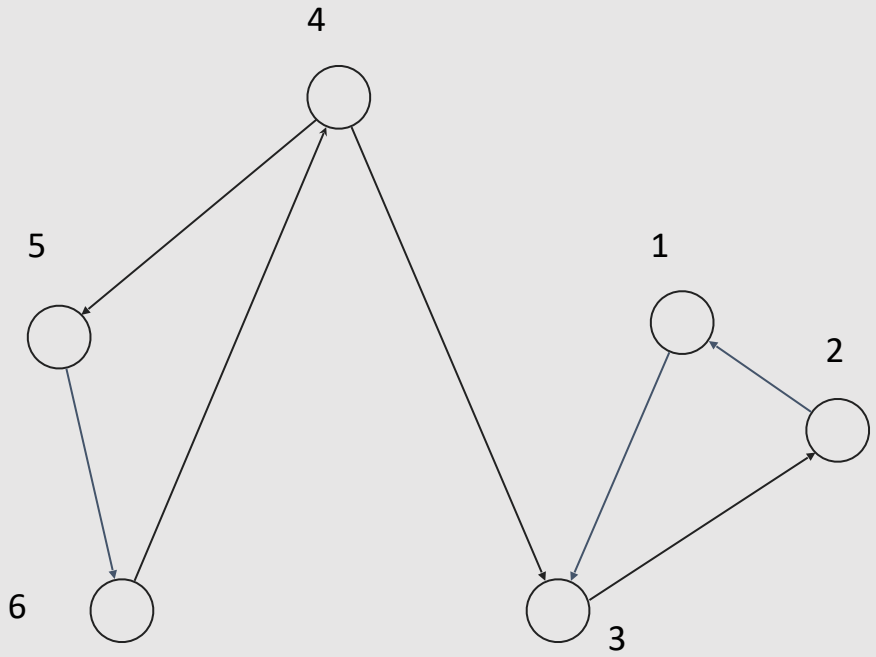
- Let make one vertex from each strongly connected component.


0_0
0_1
0_2
1_0
1_1
1_2


0’
1’

Idea

- if we have a->b and b->a, then (a, b) are in one SCC.
- Topsort can give us an order that fits the first property.
- How do we check the second one?

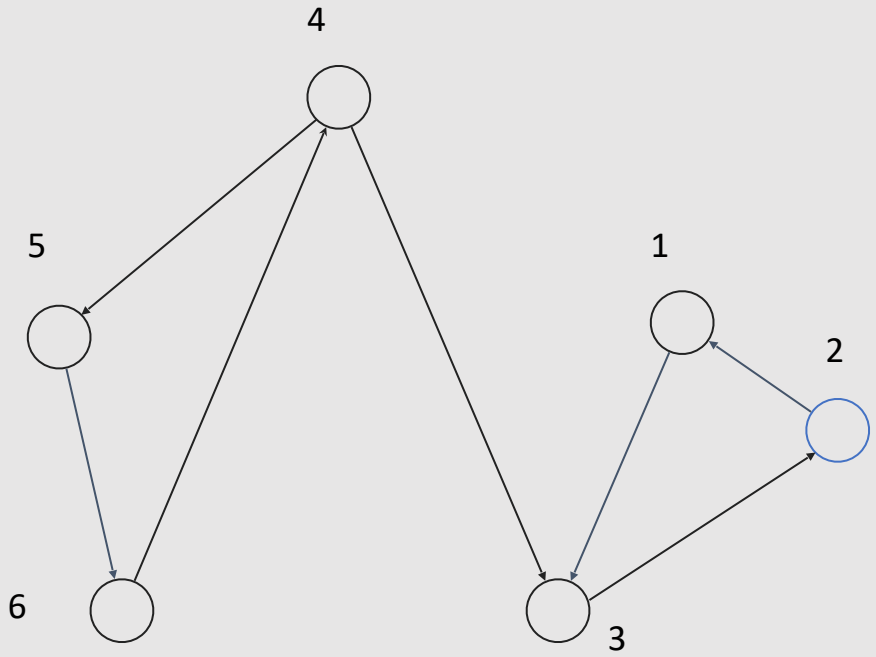

4
5
1
2
6
3

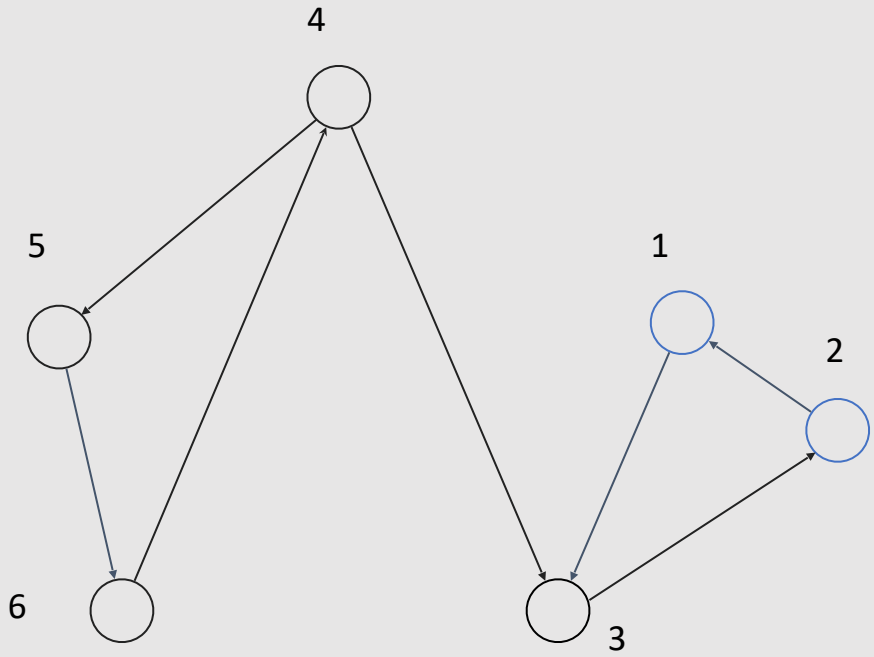
Topsort

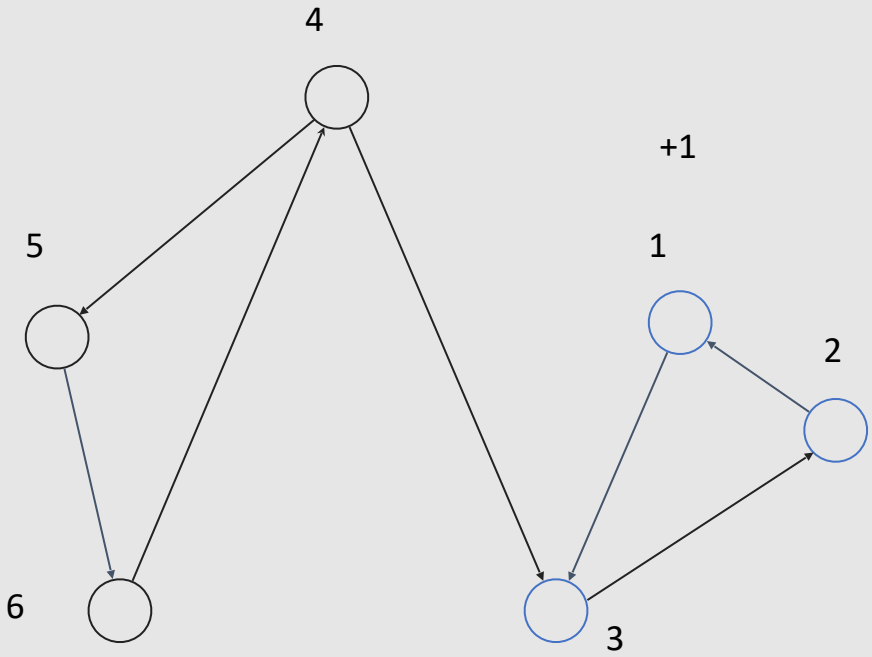
- Topsort can be [2, 3, 1, 6, 5, 4] for example
- If exists edge (C_i, C_j), then all vertices from C_j before C_i
- C_1 = [2, 3, 1] before C_0 = [6, 5, 4], exists (C_0, C_1), because exists (4, 3)

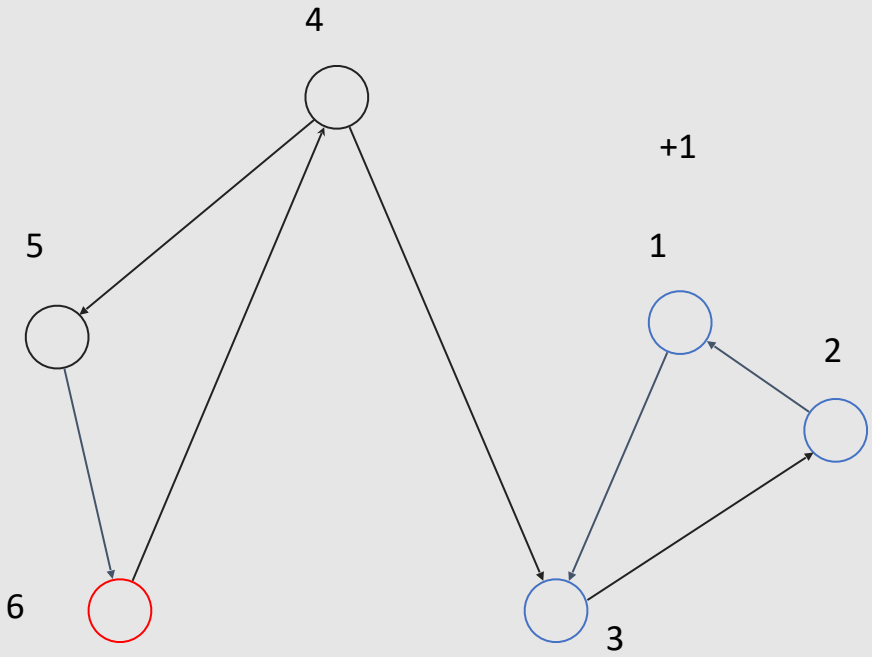
Topsort

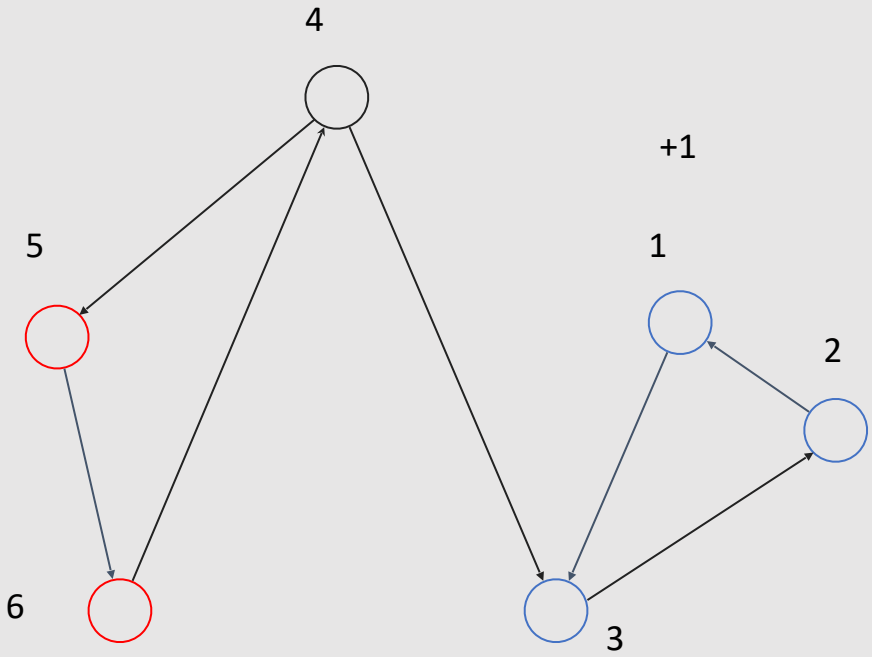
- We can go by topsort order and call dfs from vertices in this order

