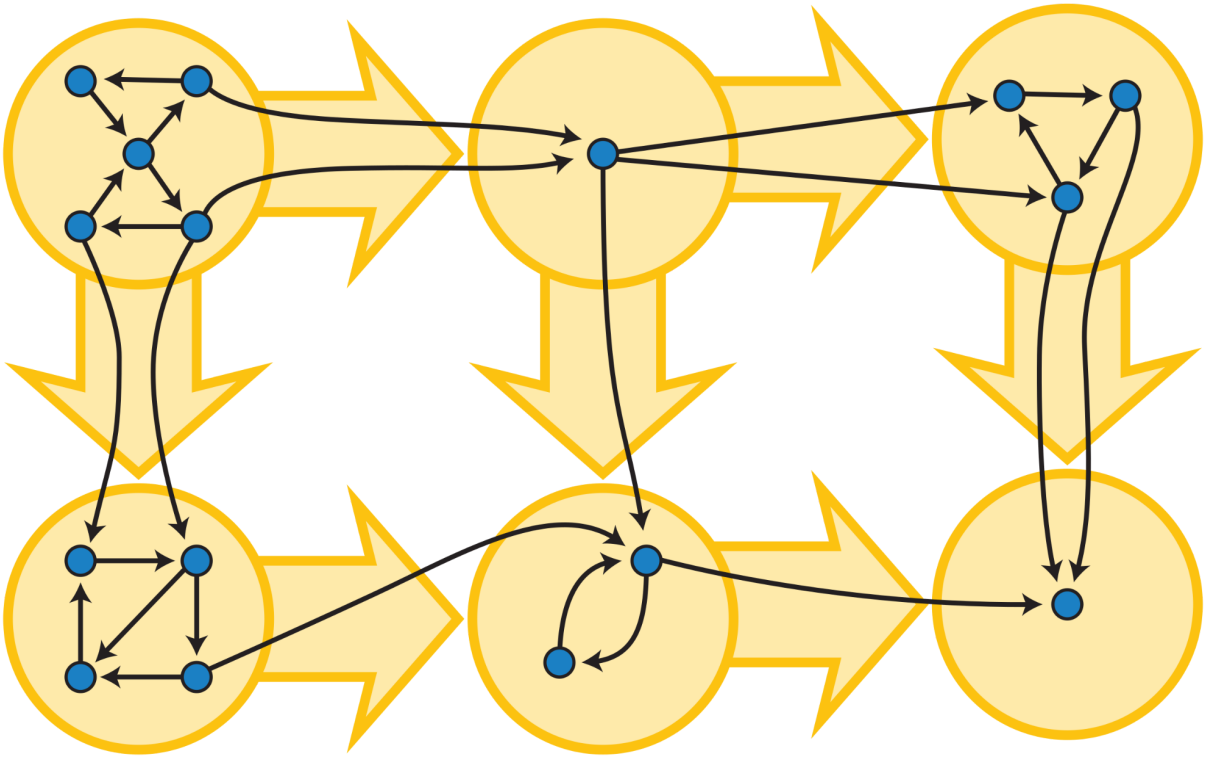
Design and Analysis of Algorithms

Dfs to SCC

Strongly Connected Components, Submitted by, Md. Emon Khan_30



Introduction

Strongly Connected Components (SCCs) and Depth-First Search (DFS) are fundamental concepts in graph theory and algorithms, and they are interconnected in the context of analyzing and understanding the connectivity of directed graphs. DFS plays a vital role in identifying SCCs through algorithms like Kosaraju's algorithm and Tarjan's algorithm.

What are Connected Components?

