Title – Finding Strongly Connected Components in a Social Network Graph
Group Number – 13
Roll Nos – 106120026, 106120094
Email IDs – sivasundar.per@gmail.com**,** s.rahulshankar@gmail.com

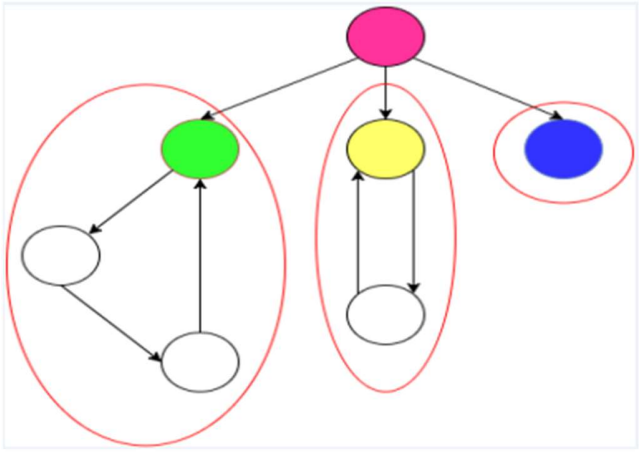
Abstract
Our research topic deals with – **Finding of Strongly Connected Components in a Social Networking Graph.** Furthermore, we use our findings and apply it on a real-world issue – **Optimizing Social Media Advertising by identifying Target Audiences through SCC Detection.**
A major issue with Social Media Advertising is that companies try to advertise to as many people as possible without conducting any research on who their target consumers are. Let's call this **Inefficient Social Media Advertising.**
By utilizing the concept of SCCs in a Social Networking Graph, companies can use it to identify their target users and selectively advertise to such groups. This can save a lot of wasted computing power, time, and money. This is what we call **Efficient Social Media Advertising.**
We can do so by identifying the Strongly Connected Components in a Social Network Graph which would represent groups of users having shared interests (Sports, Bollywood, Technology etc.).
One way of doing so is by using a brute force algorithm to identify the number of Strongly Connected Components. We can use **Floyd Warshall Algorithm** which is used to find the shortest distance between all pairs of vertices in a directed graph. If the distance between a pair of vertices turns out to be infinity, then we have identified a SCC.
The main issue with the pre-existing algorithm is its' large time complexity O(V^3). This can lead to a huge waste of computing resources and time.
We look to address this issue by considering 3 other algorithms which can be used to find SCCs in a graph – **Kosaraju's, Tarjan's** and **Cheriyan-Mehlhorn-Gabow** algorithms.
We will be implementing the codes for these algorithms as proof of their concepts and clearly displaying the output.
We will be using x different performance metrics (insert here) and tabulating the results based on these metrics to present a comparative study on the 3 algorithms and to arrive at a conclusion.

Introduction
The Internet has turned our existence upside down. It has revolutionized communications, to the extent that it is now our preferred medium of everyday communication.
As of 2022, there are 4.55 billion active users on social media!
This provides a huge opportunity for companies to advertise their products or services. Companies can identify their target audience and then specifically target them.

A major issue that companies run into is trying to advertise to all groups of people rather than trying to stick to their target audience.
This has 2 negative effects –

- Waste of computing resources and money.
- A loss of audience as compared to an increase in advertising.

Social Media Advertising can be smartly done with the help of SCC Detection Algorithms.
In Social Media Networking Sites such as Facebook and Twitter, users are represented in the form of a Social Network Graph (SNG) where the users are represented as nodes and the edges represent relationships.
A Strongly Connected Component in a SNG would indicate that all the vertices (users) are connected to each other through some mutual interests. An example is given below.
Here, the Pink node represents the root node while the Green, Yellow and Blue nodes represent a Sports, Cars and Bollywood page. The White nodes represent the users subscribed to those pages.