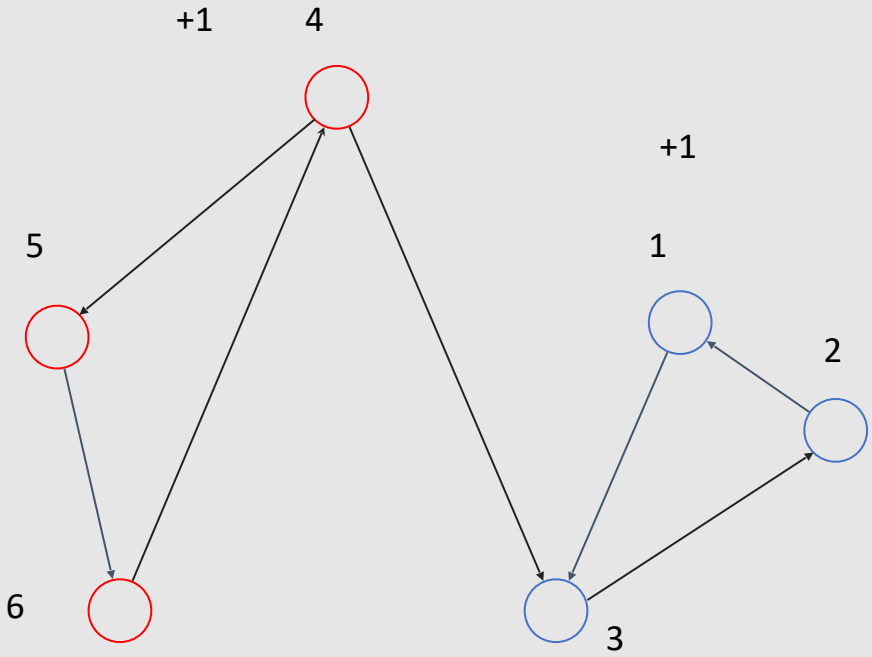

4
5
1
2
6
3


4
5
1
2
6
3


4
+1
5
1
2
6
3

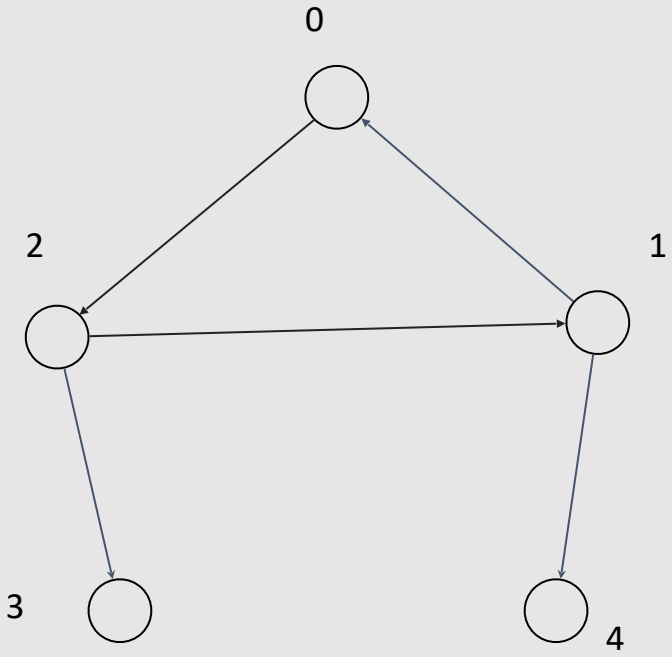

4
+1
5
1
2
6
3


4
+1
5
1
2
6
3


+1
4
+1
5
1
2
6
3

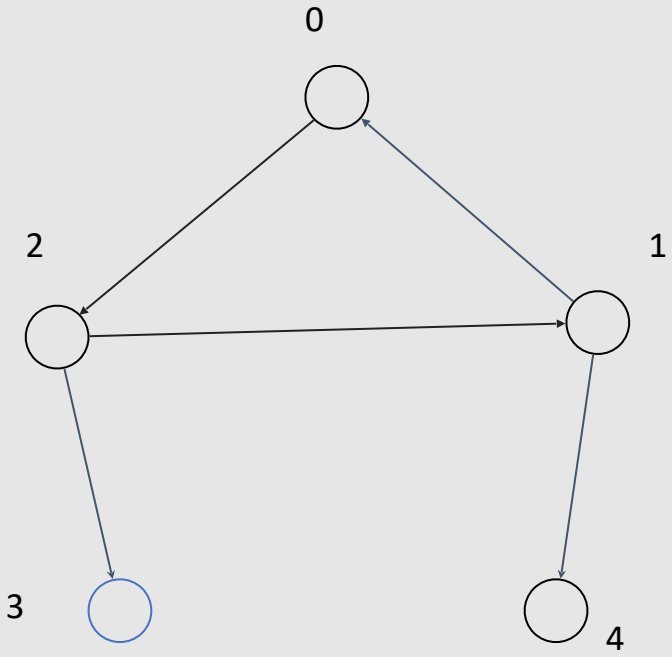
No

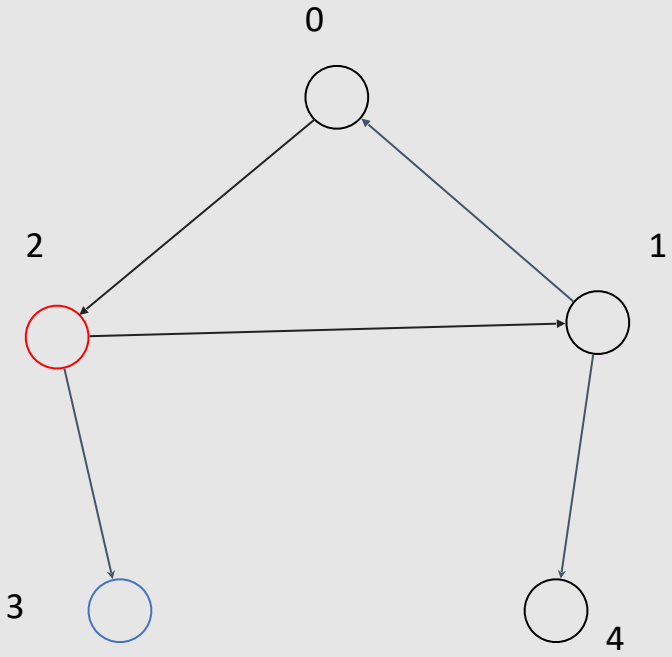
- I lied

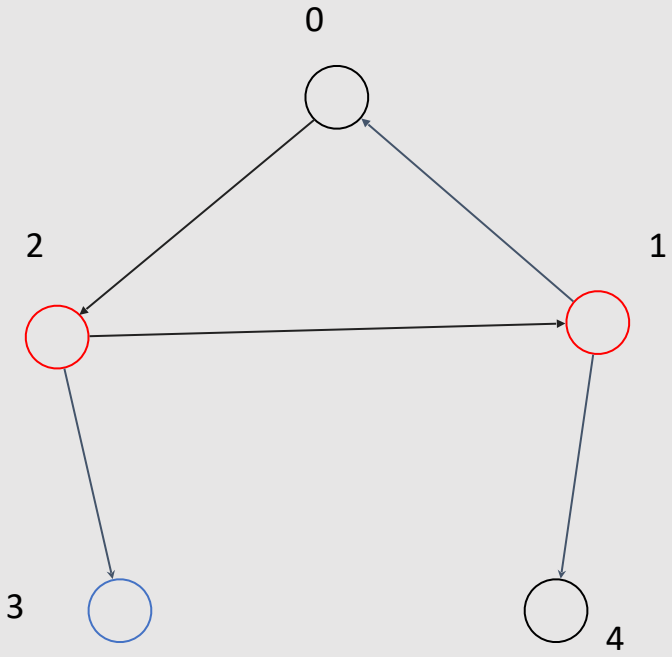

0
2
1
3
4

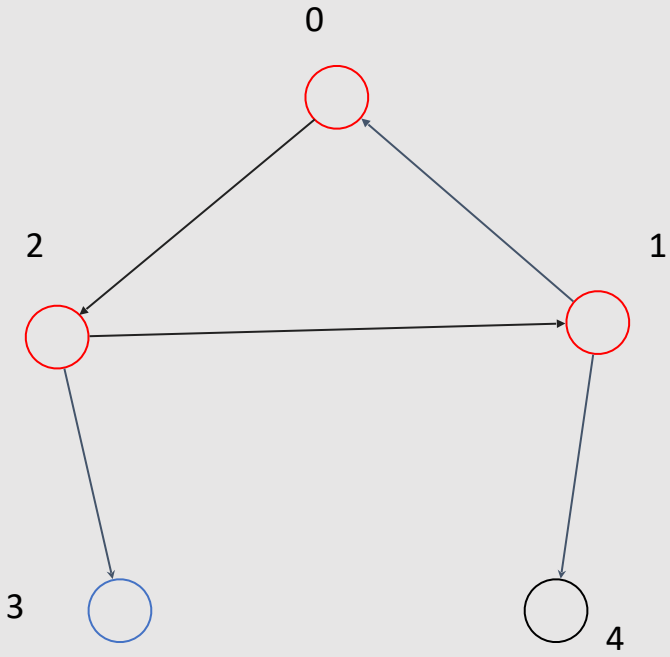
Topsort

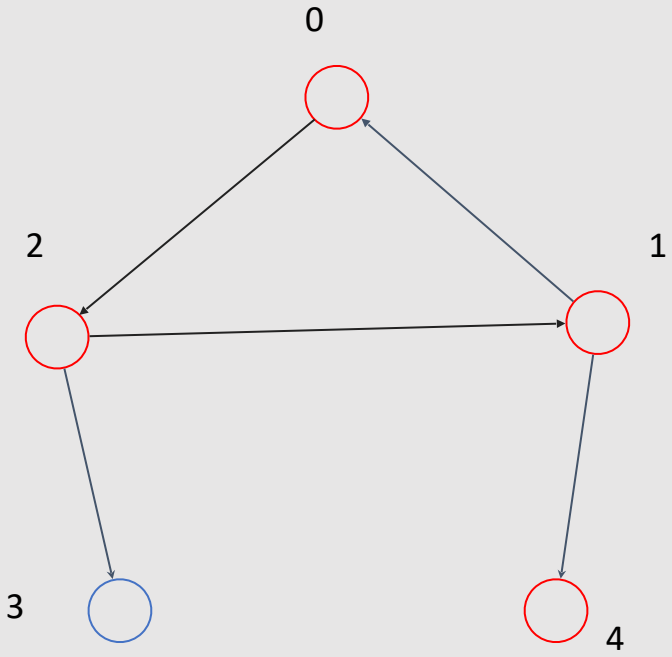
- Topsort can be [3, 2, 4, 1, 0] for example
- OOOPS


0
2
1
3
4


0
2
1
3
4

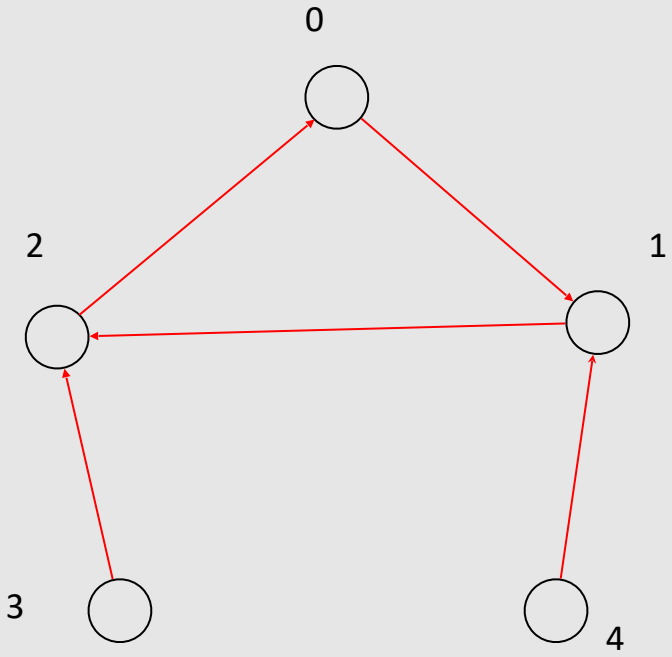

0
2
1
3
4


0
2
1
3
4


0
2
1
3
4

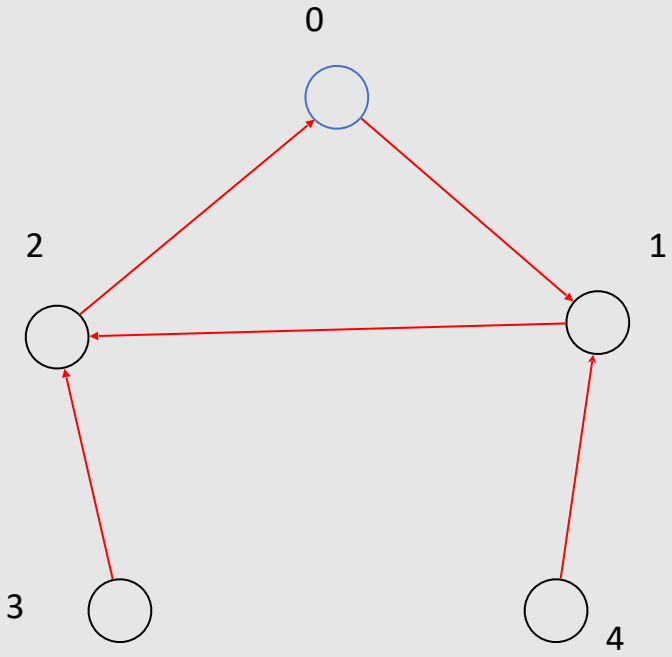
Topsort

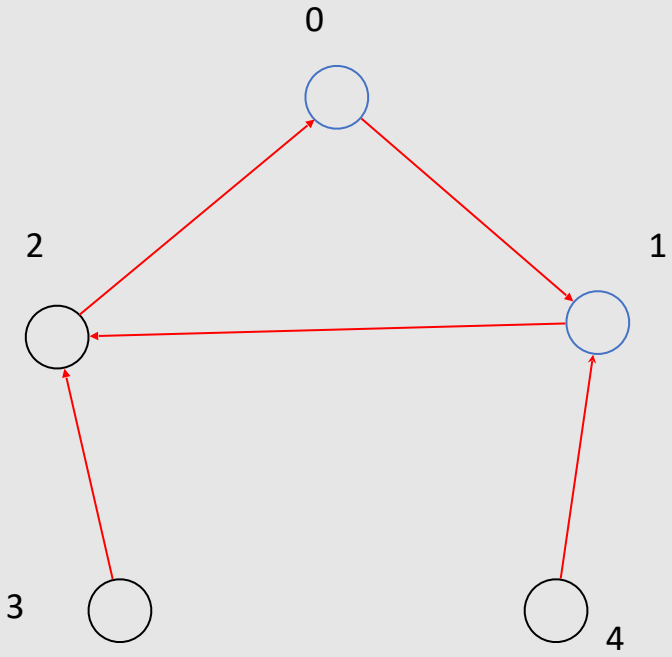
- Let’s go by reversed topsort order
- And let’s go by reversed graph