In graph theory, a connected component is a subgraph within a larger graph in which each pair of vertices is connected by a path. In other words, a connected component is a subgraph where there is a path between every pair of vertices**(ignoring the direction in Directed Graph**)

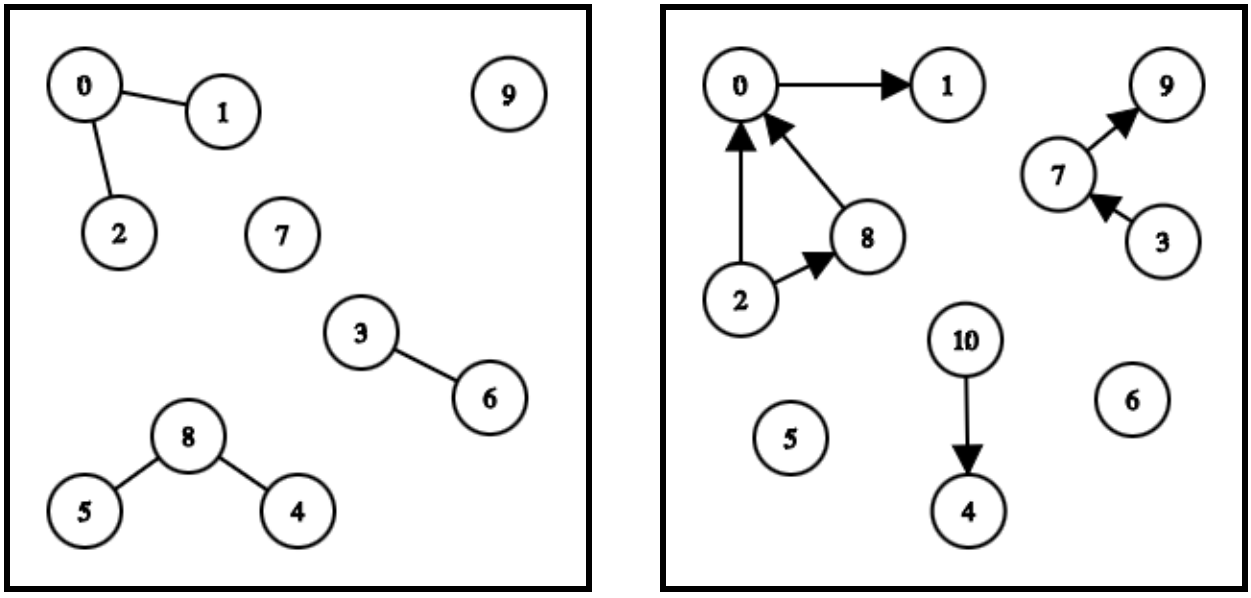


Figure-1,2: Connected Components in Undirected and Directed Graph

Connected components can exist in both directed and undirected graphs. In an undirected graph, a connected component is a subgraph where there is a path between every pair of vertices. In a directed graph, connected components are 2 kinds of **SCCs**(strongly connected components) and **WCCs**(weakly connected components).
Figure-1, 2 represent 5 connected components in an undirected graph and a directed graph accordingly

Connected components provide a way to break down a graph into smaller, more manageable parts, which can be particularly useful for analyzing the structure and connectivity of complex networks.

2 major key features of connected components,

Maximality: A Connected Component is maximal. That means you can not extend it by adding one or more vertices.

Disconnected Subgraphs: In a graph, there can be multiple connected components. These components are disjoint, which means there are no edges that connect different components vertices

Difference between Strongly Connected and Weakly Connected Graph?

Strongly Connected Graph: A directed Graph is said to be strongly connected if there is a path between **a to b** and **b to a** where a, and b are the vertices in that graph.

Weakly Connected Graph: A directed Graph is said to be weakly connected if there is a path between every two vertices in that graph. If you can find an undirected path between all pairs of vertices, it will make the graph weakly connected. **(ignoring the direction**)