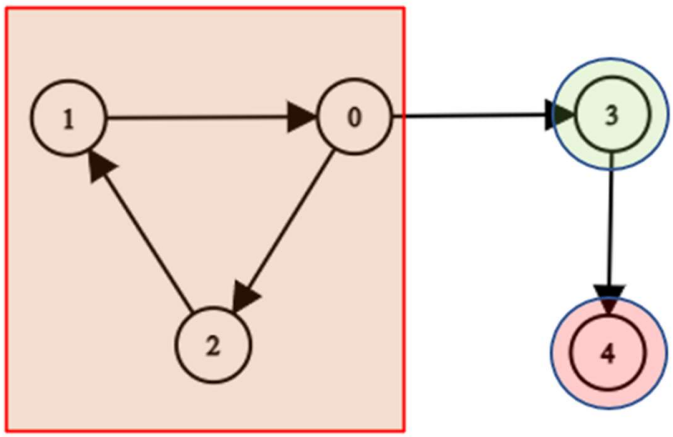
We propose to apply SCC detection algorithm to the Social Network Graphs to identify smaller groups of nodes which are related to each other by some criteria (Interests such as Sports, Technology, Celebrities etc.).
For example, in the above example, a Shoe company can selectively advertise only to those users in the SCC for Sports pages.
One way of doing so is by using a brute force algorithm to identify the number of Strongly Connected Components. We can use **Floyd Warshall Algorithm** which is used to find the shortest distance between all pairs of vertices in a directed graph. If the distance between a pair of vertices turns out to be infinity, then we have identified a SCC.
The main issue with this algorithm is its' large time complexity i.e. O(V^3). This can lead to a huge waste of computing resources and time.
To combat this issue, we look at 3 other algorithms – **Kosaraju's, Tarjan's** and **Cheriyan-Mehlhorn-Gabow** algorithms, which are much more optimized and have better time complexities and compare them based on different performance metrics

Algorithm 1 – Kosaraju's Algorithm

Kosaraju's Algorithm is a linear time algorithm used to find the strongly connected components of a directed graph.
It takes advantage of the fact that the transpose of a directed graph has the same SCCs as that of the original graph.
Pseudocode -

- `Perform DFS traversal of the graph. Push starting node to stack before returning.`
- `Find the transpose graph by reversing the edges.`
- `Pop nodes one by one from the stack and again to DFS on the modified graph.`

Example graph-



Code-

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;
    list<int> *adj;
```

```
    void fillOrder(int v, bool visited[], stack<int> &Stack);
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);
    void printSCCs();
    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
void Graph::DFSUtil(int v, bool visited[])
{
    visited[v] = true;
    cout << v << " ";
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
```

```
    visited[v] = true;
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);
    Stack.push(v);
}
void Graph::printSCCs()
{
    stack<int> Stack;
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;
    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, Stack);

    Graph gr = getTranspose();
    for(int i = 0; i < V; i++)
        visited[i] = false;
    while (Stack.empty() == false)
    {
        int v = Stack.top();
        Stack.pop();
        if (visited[v] == false)
        {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
}

int main()
{
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "SCC in given graph is :\n";
    g.printSCCs();

    return 0;
```

```
}
```

Output-

```
SCC in given graph is :
0 1 2
3
4
```

Time Complexity of Kosaraju's Algorithm – **O(V+E)**

This algorithm however, uses 2 traversals of the graph using DFS. This can be improved upon by using Tarjan's or Gabow's Algorithms.

Applications –

Vehicle Routing

Algorithm 2 – Tarjan's Algorithm

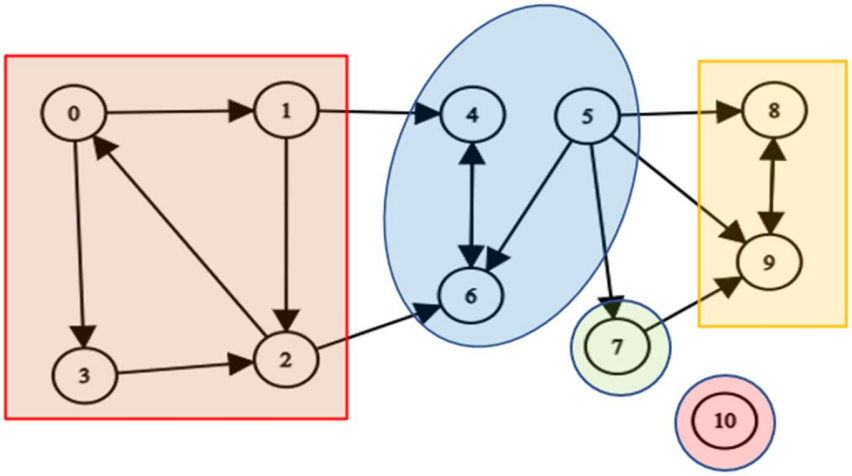
Tarjan's Algorithm is used to find strongly connected components of a directed graph. It requires only one DFS traversal to implement this algorithm**.** Using DFS traversal we can find DFS tree of the forest. From the DFS tree, strongly connected components are found. When the root of such sub-tree is found we can display the whole subtree. That subtree is one strongly connected component.

Pseudocode **-**

- Perform a DFS traversal over the nodes so that the sub-trees of the Strongly Connected Components are removed when they are encountered.
- Then two values are assigned:
- The first value(disk) is the counter value when the node is explored for the first time.
- Second value(low) stores the lowest node value reachable from the initial node which is not part of another SCC.
- When the nodes are explored, they are pushed into a stack.
- If there are any unexplored children of a node are left, they are explored and the assigned value is respectively updated.

- A node is encountered with low(u) == disk(u) is the first explored node in its strongly connected component and all the nodes above it in the stack are popped out and assigned the appropriate SCC number.

Example Graph-



Code-

