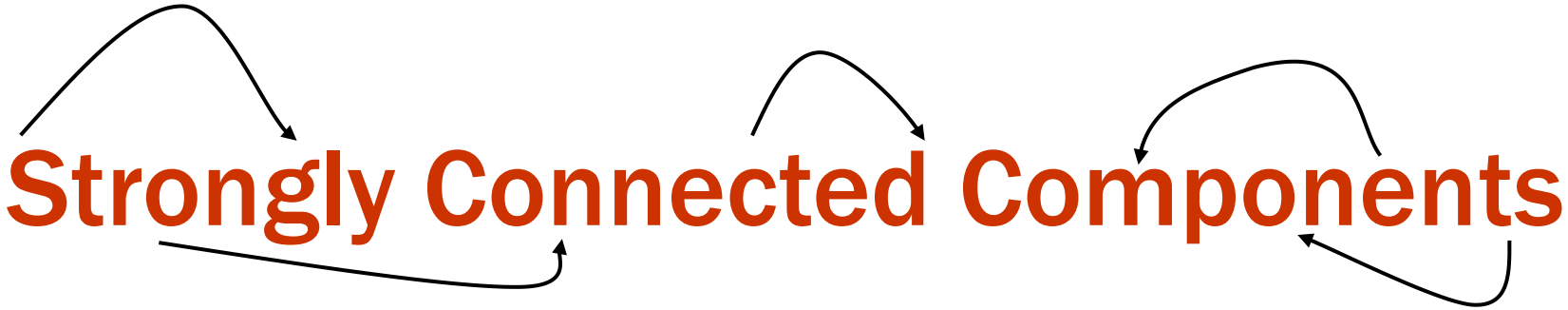


Strongly Connected Components

IOI Training Camp 2 – 2021/22

Kenna Geleta

EXAMPLE PROBLEM

CSES 1683 : Planets and Kingdoms

Time limit: 1.00 s Memory limit: 512 MB

A game has n planets, connected by m teleporters. Two planets a and b belong to the same kingdom exactly when there is a route both from a to b and from b to a. Your task is to determine for each planet its kingdom.

EXAMPLE PROBLEM

CSES 1683 : Planets and Kingdoms

Time limit: 1.00 s Memory limit: 512 MB

A game has n planets, connected by m teleporters. Two planets a and b belong to the same kingdom exactly when there is a route both from a to b and from b to a. Your task is to determine for each planet its kingdom.

BRUTE FORCE?

EXAMPLE PROBLEM

CSES 1683 : Planets and Kingdoms

Time limit: 1.00 s Memory limit: 512 MB

A game has n planets, connected by m teleporters. Two planets a and b belong to the same kingdom exactly when there is a route both from a to b and from b to a. Your task is to determine for each planet its kingdom.

DEFINITION

A graph is **strongly connected** when a path exists from every node to every other node.

A **strongly connected component** is a subset of nodes in a graph where a path exists from every node to every other node.