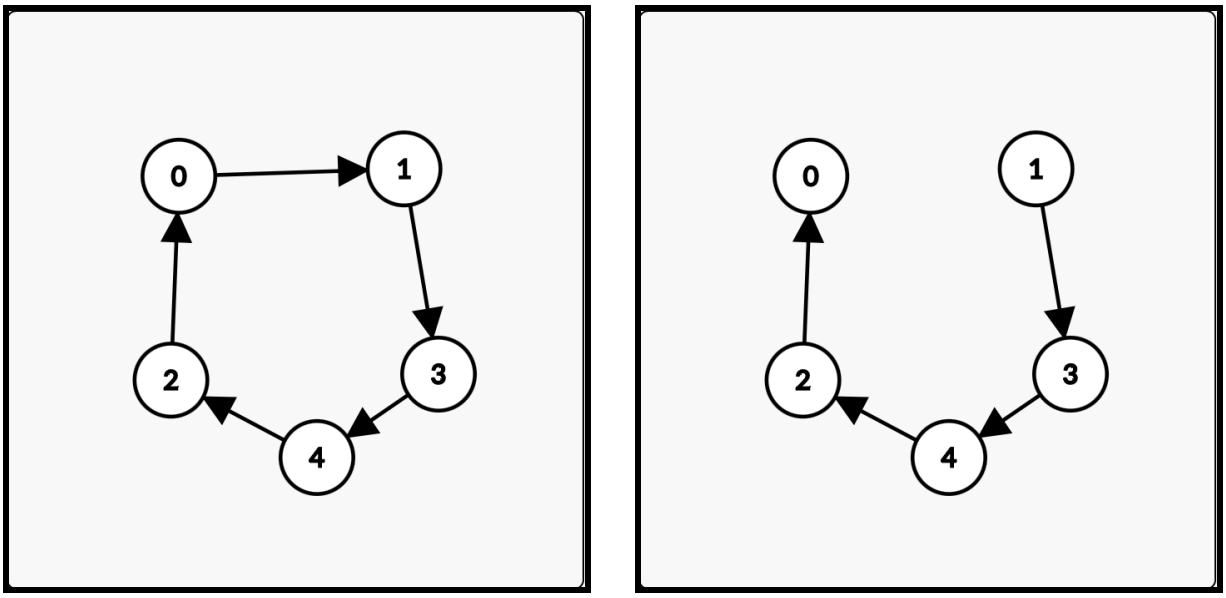


Figure-3,4: Strongly Connected Graph and Weakly Connected Graph

What are Weakly Connected Components?

A weakly connected component (WCC) is a subgraph of the larger graph where all vertices are connected to each other by some undirected path. **(ignoring the direction**)

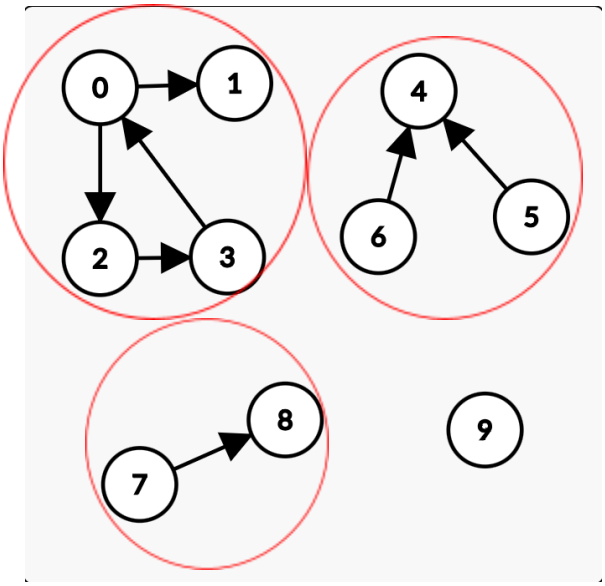


Figure-5: Weakly Connected Components

There are 3 weakly connected components in Figure 5

***Note that subgraphs with 1 vertices are considered to be strongly connected by definition**

***A Weakly Connected Component consists of one or more strongly connected components.**

Strongly Connected Components

A strongly connected component is a directed subgraph within a larger graph where the subgraph is a **strongly connected graph.** That means there is a path between **a to b** and **b to a** where a, and b are the vertices in that subgraph. More specifically SCC is a maximally

connected subgraph where you can reach any vertex from any other vertex by following directed edges.