```
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;

class Graph
{
    int V;
    list<int> *adj;
    void SCCUtil(int u, int disc[], int low[],
                 stack<int> *st, bool stackMember[]);
public:
    Graph(int V);
    void addEdge(int v, int w);
    void SCC();
};

Graph::Graph(int V)
```

```
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::SCCUtil(int u, int disc[], int low[], stack<int> *st,
                    bool stackMember[])
{
    static int time = 0;
    disc[u] = low[u] = ++time;
    st->push(u);
    stackMember[u] = true;
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;
        if (disc[v] == -1)
        {
            SCCUtil(v, disc, low, st, stackMember);

            low[u] = min(low[u], low[v]);
        }
        else if (stackMember[v] == true)
            low[u] = min(low[u], disc[v]);
    }

    int w = 0;
    if (low[u] == disc[u])
    {
        while (st->top() != u)
        {
            w = (int) st->top();
            cout << w << " ";
            stackMember[w] = false;
            st->pop();
        }
        w = (int) st->top();
        cout << w << "\n";
        stackMember[w] = false;
        st->pop();
```

```
    }
}

void Graph::SCC()
{
    int *disc = new int[V];
    int *low = new int[V];
    bool *stackMember = new bool[V];
    stack<int> *st = new stack<int>();
    for (int i = 0; i < V; i++)
    {
        disc[i] = NIL;
        low[i] = NIL;
        stackMember[i] = false;
    }
    for (int i = 0; i < V; i++)
        if (disc[i] == NIL)
            SCCUtil(i, disc, low, st, stackMember);
}
int main()
{
    cout << "\nSCCs in the graph \n";
    Graph g(11);
    g.addEdge(0,1);
    g.addEdge(0,3);
    g.addEdge(1,2);
    g.addEdge(1,4);
    g.addEdge(2,0);
    g.addEdge(2,6);
    g.addEdge(3,2);
    g.addEdge(4,5);
    g.addEdge(4,6);
    g.addEdge(5,6);
    g.addEdge(5,7);
    g.addEdge(5,8);
    g.addEdge(5,9);
    g.addEdge(6,4);
    g.addEdge(7,9);
    g.addEdge(8,9);
    g.addEdge(9,8);
    g.SCC();