The strongly connected components form an acyclic **component graph.**

EXAMPLE

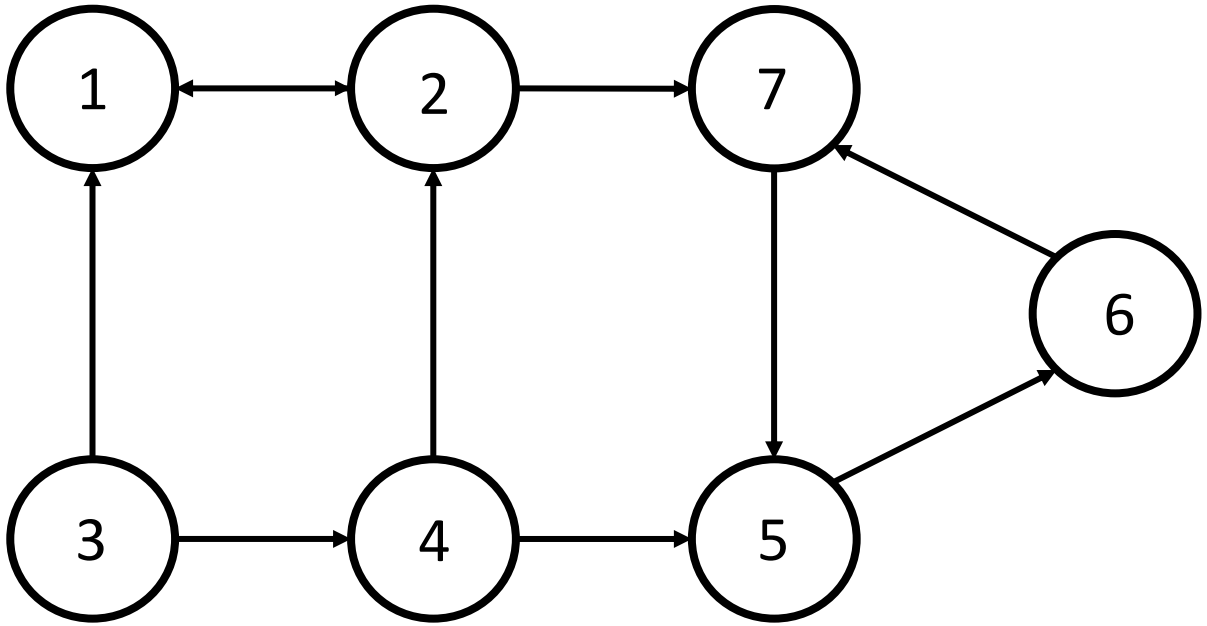