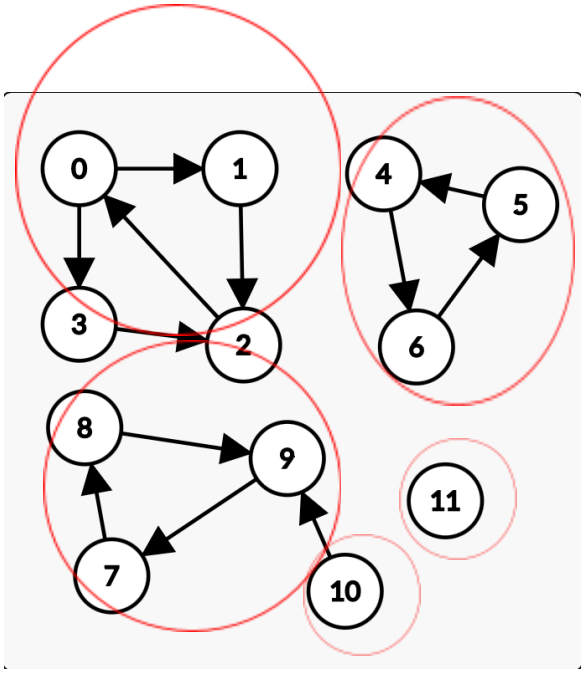


Figure-6: Strongly Connected Components

Key Characteristics

1. Every vertex in a SCC is reachable from every other vertex in the same SSC
2. Each vertex belongs to only one SSC
3. Subgraphs with **1** vertices are considered to be strongly connected by definition
4. A directed graph is **acyclic** if and only if it has no strongly connected subgraphs with more than one vertex
5. If all of the SCC consists of one vertex then the resulting graph is **acyclic**

Difference between Connected and Strongly Connected Components?

*In a Directed graph when we search for connected components we ignore the direction,

But in the Strongly connected component, we need to consider the direction.

*A connected component can consist of one or more Strongly connected components

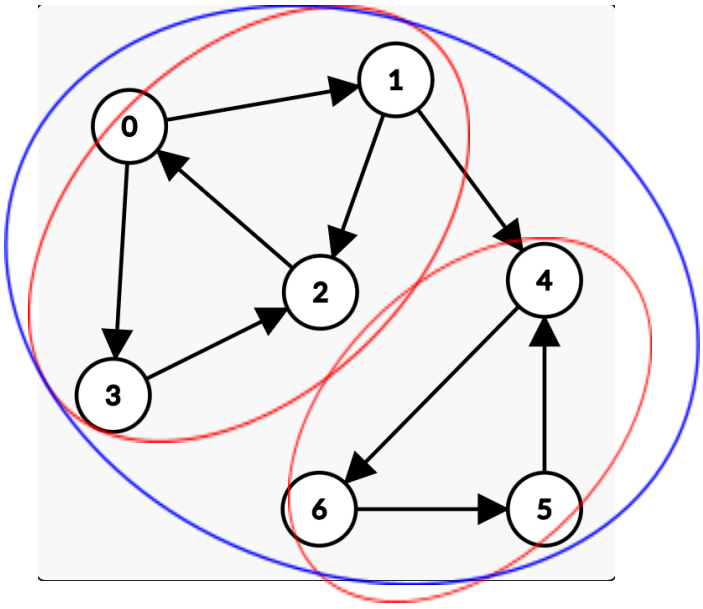


Figure-7

Red marked components are the 2 strongly connected components and blue one is the connected components. That means 1 connected component consists of 2 SCCs.

Algorithm to find Strongly connected components?

There are several DFS-based algorithms to find Strongly connected components in directed graphs.

Kosaraju's algorithm

- It is a two-pass algorithm used to identify
- In the first pass of this algorithm, a depth-first search is run to the original graph. It assigns an order to all the vertices based on their finishing time. This order is used in the second pass.
- In the second pass of this algorithm, there is another DFS, But this time DFS run the transpose graph of the original graph.
- It recursively explores all the vertices in the reverse order of the finishing time of the first pass
- In the second pass, each exploration identified a newly strongly connected component.