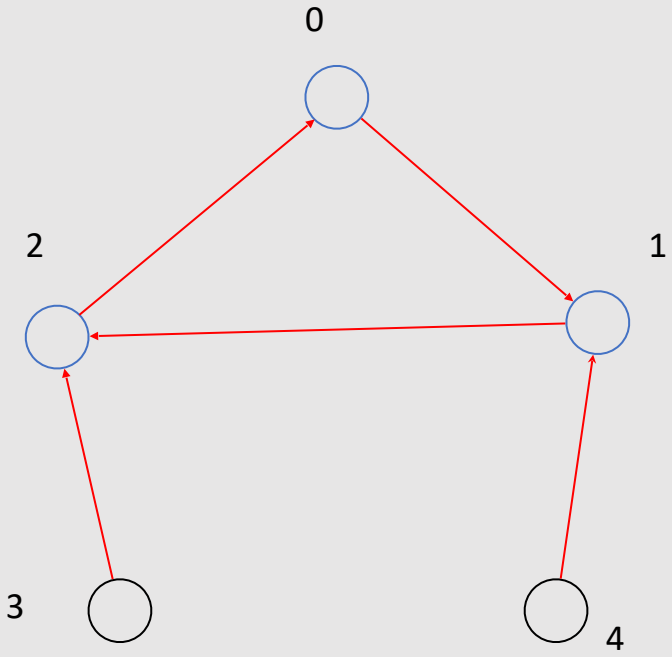

0
2
1
3
4

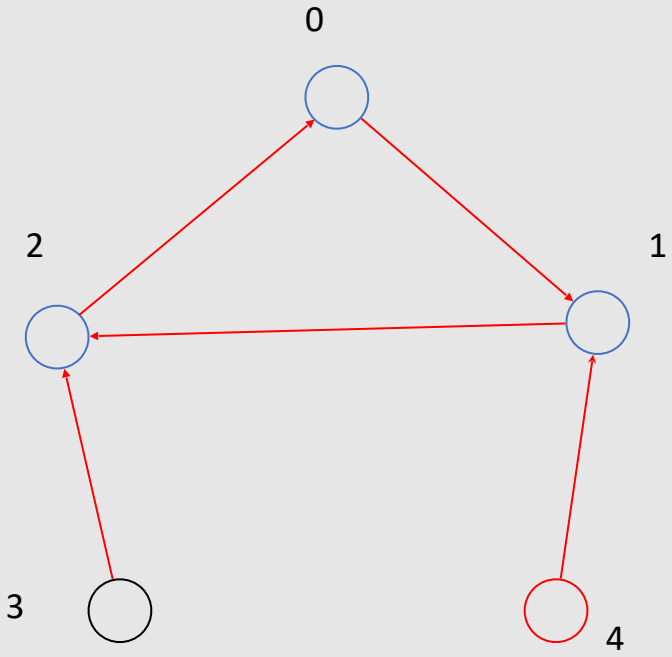
Topsort

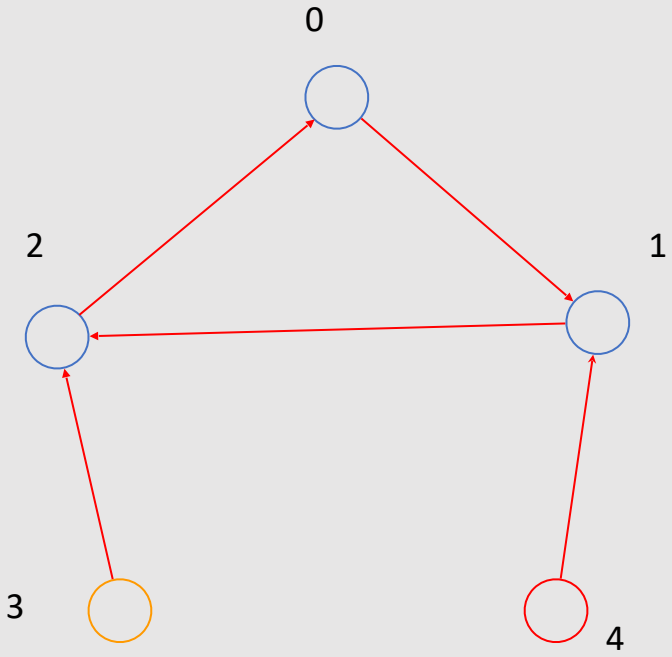
- Reversed topsort can be [0, 1, 4, 2, 3] for example

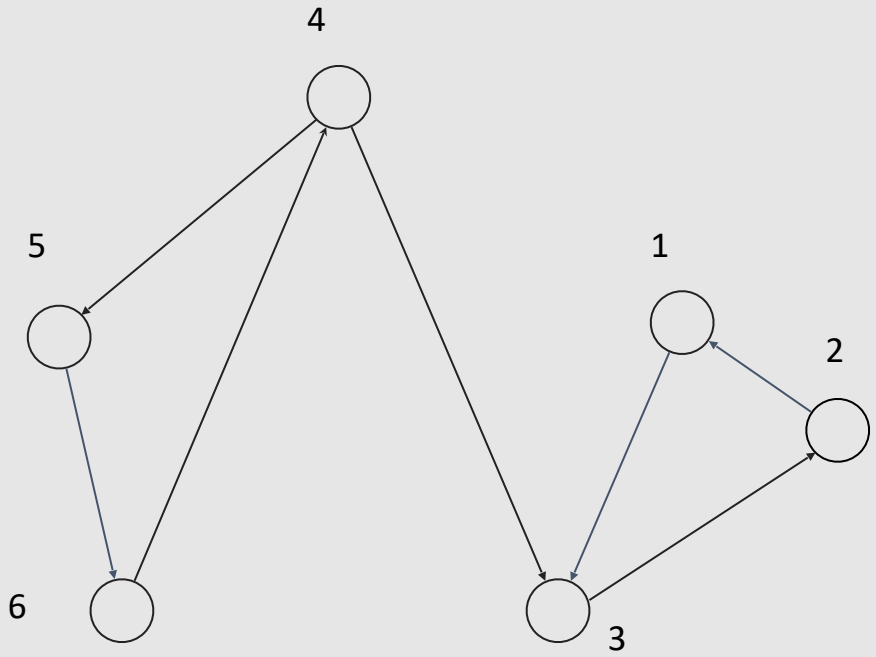

0
2
1
3
4


0
2
1
3
4


0
2
1
3
4

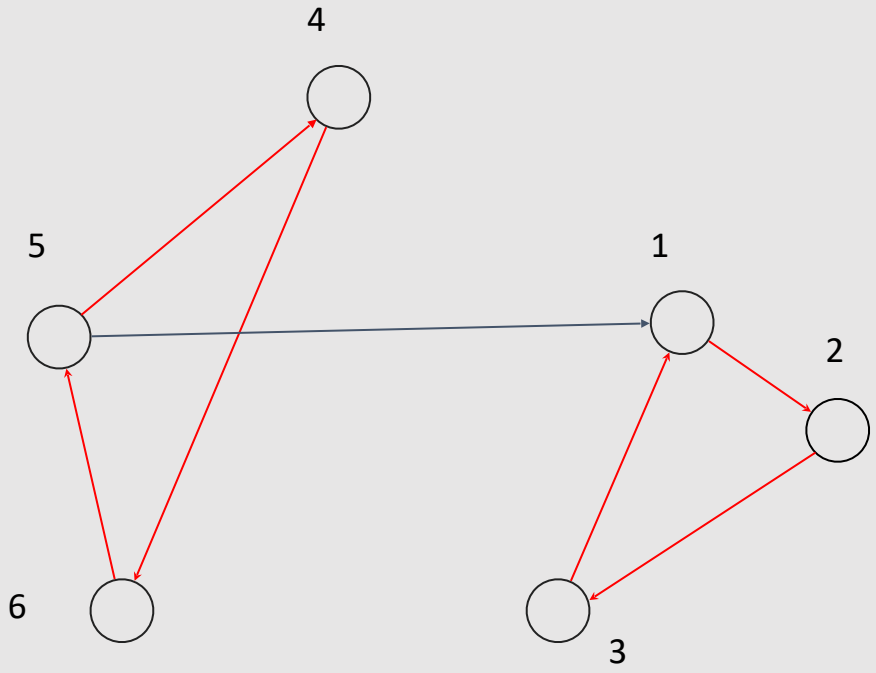

0
2
1
3
4


0
2
1
3
4


4
5
1
2
6
3

Topsort

- Topsort can be [2, 3, 1, 6, 5, 4]
- Reversed can be [4, 5, 6, 1, 3 ,2]