    return 0;
}
```

Output-

```
SCCs in the graph
8 9
7
5 4 6
3 2 1 0
10
```

Time Complexity of Tarjan's Algorithm – **O(|V|+|E|)**

Applications –

- To converted the graph into a Directed Acyclic Graph of strongly connected components
- For solving 2 SAT problems, where the problem is unsatisfiable if both a variable and its complement lie in the same SCC.

Algorithm 3 – Cheriyan-Mehlhorn-Gabow Algorithm

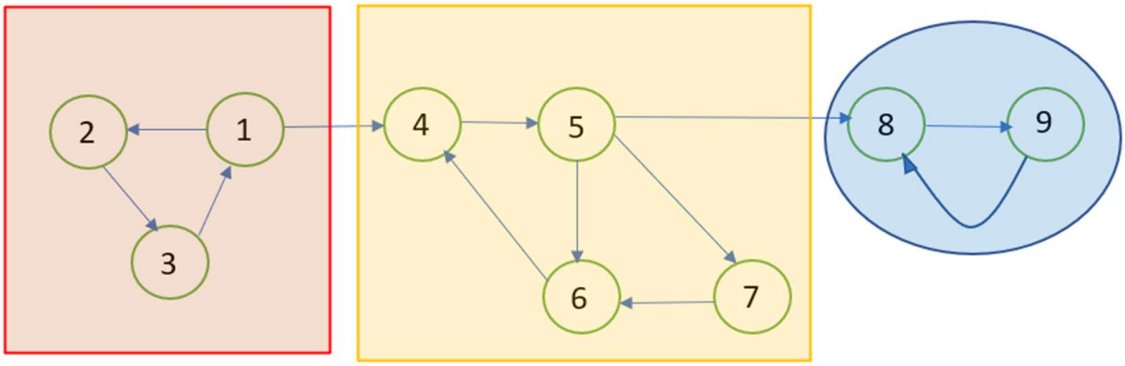
Cheriyan-Mehlhorn-Gabow Algorithm is a linear time algorithm used to find the strongly connected components of a directed graph.
It also uses depth first search to explore all the nodes of the directed graph. Gabow algorithm maintains two stacks, one of them contains a list of nodes not yet computed as as strongly connected components and other contains a set of nodes not belong to different strongly connected components. A counter is used to count number of visited.

Pseudocode **-**

- Lest S and B are empty stacks.
- Set the pre-order number v to C, and increment C.
- Push v on S and B.
- For each edge v →u:
- If pre-order number of u has not been assigned:
- Start DFS(u).
- Else if u has not yet been assigned to a scc.
- Repeatedly pop vertices from B until the top element has a pre-order number less than or equal to pre-order number of u.
- If v is the top element of B.
- Pop vertices from S until v has been popped and assign the popped vertices to a new component.

- Pop v from B

Example Graph-



Code-

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

int adjacencyTable[100][100] = {0};
int visited[100] = {0};
int isInStack[100] = {0};
int id = 0;
stack<int> st1, st2;
vector<vector<int>> ans;
int n = 9, m = 12;
int E[12][2] = {{1, 2},{2, 3},{3, 1},{1, 4},{4, 5},{5, 6},{5, 7},{7, 6},{6,
4},{5, 8},{8, 9},{9, 8},};

void dfs(int num, vector<int>& vec){
    id++;
    visited[num] = id;
    st1.push(num);
    st2.push(num);
    int node;
    for(int i = 1; i <= adjacencyTable[num][0]; i++){
        node = adjacencyTable[num][i];
        if(visited[node] == 0){
```

```
            dfs(node, vec);
        }else if(isInStack[node]==0){
            while(visited[st2.top()] > visited[node]){
                st2.pop();
            }
        }
    }
    if(num == st2.top()){
        while(num != st1.top()){
            vec.emplace_back(st1.top());
            isInStack[st1.top()] = 1;
            st1.pop();
        }
        vec.emplace_back(num);
        isInStack[num] = 1;
        st1.pop();
        st2.pop();
        ans.emplace_back(vec);
        vec = {};
    }
}

int main()
{
    for(int i = 0; i < m; i++){
        int from = E[i][0], to = E[i][1];
        adjacencyTable[from][++adjacencyTable[from][0]] = to;
    }
    for(int i = 1; i <= n; i++){
        if(visited[i]==0){
            vector<int> vec;
            dfs(i, vec);
        }
    }
    cout<<"SCC in this graph: "<<endl;
    for(auto item : ans){
        for(int number :item){
            cout<<number<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Output-