Tarjan's algorithm

- It is a single-pass DFS-based algorithm to find strongly connected components in a directed graph.
- The main idea of this algorithm is to maintain a stack of vertices that have been explored but not yet assigned to a component.
- Tarjan's algorithm calculates "low numbers" for each vertex, which represent the index number of the highest ancestor reachable in one step from a descendant of the vertex.
- When a set of vertices is encountered with a common ancestor, they can be popped off the stack and form a new strongly connected component.

Why are these algorithms designed?

Tarjan's and Kosaraju's algorithms are designed to find the strongly connected component of a directed graph. They both use the depth-first search method to find SCCs.

These algorithms are fundamental in graph theory and have various applications in computer science and real-world scenarios,
Graph Analysis: The primary application of these algorithms is in graph analysis. They help in understanding the connectivity structure of directed graphs by partitioning the vertices into SCCs.

Dependency Analysis: In software engineering, these algorithms find dependencies between modules or functions in a codebase. Detecting SCCs can help manage software architecture and understand the impact of code changes.

Network Analysis: In network analysis, these algorithms can be used to discover communities or clusters of nodes with strong connections in social networks, communication networks, or web graphs.

Natural Language Processing: In text analysis, strongly connected components can be used to identify tightly related phrases or entities within a text corpus, enabling the extraction of meaningful information from unstructured data. Transportation and Route

Game Development: In game development, SCCs can be used to optimize game levels by identifying areas that can be explored or encountered together, allowing designers to create more engaging gameplay experiences.

Social Network Analysis: Analyzing social networks, such as Facebook or Twitter, involves finding clusters of users with mutual connections. SCCs can help in understanding network dynamics and identifying influential users or communities.

Historical context

Kosaraju-Sharir's algorithm(Kosaraju's algorithm) was discovered by Sambasiva Rao Kosaraju an Indian-American professor of computer science at Johns Hopkins University. He suggested this algorithm in 1978 but did not publish it. Later another Israeli mathematician and computer scientist **Micha Sharir** independently discovered it and published it in 1981

The need to understand connectivity and cycle in a directed graph and find strongly connected components and optimization this algorithm was designed.

At the current time, this algorithm is used in various fields. Mainly it is used in graph analysis, compiler optimization, game theory, network analysis, and Dependency analysis.

Tarjan's algorithm remains relevant in computer science, and its significance has expanded even further due to the increasing complexity of systems and the growth of graph-based data. Over time, the basic principles of Kosaraju's algorithm have remained relatively unchanged since its development, and various optimizations and improvements have been proposed over the years to make it even more efficient in practice. These optimizations may involve reducing memory usage and minimizing the number of data structures and operations.

Kosaraju's Explanation

We need to learn some properties before we going to explanation of this algorithm

Properties 1: Reversing all the edges of a graph won't change the number of SCCs.