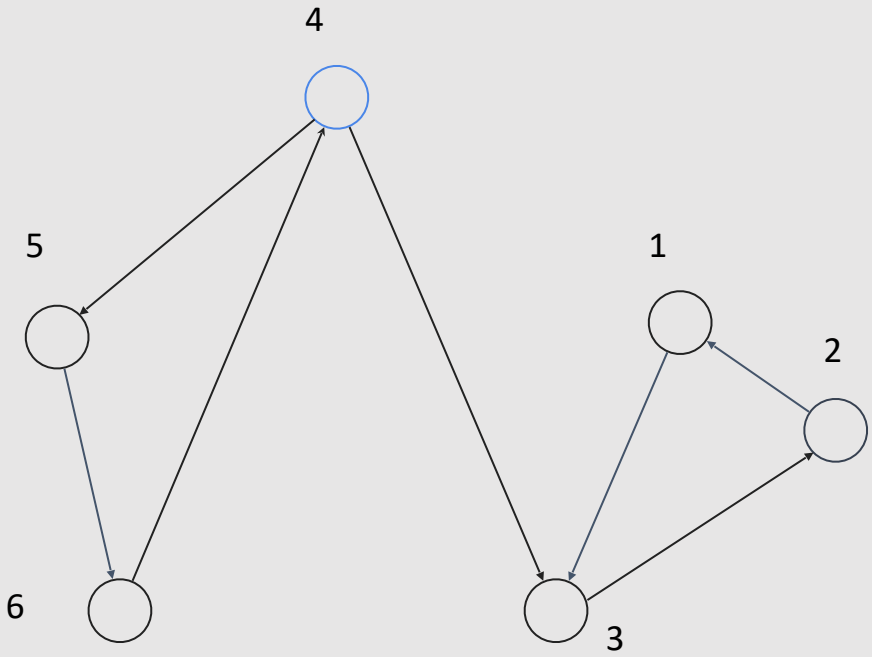

4
5
1
2
6
3

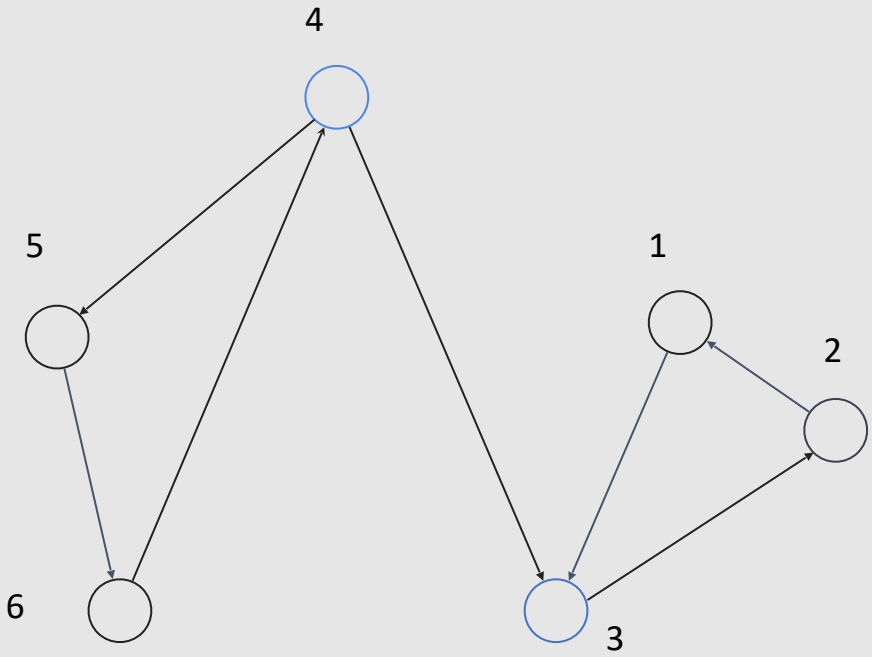
Proof

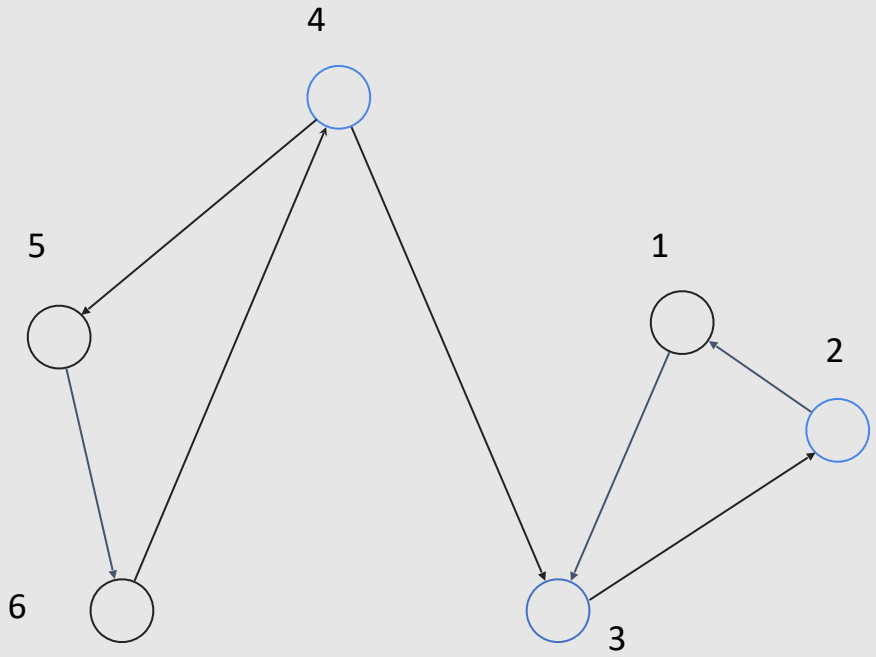
- tout[X] = max(tout[v]) for all v in X
- Let C and C' be two distinct strongly connected components, and let there be an edge (C,C') in the condensed graph. Then tout[C] > tout[C'].
- When proving, two fundamentally different cases arise depending on which component the depth-first search enters first, depending on the relationship between tin[C] and tin[C'].

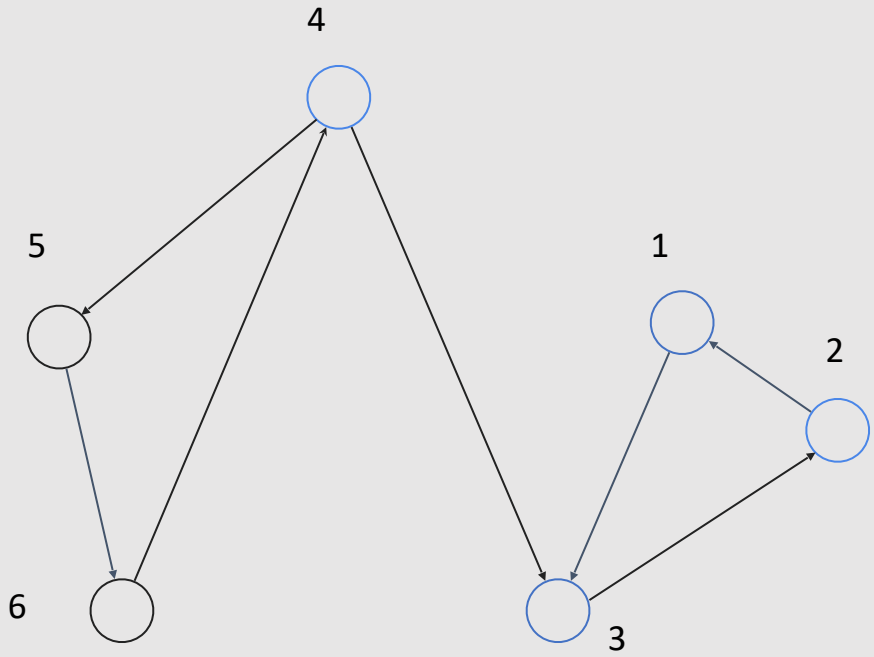
Proof

- The component C was reached first.
- We enter a certain vertex v of component C, while all other vertices of components C and C' are not yet visited.
- However, since there is an edge (C,C') in the condensed graph by condition, from vertex v not only the entire component C will be reachable, but also the entire component C'.
- This implies that when launched from vertex v, the depth-first search will go through all the vertices of components C and C'.


4
5
1
2
6
3

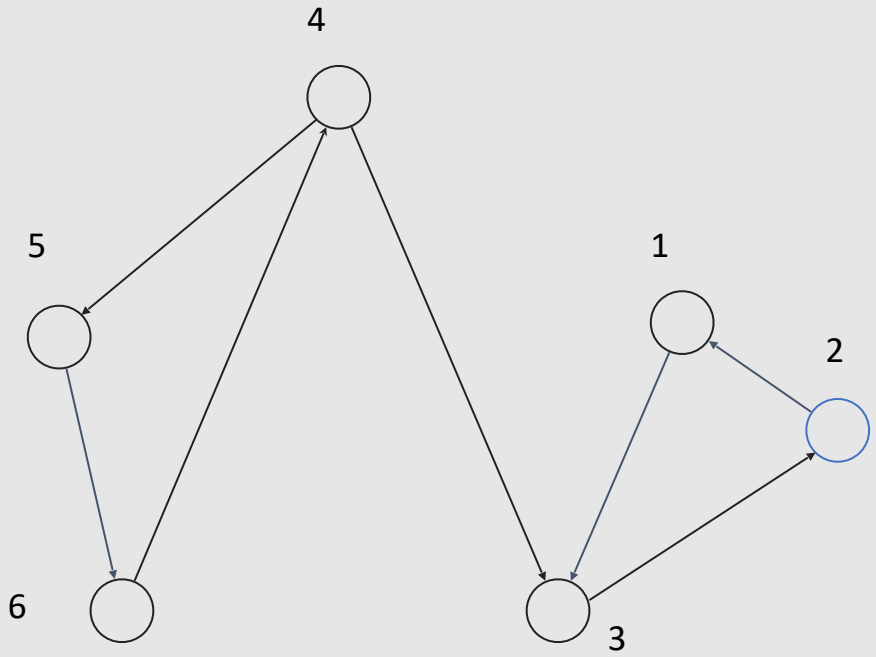

4
5
1
2
6
3

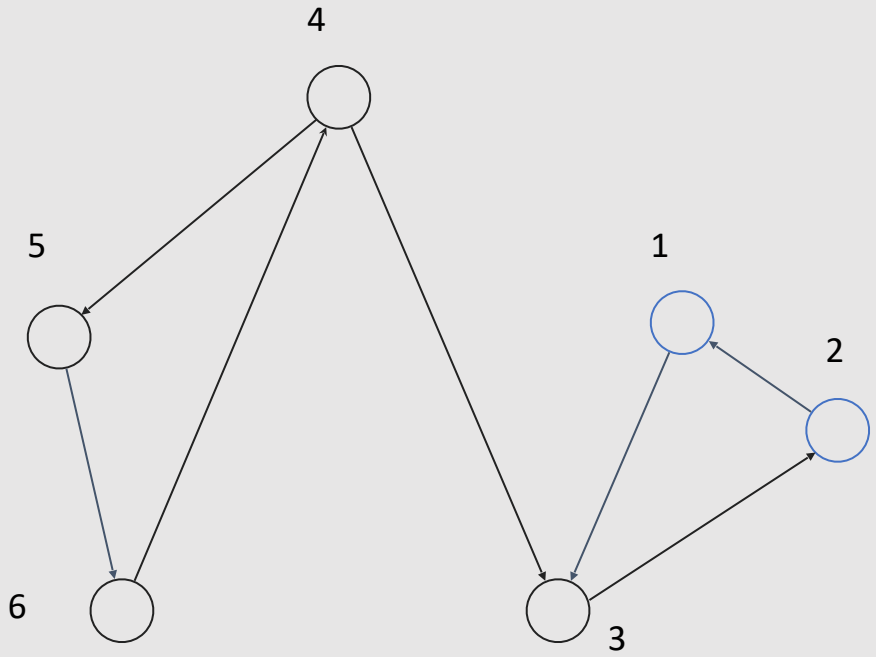

4
5
1
2
6
3

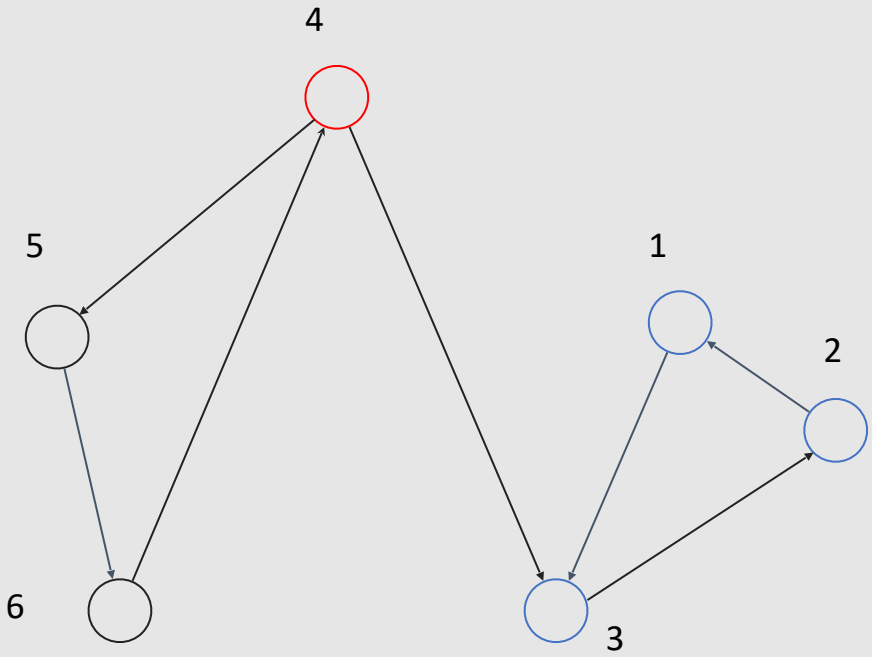

4
5
1
2
6
3

Proof

- The component C' was reached first.
- Similarly, the depth-first search enters a certain vertex v in C', and all other vertices of components C and C' are not yet visited.
- Since there was an edge (C, C') in the condensed graph by condition, due to the acyclic nature of the condensed graph, there is no reverse path,
- This means they will be visited by the depth-first search later, which also proves the claim.


4
5
1
2
6
3


4
5
1
2
6
3


4
5
1
2
6
3

Proof

- If we go in reverse order we will firstly check the first component and so on.
- And because of transposed graph we will find the whole component(they are cyclic) and not more(no bidirect edges between two components)

```python
def dfs_order(v, visited, g, order):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs_order(u, visited, g, order)
    order.append(v)

def dfs_scc(v, visited, rg, current_scc):
    visited[v] = True
    current_scc.append(v)
    for u in rg[v]:
        if not visited[u]:
            dfs_scc(u, visited, rg, current_scc)

def find_sccs(g):
    n = len(g)
    order, visited = [], [False] * n

    for i in range(n):
        if not visited[i]:
            dfs_order(i, visited, g, order)
```

```
    rg = [[] for _ in range(n)]
    for v in range(n):
        for u in g[v]:
            rg[u].append(v)

    visited = [False] * n
    sccs = []
    for v in reversed(order):
        if not visited[v]:
            current_scc = []
            dfs_scc(v, visited, rg, current_scc)
            sccs.append(current_scc)

    return sccs

n, m = map(int, input().split())
graph = [[] for _ in range(n)]
for _ in range(m):
    from_, to_ = map(int, input().split())
    graph[from_ - 1].append(to_ - 1)

print(find_sccs(graph))
```

Success #stdin #stdout 0.04s 9728KB comm

stdin

```
5 5
1 2
2 3
3 1
2 5
3 4
```

stdout

```
[[0, 2, 1], [4], [3]]
```

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs_order(int v, vector<bool>& visited, const vector<vector<int>>& g, vector<int>& order) {
    visited[v] = true;
    for (int u : g[v]) {
        if (!visited[u]) {
            dfs_order(u, visited, g, order);
        }
    }
    order.push_back(v);
}

void dfs_scc(int v, vector<bool>& visited, const vector<vector<int>>& rg, vector<int>& current_scc) {
    visited[v] = true;
    current_scc.push_back(v);
    for (int u : rg[v]) {
        if (!visited[u]) {
            dfs_scc(u, visited, rg, current_scc);
        }
    }
}

vector<vector<int>> find_sccs(const vector<vector<int>>& g) {
    int n = g.size();
    vector<int> order;
    vector<bool> visited(n, false);

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs_order(i, visited, g, order);
        }
    }
```