```
SCC in this graph:
9 8
7 6 5 4
3 2 1
```

Time Complexity of Cheiyan-Mehlhorn-Gabow Algorithm – **O(|V|+|E|)**

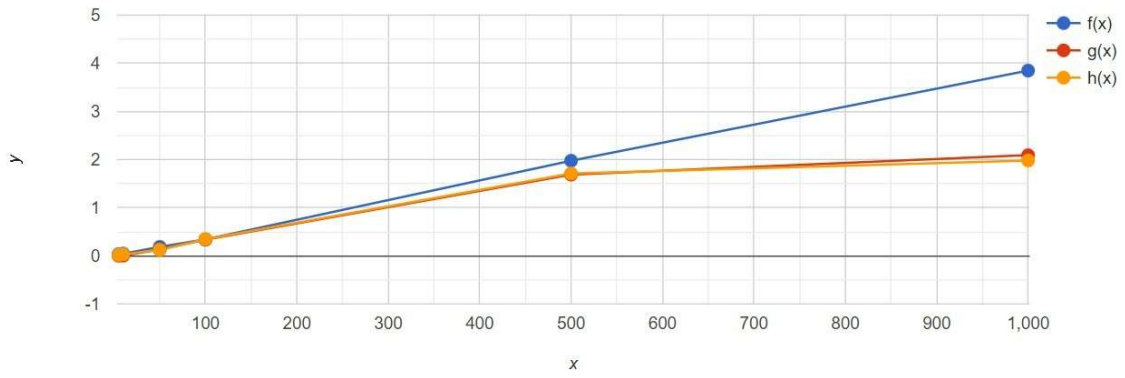
Applications –

- Maps
- Model-checking in formal verification

Performance analysis

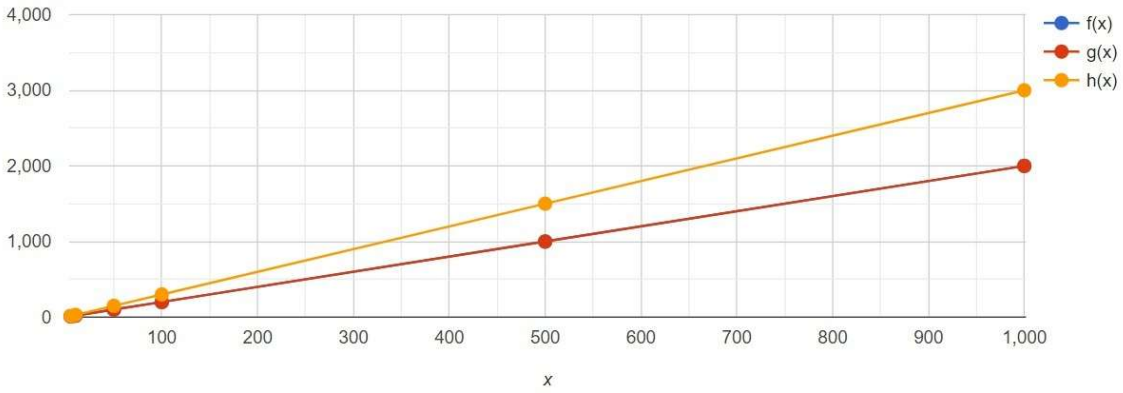
Time complexity-

No of inputs	Kosaraju	Tarjans	Gabow
5	3.4 x 10^-5	2x10^-5	1.92 x 10^-5
10	4.6x10^-5	3x10^-5	3.12 x 10^-5
50	1.85 x 10^ -4	1.32 x 10^-4	1.25 x 10^-4
100	3.4 x 10^-4	3.36 x 10^-4	3.46 x 10^-4
500	1.975 x 10^-3	1.687 x 10^-3	1.71 x 10^-3
1000	3.848 x 10^-3	2.09 x 10^-3	1.98 x 10^-3



Space Complexity

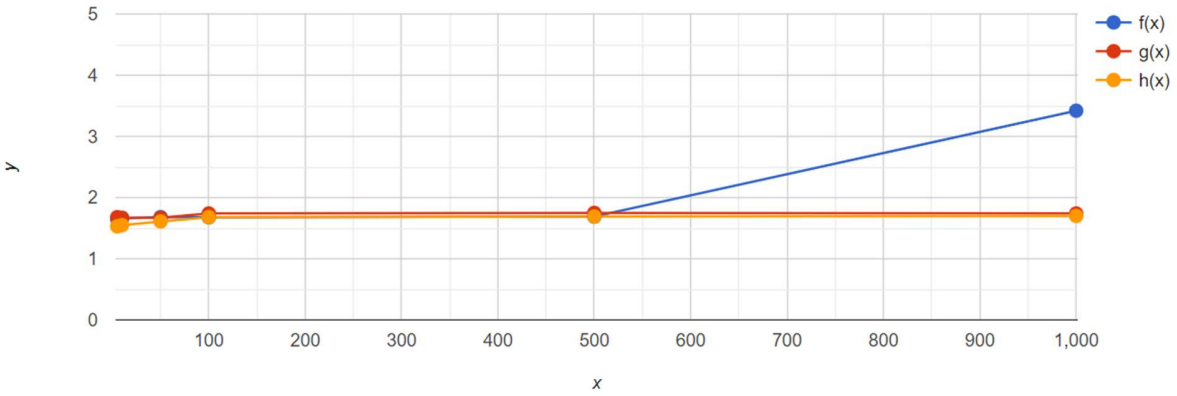
Inputs	Kosaraju	Tarjan	Gabow
5	10	10	15
10	20	20	30
50	100	100	150
100	200	200	300
500	1000	1000	1500



Memory utilization

No of inputs	Kosaraju	Tarjasn	Gabow
5	1.65 MB	1.68 MB	1.53
10	1.66 MB	1.67 MB	1.55
50	1.68 MB	1.67 MB	1.61
100	1.68 MB	1.74MB	1.68

500	1.69 MB	1.75 MB	1.69
1000	3.42 MB	1.74 MB	1.7



No of DFS Traversals

No of DFS Traversals	Kosaraju	Tarjans	Gabow
5	2	1	1
10	2	1	1

50		2	1	1
100		2	1	1
500		2	1	1
1000		2	1	1

Kosaraju's algorithm takes 2 DFS.
Tarjan's and Gabow's algorithm both take 1 DFS cycle each.

Numerical Stability

Tarjan and Gabow more stable than Kosaraju for higher values of input