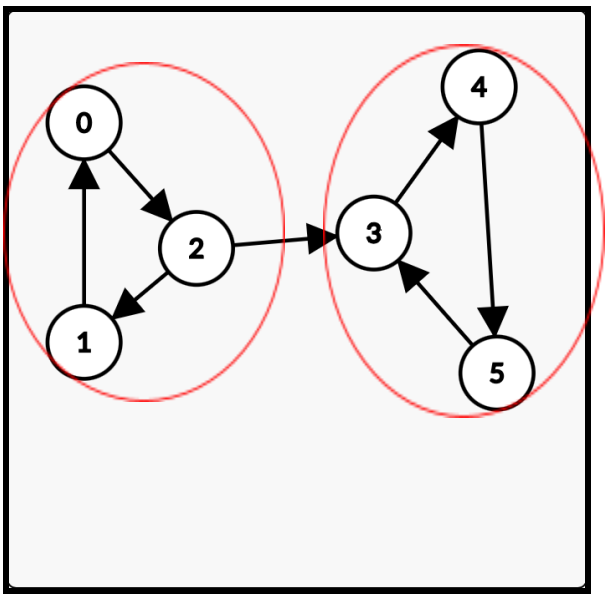


Figure 8: Original Graph

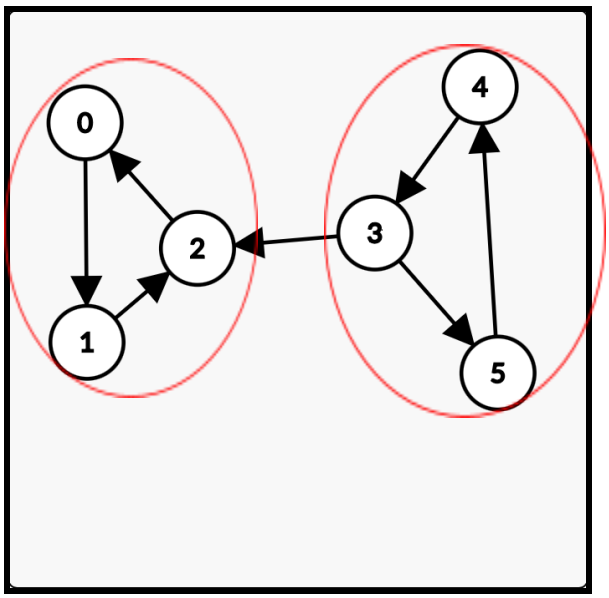
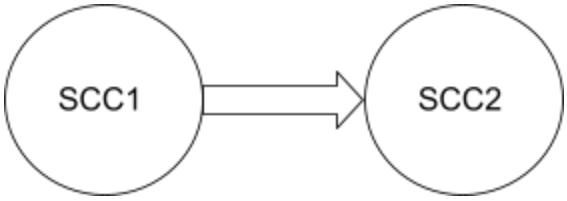
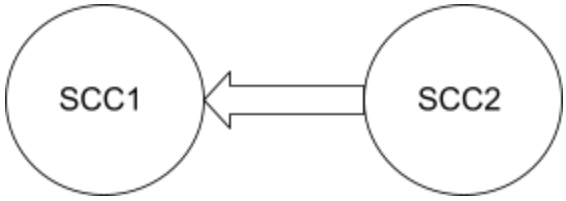


Figure 9: Reverse Graph

Properties 2: If we start dfs from SCC1 we can reach to SSC2 in original graph





But if we start from SCC1 in the reverse graph we can't reach SSC2 in the reverse graph As there is no forward path between SSC1 and SCC2.

We need to maintain a specific order so that after visiting all the vertices in the SSC1 we stop traversing and find all the vertices and push in a list of component-1. Then we do a manual jump to SCC2 then find other components.

First, we are going to maintain a data structure vector to store the topological order. Let the vector be V. also another vector C to store the component

Algorithm:

1. Let V be empty. Mark each vertex u as unvisited

2. For each vertex u of the original graph recursive call dfs(u)

   If u is unvisited

   - Mark u as visited
   - Run dfs(u) in all the neighbours of u
   - Push u to vector V

   Else do nothing

3. Reverse the vector V

4. For each vertex u in the vector V recursive call dfs(u) in the reversed graph

   If u is not assigned to a component

   - Push u to vector C
   - Run dfs(u) in all the neighbors of u

   Else do nothing

5. When there is no dfs call, store vector C as a component

6. Clear vector C and run dfs all the remaining vertex in V

```
vector<int> G[N+5], RG[N+5], order, comp;

vector<vector<int>> components;

vector<bool> vis;

int n, m;
```

Defining the original vector as G, reverse vector as RG, order for topological order, comp to store the all the vertex that belongs to same component, components to store all the SCC, vis vector check already visited or not

```
  for(int i = 0; i < m; i++) {

      int u, v;

      cin >> u >> v;

      G[u].push_back(v);

      RG[v].push_back(u);

  }
```

Taking the input original graph and reversed graph

```
vis.assign(n+1, false);

  for(int i = 1; i <= n; ++i) {

      if(!vis[i]) {

          dfs(i);

      }

  }
```