```
    vector<vector<int>> rg(n);
    for (int v = 0; v < n; v++) {
        for (int u : g[v]) {
            rg[u].push_back(v);
        }
    }

    visited.assign(n, false);
    vector<vector<int>> sccs;

    for (int i = n-1; i >= 0; i--) {
        int v = order[i];
        if (!visited[v]) {
            vector<int> current_scc;
            dfs_scc(v, visited, rg, current_scc);
            sccs.push_back(current_scc);
        }
    }

    return sccs;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> graph(n);
```

```
    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }

    vector<vector<int>> sccs = find_sccs(graph);
    for (const auto& scc : sccs) {
        for (int v : scc) {
            cout << v+1 << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5392KB

comments (0)

stdin

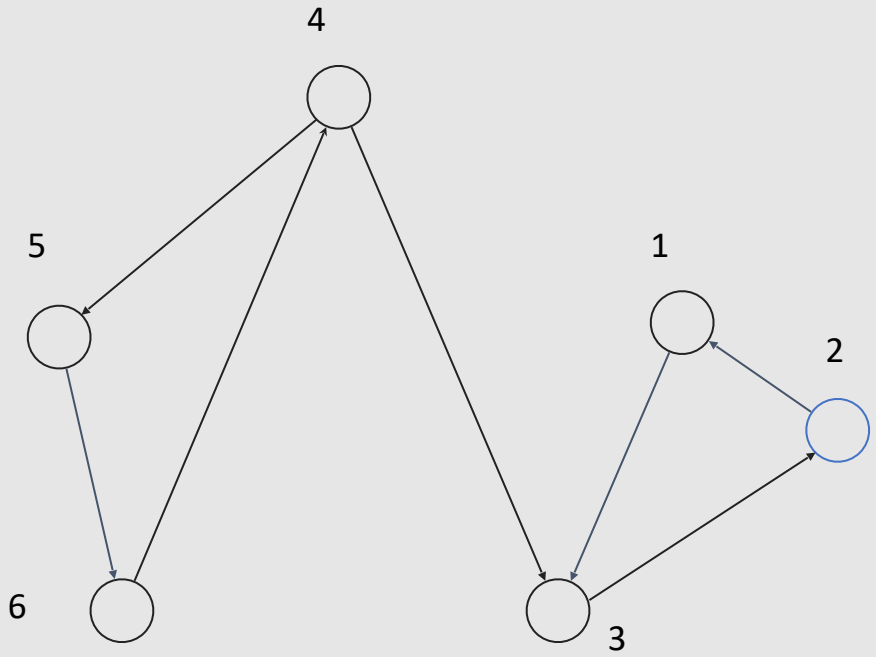
copy

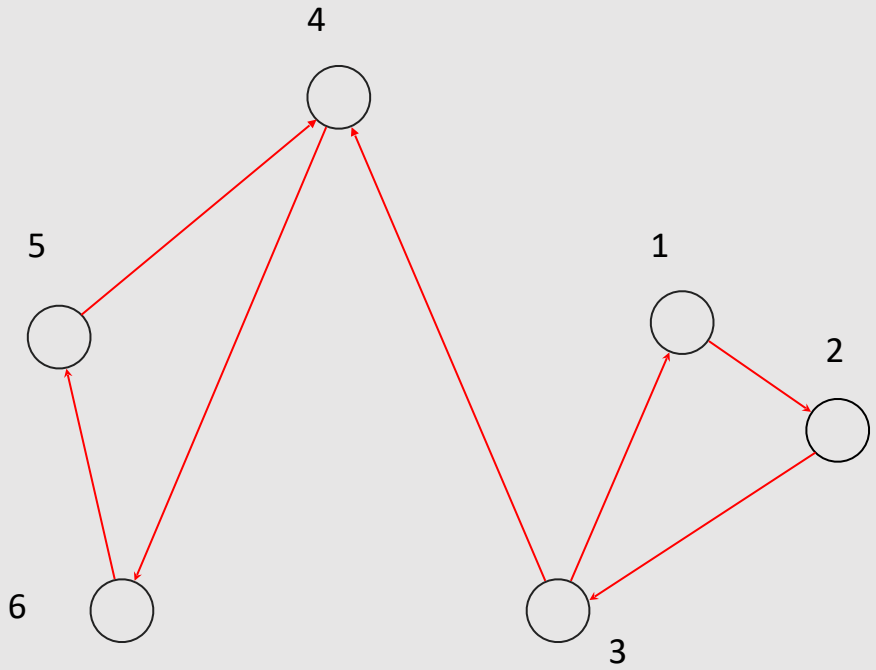
```
5 5
1 2
2 3
3 1
2 5
3 4
```

stdout

copy

```
1 3 2
5
4
```


4
5
1
2
6
3


4
5
1
2
6
3

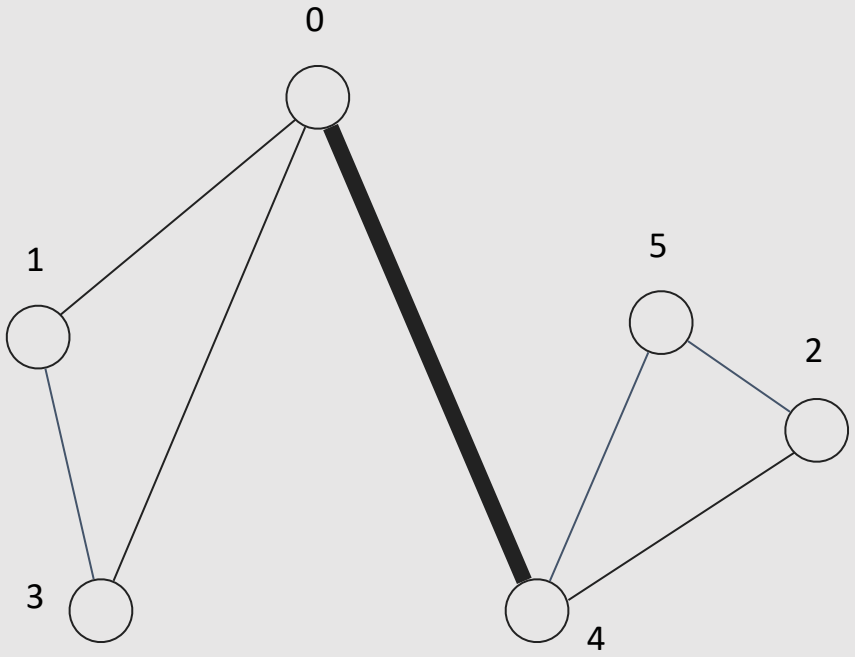
Topsort

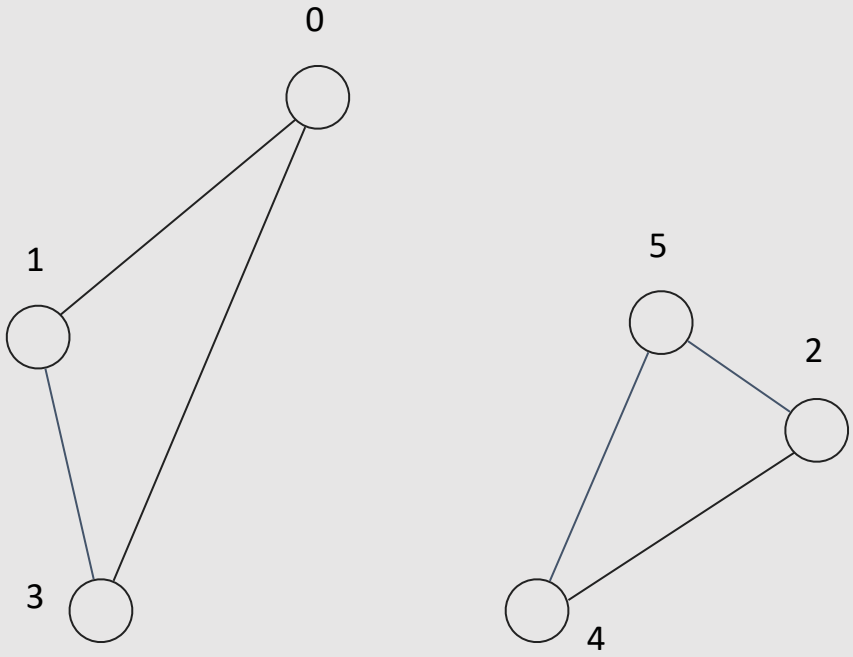
- Topsort can be [2, 3, 1, 6, 5, 4]
- Reversed can be [4, 5, 6, 1, 3, 2]

Bridges and articulation points

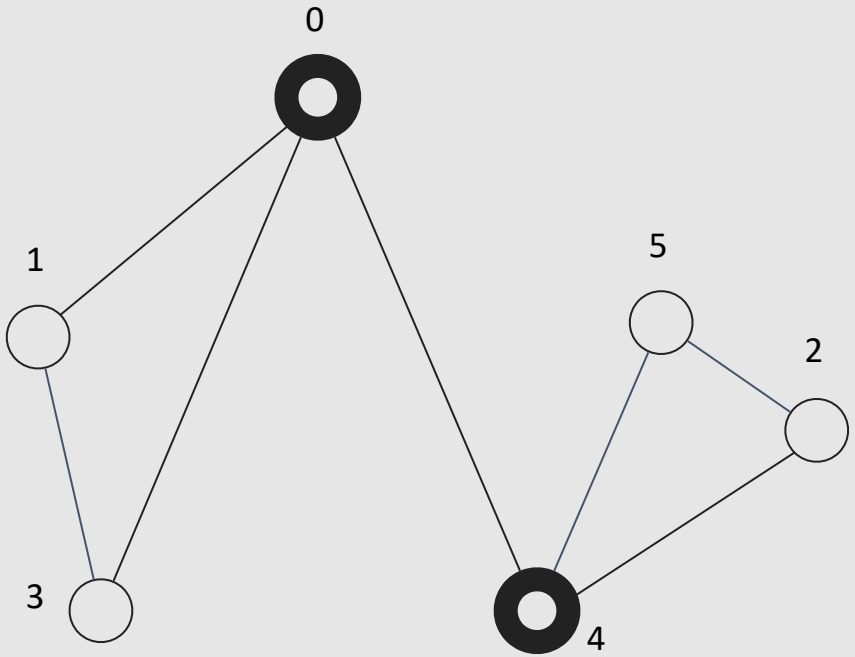
- Definition. A bridge is an edge that, when removed, makes a connected undirected graph disconnected.
- Definition. An articulation point (or cut vertex) is a vertex that, when removed, makes a connected undirected graph disconnected.