Figure: Directed Graph

EXAMPLE

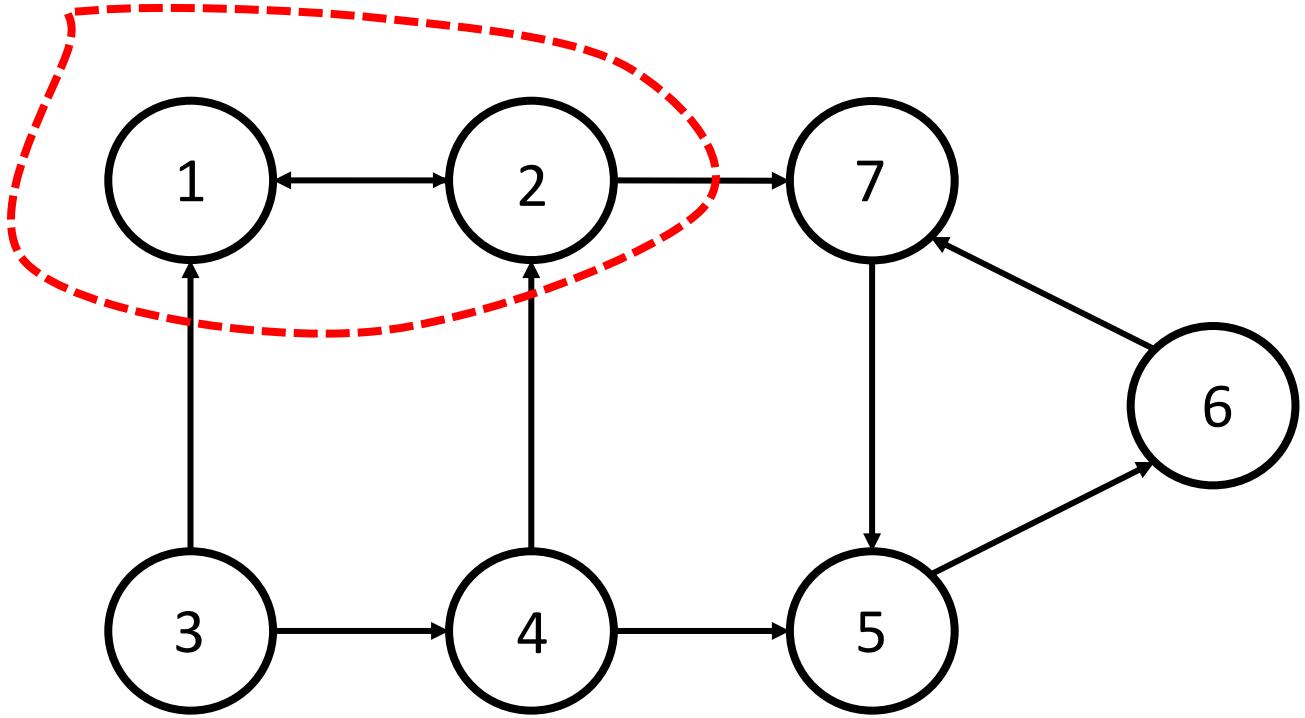


Figure: Directed Graph

EXAMPLE

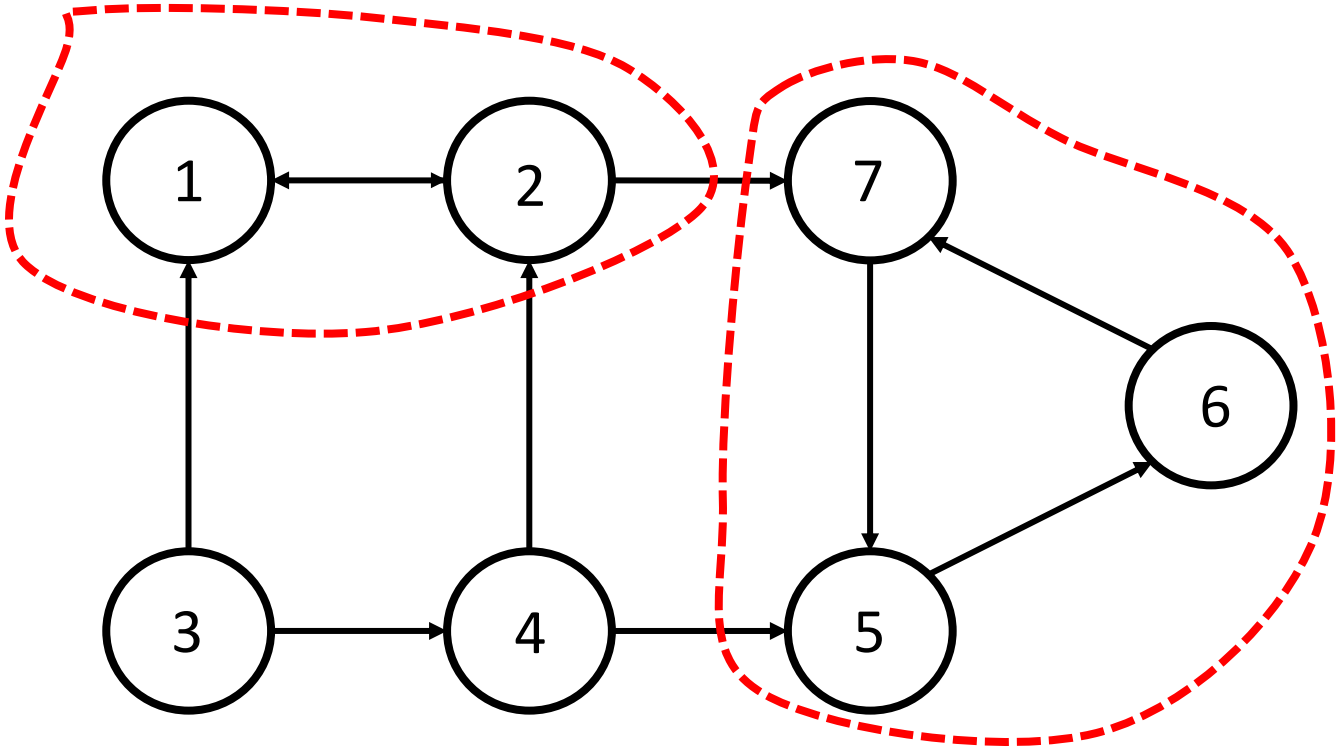