Assigning all the vertex as unvisited

```
void dfs(int vertex) {

   vis[vertex] = true;

   for(int child : G[vertex])

       if(!vis[child])

           dfs(child);

   order.push_back(vertex);

}
```

First dfs call to find the topological order

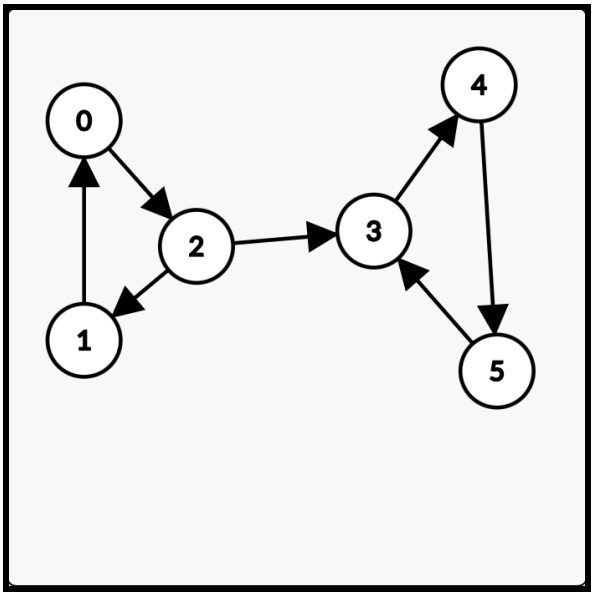
```
   reverse(order.begin(), order.end());

   vis.assign(n+1, false);

   for(auto ele : order) {

       if(!vis[ele])dfs2(ele);

       if(comp.size())components.push_back(comp);

       comp.clear();

   }
```

Reversing the topological order and assigning all the visited vertex as unvisited to run second dfs again. Calling dfs2 to all the vertices in order vector. After a dfs call, if component size is greater than 1 we are considering it as a SCC. And storing it in components nested vectors.

```
void dfs2(int vertex) {

   vis[vertex] = true;

   comp.push_back(vertex);

   for(auto child : RG[vertex])

       if(!vis[child])

           dfs2(child);

}
```

This is dfs2 to find all the vertex that belongs to same SCC.

Practical Example




```
6 7
0 2
2 1
1 0
2 3
3 4
4 5
5 3
```

If we run the first dfs to this graph in the topological order based on their finishing time

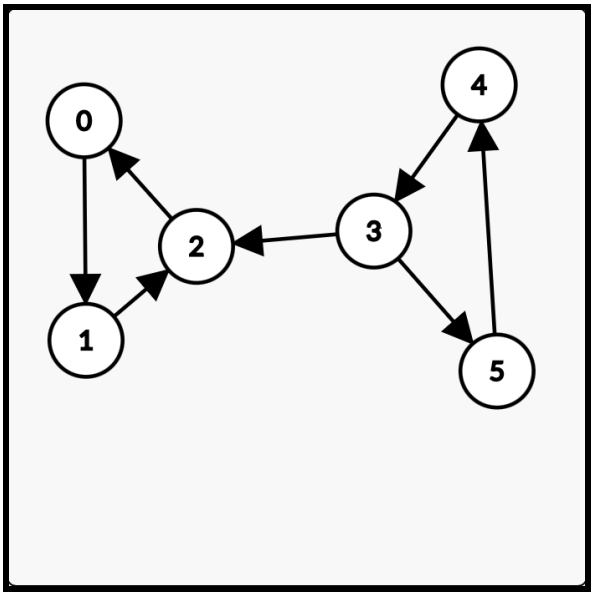
We will find this order

1	5	4	3	2	0

After reversing the topological order, this is initial state

0	2	3	4	5	1

The we call dfs2 to 0 to reversed graph



0	2	3	4	5	1

In the first call (0, 1, 2) will be in the same component

And we start dfs2 again in vertex 3 then (3, 4, 5) will be in another component

0	2	3	4	5	1

So we found (0, 1, 2), (3, 4, 5) two SCC.