bridge

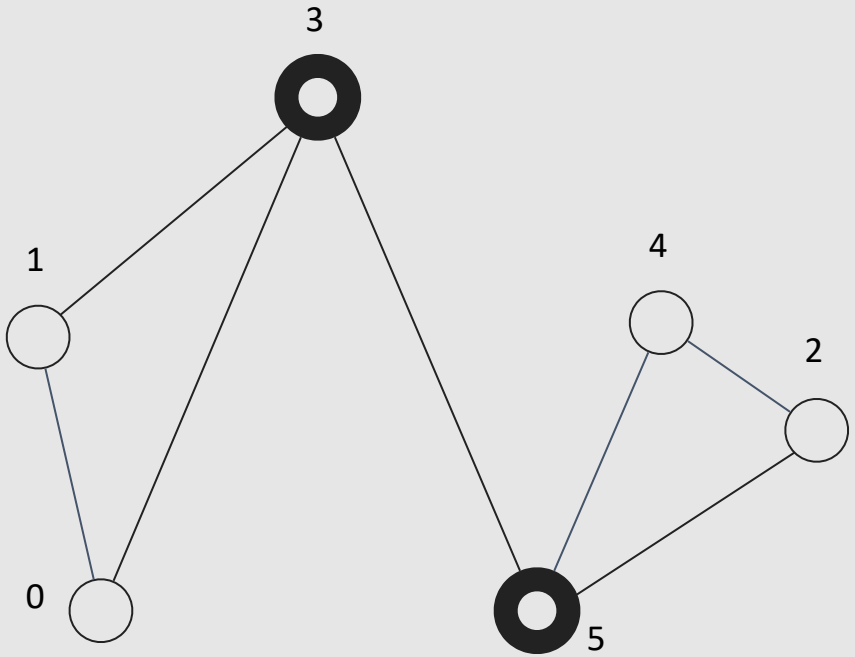



0
1
3
5
2
4

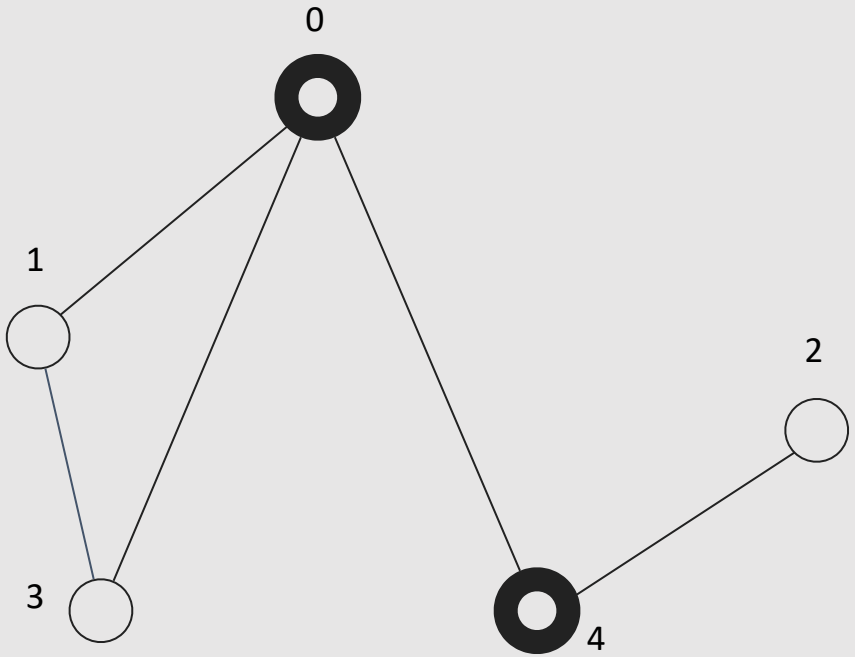
articulation point



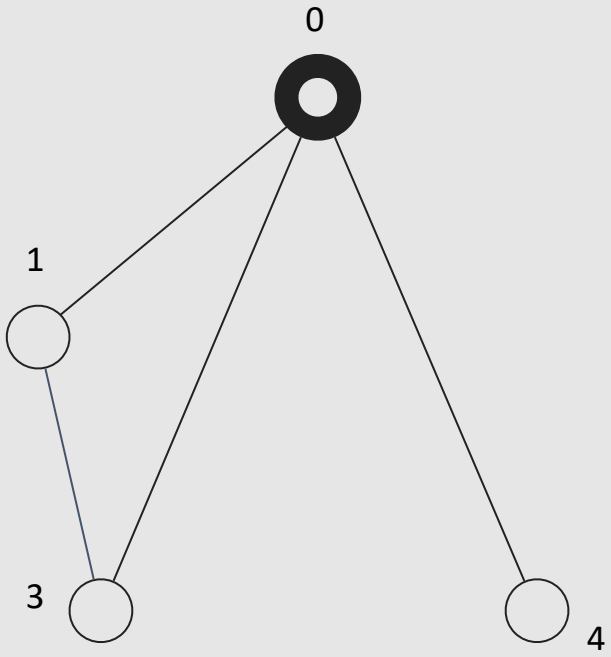
articulation point

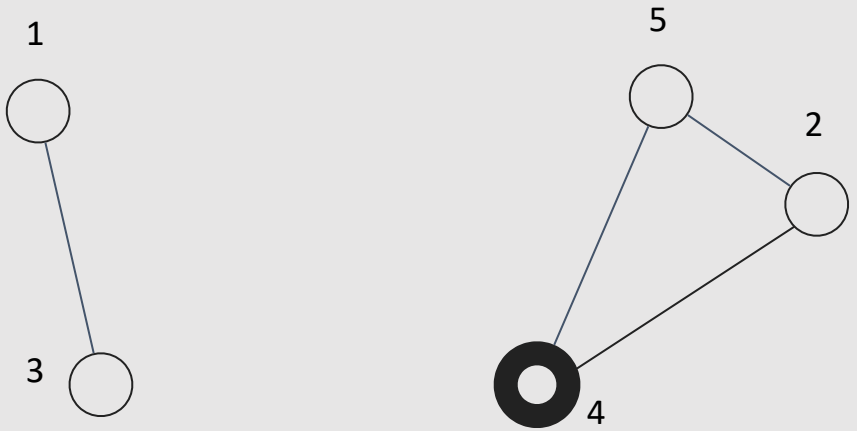


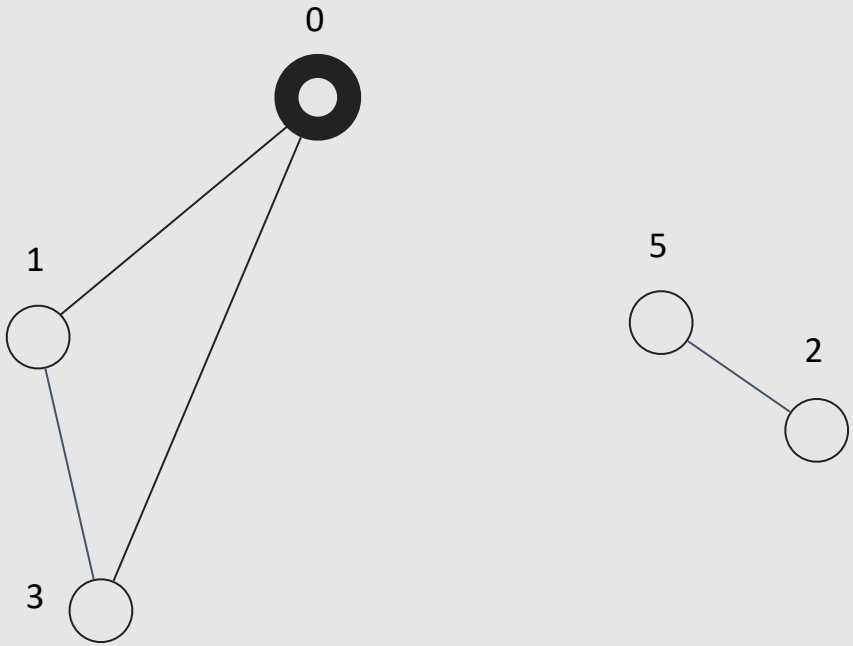
articulation point



articulation point

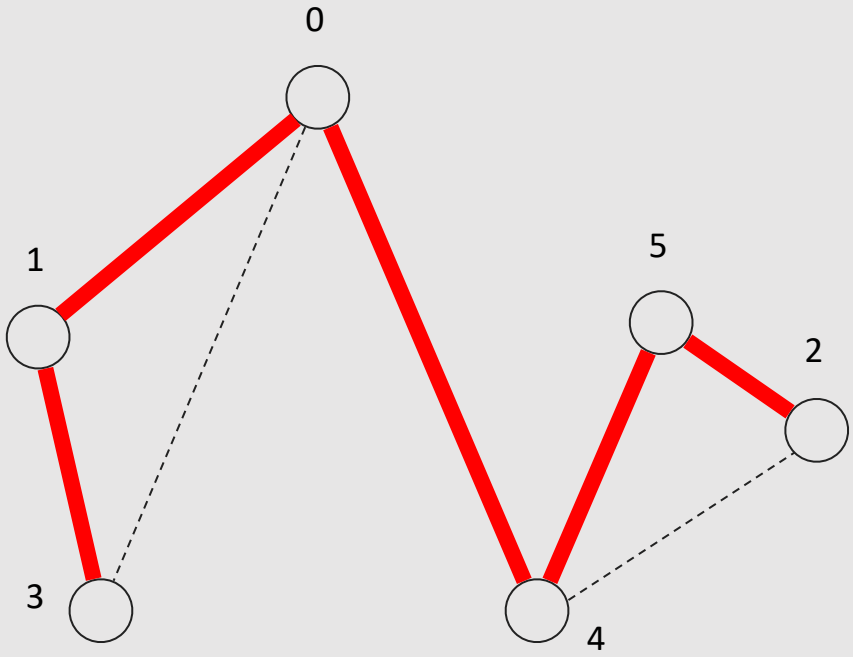



1
3
5
2
4


0
1
3
5
2

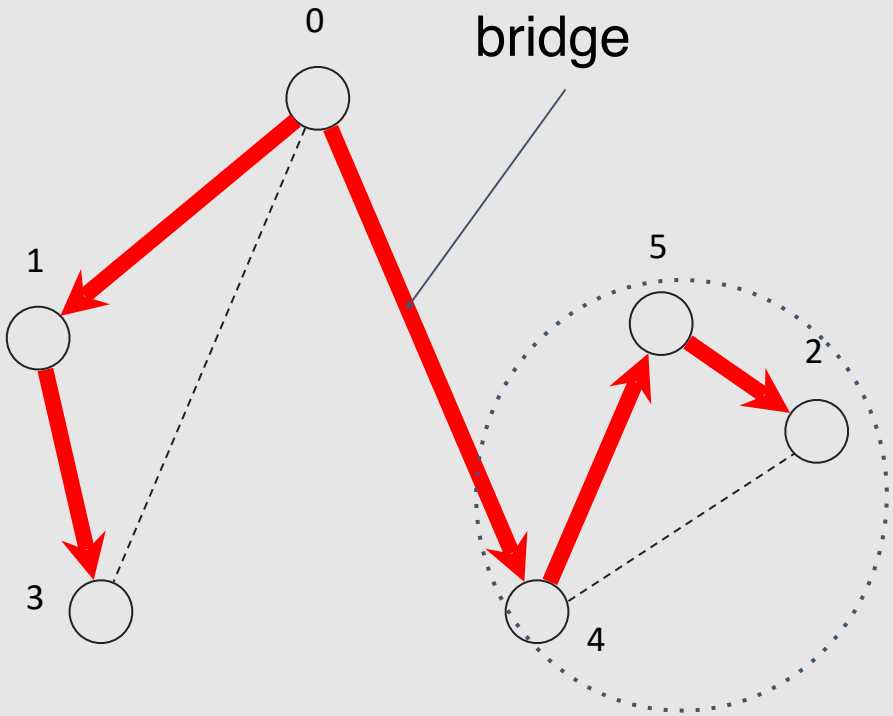
Bridges and articulation points

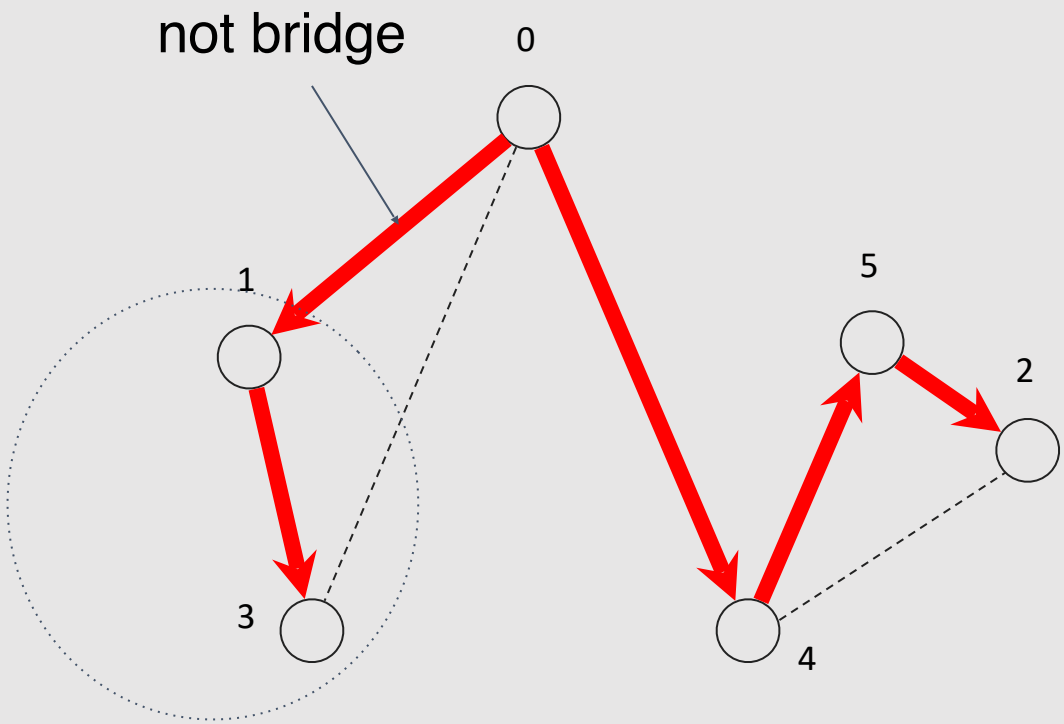
- Let's run DFS from an arbitrary vertex. We'll introduce new types of edges:
- Direct edges — They were visited in DFS.
- Back edges — They were not visited.


0
1
5
2
3
4

Idea

- (u, v) is a bridge if there is no path upward using downward edges and some back edges.


0
bridge
1
5
2
3
4


not bridge
0
1
2
3
4
5

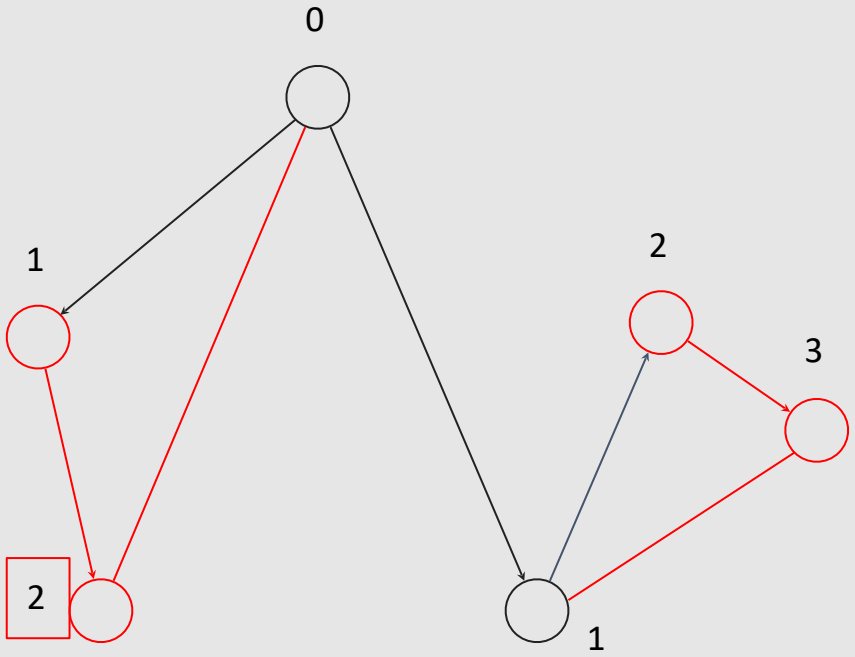
Idea

dp[u] - how high we can go using only downward edges and back edges.

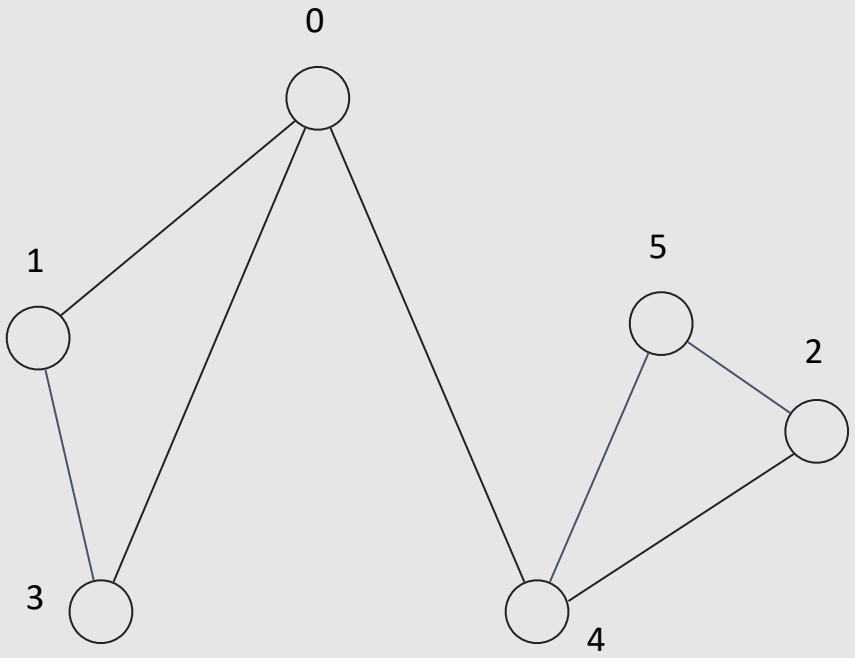
- dp[u] = h[u]
- dp[u] = dp[v], where (u, v) - downward edge
- dp[u] = h[v], where (u, v) - back edge

(u, v) is a bridge if h[u] < dp[v]