Practice Problem

The authorities at the University of Dhaka have decided to select some students from their institution for a study tour and have established specific rules for the selection process. Students at Dhaka University can pass information in one way manner. This means that Student A can pass information to Student B, but B cannot pass information back to Student A. If Student A can pass information to Student B, it is considered a **direct** connection. Furthermore, if Student A can pass information to Student B, and Student B can pass information to Student C, then Student A will be able to pass information to Student C. This situation is referred to as an **indirect** connection.

The authorities want to create some groups such that each group has the maximum number of members, and all the students in the same group can pass information xto all the other students in the same group via direct or indirect connection. They aim to send the group with the highest number of students.

Input

The first line contains two integers, n, and m, representing the number of students and the number of one-way connections.

The next m lines each contain two integers, u and v, indicating that student u can pass information to student v.

Output

Print the group member with the highest number of students that can be . If 2 groups have the same highest number of members print them any.

Input	Output
7 8 1 2 2 4 4 7 2 3 3 1 4 5 5 6 6 4	1 2 3

```cpp
#include<bits/stdc++.h>

using namespace std;

const int N = 100 + 5;

vector<int> G[N+5], RG[N+5], order, comp;

vector<vector<int>> components;

vector<bool> vis;

int n, m;



void dfs(int vertex) {

   vis[vertex] = true;

   for(int child : G[vertex])

       if(!vis[child])

           dfs(child);

   order.push_back(vertex);

}

void dfs2(int vertex) {

   vis[vertex] = true;

   comp.push_back(vertex);

   for(auto child : RG[vertex])

       if(!vis[child])

           dfs2(child);

}

int main() {
```

```
cin >> n >> m;

for(int i = 0; i < m; i++) {

    int u, v;

    cin >> u >> v;

    G[u].push_back(v);

    RG[v].push_back(u);

}

vis.assign(n+1, false);

for(int i = 1; i <= n; ++i)

    if(!vis[i])

        dfs(i);

reverse(order.begin(), order.end());



vis.assign(n+1, false);

for(auto ele : order) {

    if(!vis[ele])dfs2(ele);

    if(comp.size())components.push_back(comp);

    comp.clear();

}

vector<int> ans;

int mx = 0;

for(auto ele : components) {

    if(ele.size() > mx) {

        mx = ele.size();
```

```
            ans.clear();

            for(auto i : ele)ans.push_back(i);

        }

    }

    sort(ans.begin(), ans.end());

    for(auto ele : ans)cout << ele << " ";

}
```

In this problem we used the basic idea of Kosaraju's algorithm to solve.

Conclusion

Kosaraju's algorithm is a powerful tool for finding strongly connected components (SCCs) in directed graphs. It is significant because it helps to break down complex networks into smaller, interconnected subgraphs, helping us better understand the structural and connectivity aspects of the graph.

This algorithm is proven in various fields, including graph analysis, software engineering, network analysis, natural language processing, game development, and social network analysis.

The strengths of Kosaraju's algorithm include its effectiveness in identifying SCCs and its two-pass structure that efficiently computes the components. It is widely applicable and has real-world implications in optimizing software architecture, analyzing social networks, and much more. The algorithm's historical context, developed by Kosaraju and Sharir, underscores its continued relevance in the ever-growing field of computer science. However, it's important to recognize that while Kosaraju's algorithm is highly effective, it is not the most space-efficient algorithm. It can be memory-intensive for large graphs, which can be a limitation in some cases. Future developments in this area could focus on optimizing space usage and reducing the algorithm's memory requirements, making it more practical for very large graphs. In summary, Kosaraju's algorithm has significantly contributed to graph theory and practical applications, and its continued evolution to modern computational challenges holds promise for more efficient usage in the future.

Reference

1. https://cp-algorithms.com/graph/strongly-connected-components.html
2. https://csacademy.com/app/graph_editor/
3. https://en.wikipedia.org/wiki/Strongly_connected_component
4. https://www.geeksforgeeks.org/strongly-connected-components/
5. https://en.wikipedia.org/wiki/Strongly_connected_component#/media/File:Graph_Condensation.svg