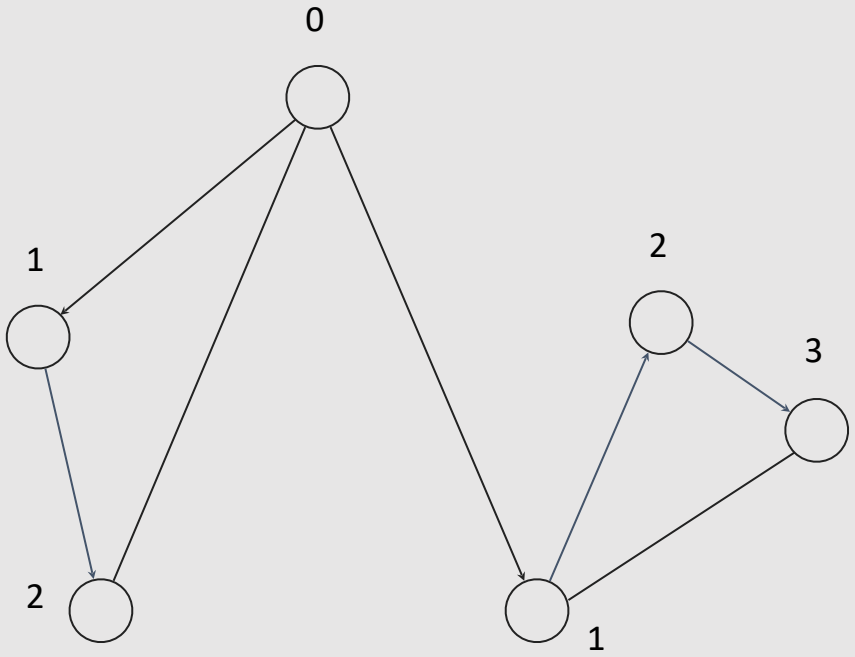
way up



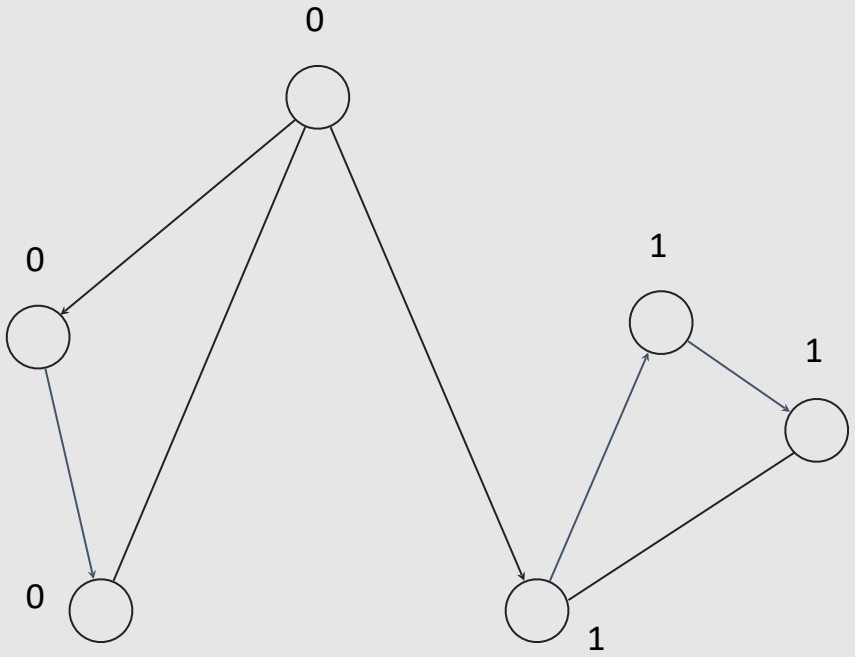
ids

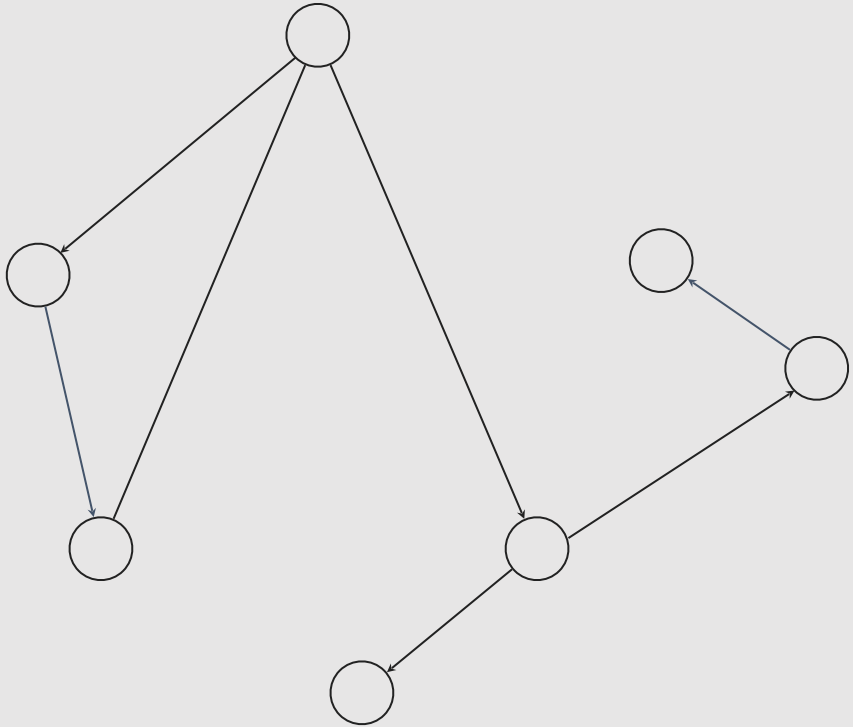


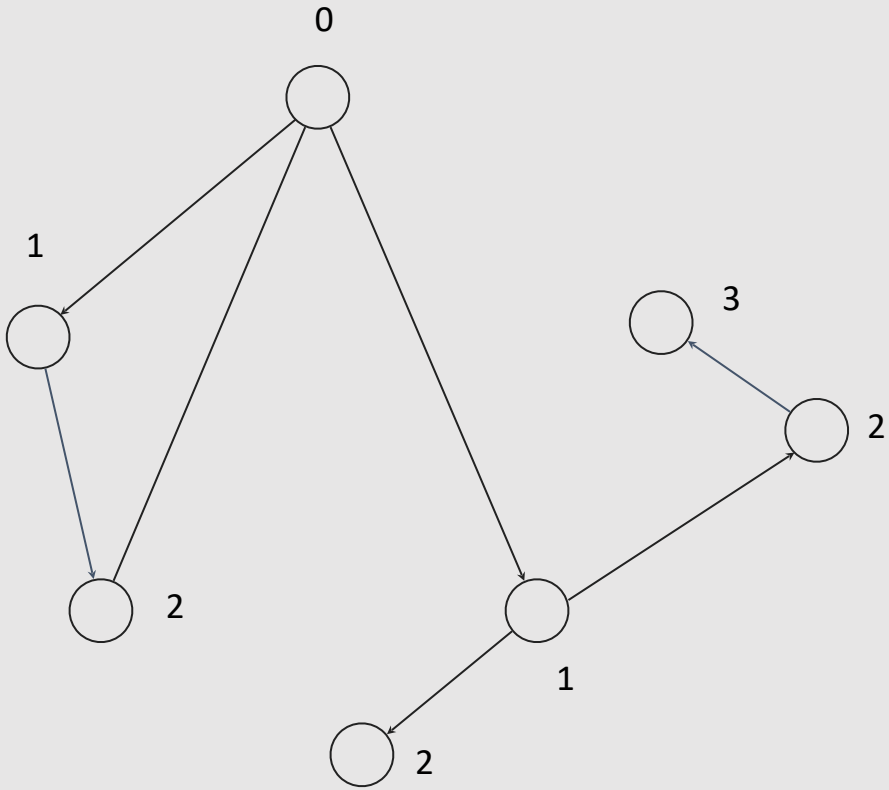
height = depth

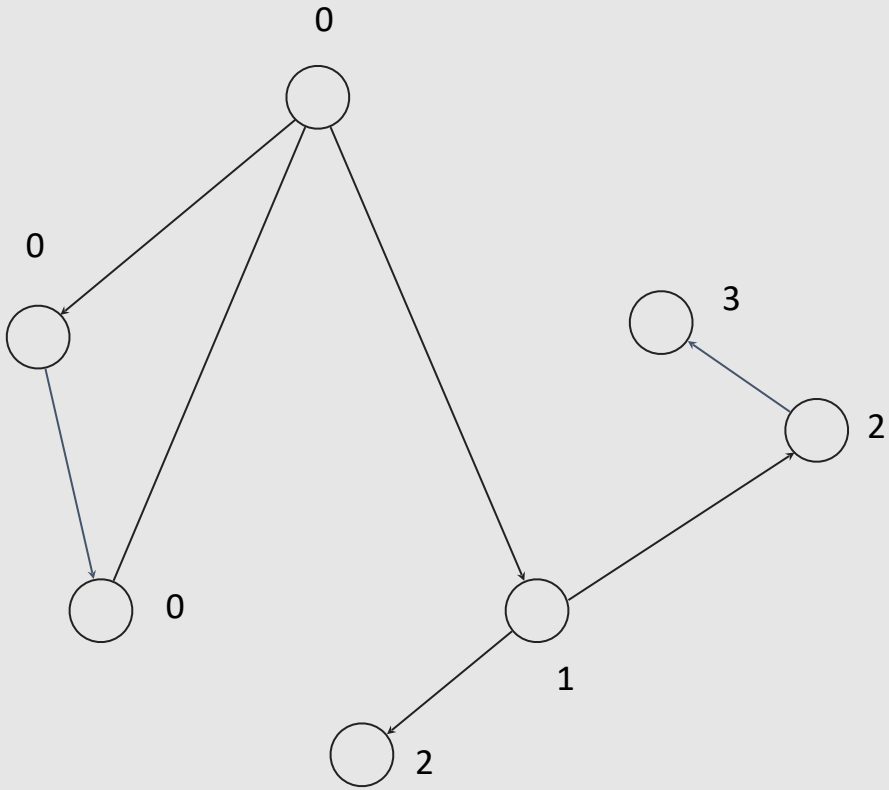


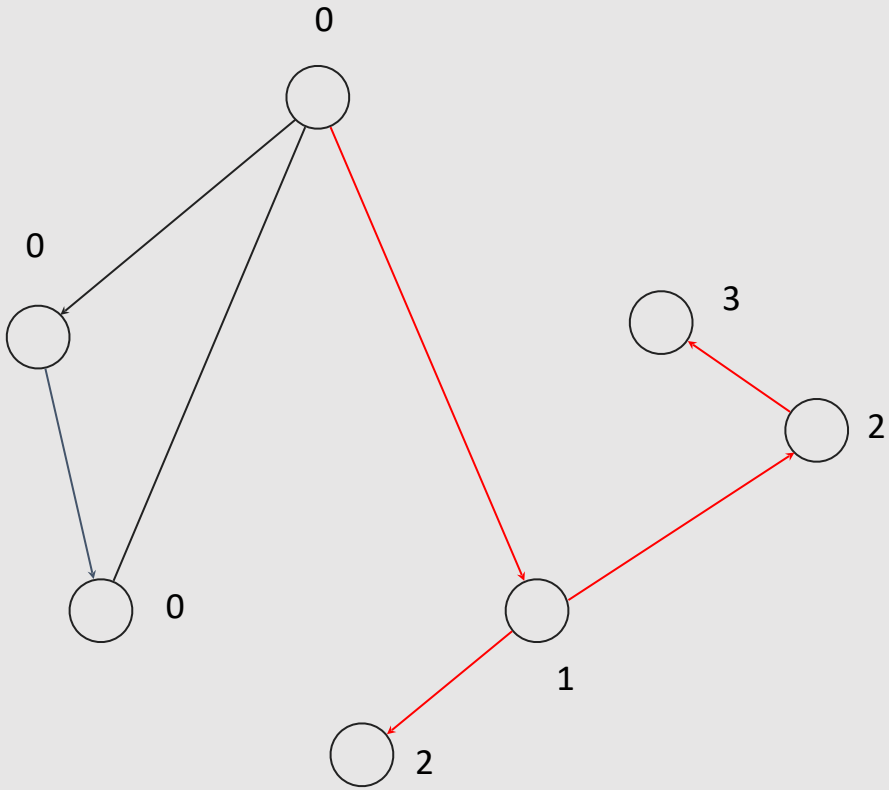
dp






0
1
3
2
2
1
2


0
0
3
2
0
1
2


0
0
3
2
0
1
2

Example

(0, 4) is a bridge, because h[0] < dp[1], 0 < 1

```cpp
void dfs(int v, int p, const std::vector<std::vector<int>>& g, std::vector<bool>& visited, int depth, vector<int> &h, vector<int> &dp) {
    visited[v] = 1;
    dp[v] = h[v] = depth;
    int children = 0;
    for (auto u : g[v]) {
        if (u != p) {
            if (visited[u]) {
                dp[v] = min(dp[v], h[u]); // back edge
            }
            else {
                dfs(u, v, g, visited, depth + 1, h, dp);
                dp[v] = min(dp[v], dp[u]); // edge down
                if (h[v] < dp[u]) {
                    cout << u + 1 << " " << v + 1 << "\n";
                }
            }
        }
    }
}

void FindBridges(const std::vector<std::vector<int>>& g, int n) {
    std::vector<bool> visited(n, false);
    std::vector<int> h(n), dp(n);

    for (int i = 0; i < g.size(); i++) {
        if (!visited[i]) {
            dfs(i, -1, g, visited, 0, h, dp);
        }
    }
}
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    FindBridges(graph, n);

    return 0;
}
```

Success #stdin #stdout 0.01s 5524KB

comments (0)

stdin

copy

```
6 7
1 2
1 4
2 4
1 5
5 6
6 3
3 5
```

stdout

copy

```
5 1
```

```python
def dfs(v, p, g, visited, depth, h, dp):
    visited[v] = True
    dp[v] = h[v] = depth
    for u in g[v]:
        if u != p:
            if visited[u]:
                dp[v] = min(dp[v], h[u])  # back edge
            else:
                dfs(u, v, g, visited, depth + 1, h, dp)
                dp[v] = min(dp[v], dp[u])  # edge down
                if h[v] < dp[u]:
                    print(u + 1, v + 1)

def find_bridges(g, n):
    visited = [False] * n
    h = [0] * n
    dp = [0] * n

    for i in range(len(g)):
        if not visited[i]:
            dfs(i, -1, g, visited, 0, h, dp)

if __name__ == "__main__":
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]

    for _ in range(m):
        from_vertex, to_vertex = map(int, input().split())
        from_vertex -= 1
        to_vertex -= 1
        graph[from_vertex].append(to_vertex)
        graph[to_vertex].append(from_vertex)

    find_bridges(graph, n)

```

Success #stdin #stdout 0.04s 9716KB

stdin

```
6 7
1 2
1 4
2 4
1 5
5 6
6 3
3 5
```

stdout

```
5 1
```

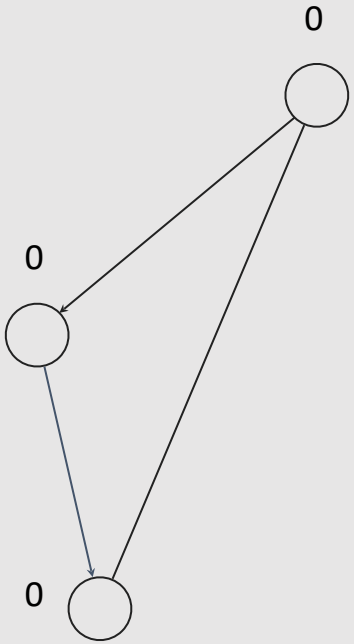
Articulation point

u is an articulation point, if for all sons v: dp[v] < h[u](we can go from v to some parent of u)

4 is an articulation point, because dp[5] >= h[4](1 >= 1)

Exception

Root


0
0
0

Exception

For the root, we check a special case - we need to see how many times DFS was initiated from it, that is, how many distinct connected components exist among its children.

```cpp
void dfs(int v, int p, const std::vector<std::vector<int>>& g, std::vector<bool>& visited, int depth, set<int> &cut_points, vector<int> &h, vector<int> &dp) {
    visited[v] = 1;
    dp[v] = h[v] = depth;
    int children = 0;
    for (auto u : g[v]) {
        if (u != p) {
            if (visited[u]) {
                dp[v] = min(dp[v], h[u]); // back edge
            }
            else {
                dfs(u, v, g, visited, depth + 1, cut_points, h, dp);
                dp[v] = min(dp[v], dp[u]); // edge down
                if (h[v] <= dp[u] && p != -1) { // root (p == -1) is corner case
                    cout << v + 1 << "\n";
                    cut_points.insert(v);
                }
                children++;
            }
        }
    }
    if (p == -1 && children > 1) {
        cout << v + 1 << "\n";
        cut_points.insert(v); // v -- root and cut point
    }
}

void FindCutPoints(const std::vector<std::vector<int>>& g, int n) {
    std::vector<bool> visited(n, false);
    std::set<int> cut_points;
    std::vector<int> h(n), dp(n);

    for (int i = 0; i < g.size(); i++) {
        if (!visited[i]) {
            dfs(i, -1, g, visited, 0, cut_points, h, dp);
        }
    }
}
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    FindCutPoints(graph, n);

    return 0;
}
```

Success #stdin #stdout 0.01s 5424KB

comments (0)

stdin

copy

```
6 7
1 2
1 4
2 4
1 5
5 6
6 3
3 5
```

stdout

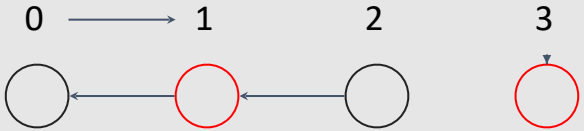
copy

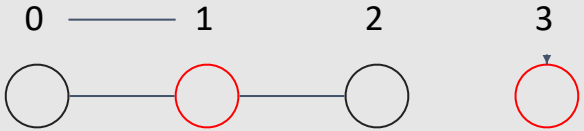
```
5
1
```

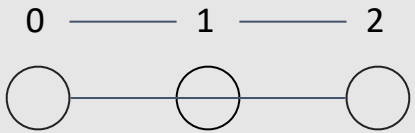
Task

Vasya has N piggy banks, numbered from 1 to N. Each piggy bank can be opened with its corresponding unique key or be broken.

Vasya put the keys into some of the piggy banks (he remembers which key is in which piggy bank). Now Vasya is planning to buy a car, and for that, he needs to take out money from all the piggy banks. At the same time, he wants to break as few piggy banks as possible (because he still needs to save money for an apartment, summer house, helicopter...). Help Vasya determine the minimum number of piggy banks he needs to break.


0
1
2
3

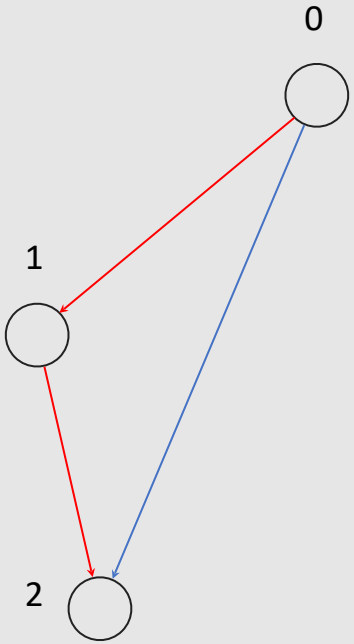

0
1
2
3

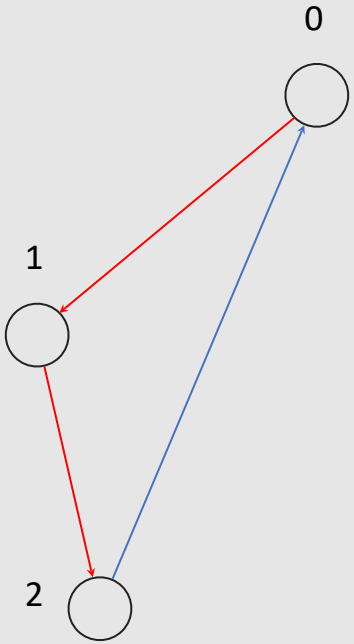

0
1
2

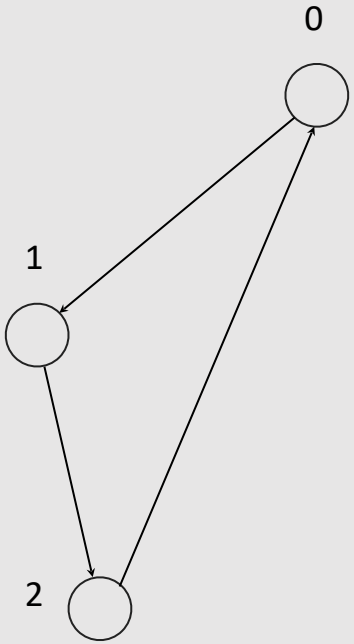
Task

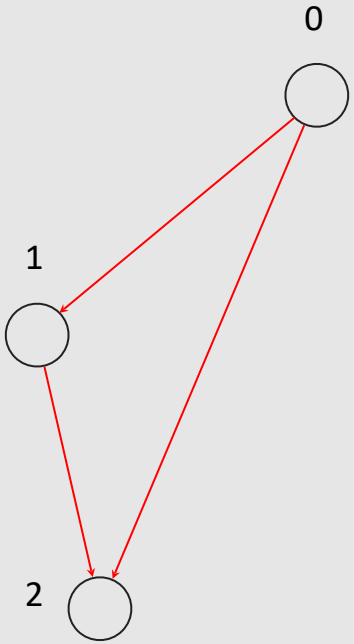
Given N points, numbered 1, 2, ..., N. From every point with a smaller number to every point with a larger number, there is an arrow of either red or blue color. The arrow coloring is called monochromatic if there are no such points A and B such that you can get from A to B using only red arrows as well as using only blue arrows.

Your task is to determine whether the given coloring is monochromatic based on the provided coloring.


0
1
2


0
1
2


0
1
2


0
1
2

All codes

edges(c++) - https://ideone.com/aBNXXJ

edges(python) - https://ideone.com/HhEJ0p

matrix(c++) - https://ideone.com/UdX0uQ

matrix(python) - https://ideone.com/aJn659

vector vector(c++) - https://ideone.com/ykU3d8

vector vector(python) - https://ideone.com/QzXRF9

dfs - tree(c++) - https://ideone.com/C7XSkH

dfs-tree(python) - https://ideone.com/RTNCRH

dfs(pyhton) - https://ideone.com/ytSMr8

dfs - graph(c++) - https://ideone.com/JRRG6b

depth(c++) - https://ideone.com/JKe2nr

depth(python) - https://ideone.com/JYnajS

dfs - without rec (c++) - https://ideone.com/8BO4S7

dfs - without rec(python) - https://ideone.com/UKIEsx

bfs(c++) - https://ideone.com/3vdOVj

bfs(python) - https://ideone.com/tOFCkm

All codes

distance(c++) - https://ideone.com/BXQ5Hq

distance(python) - https://ideone.com/rXQ2IM

euler tour(c++) - https://ideone.com/Wvg9Cx

euler tour(python) - https://ideone.com/Jl08mM

euler tour - 2(c++) - https://ideone.com/AxRLDa

euler tour - 2(python) - https://ideone.com/3frgTM

components(c++) - https://ideone.com/hcD1Na

components(python) - https://ideone.com/cpCOpZ

cycle(c++) - https://ideone.com/GOMrPO

cycle(python) - https://ideone.com/utO8VS

topsort(c++) - https://ideone.com/u5hsOk

topsort(python) - https://ideone.com/qYcbfF

cut points(c++) - https://ideone.com/Y5sL9r

bridges(c++) - https://ideone.com/veoWDG

bridge(python) - https://ideone.com/FuyXEJ

condensation(python) - https://ideone.com/Njq8Ys

condensation(C++) - https://ideone.com/VQVFpf

0-1 bfs(c++) - https://ideone.com/AZNd43

0-1 bfs(python)