Figure: Directed Graph

EXAMPLE

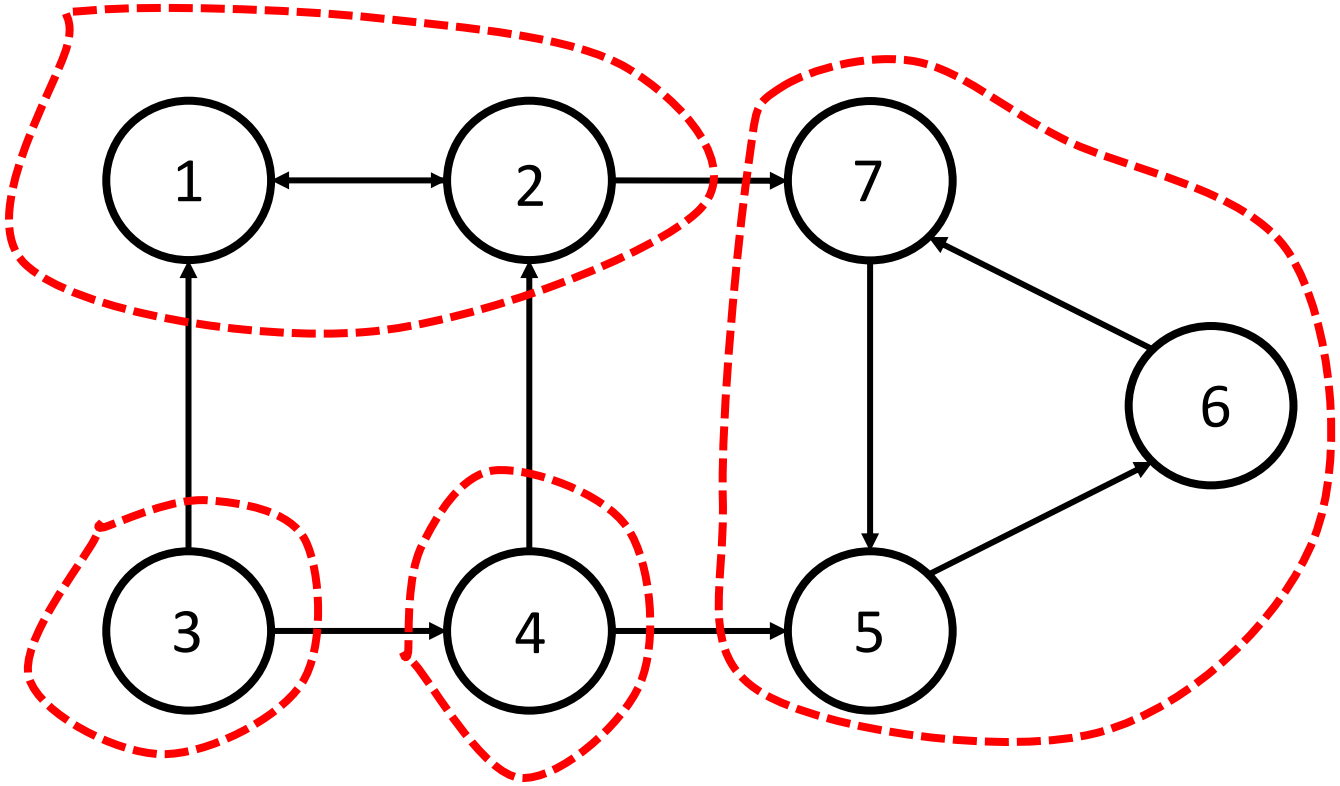


Figure: Directed Graph


KOSARAJU'S
DOUBLE DFS GAMBIT

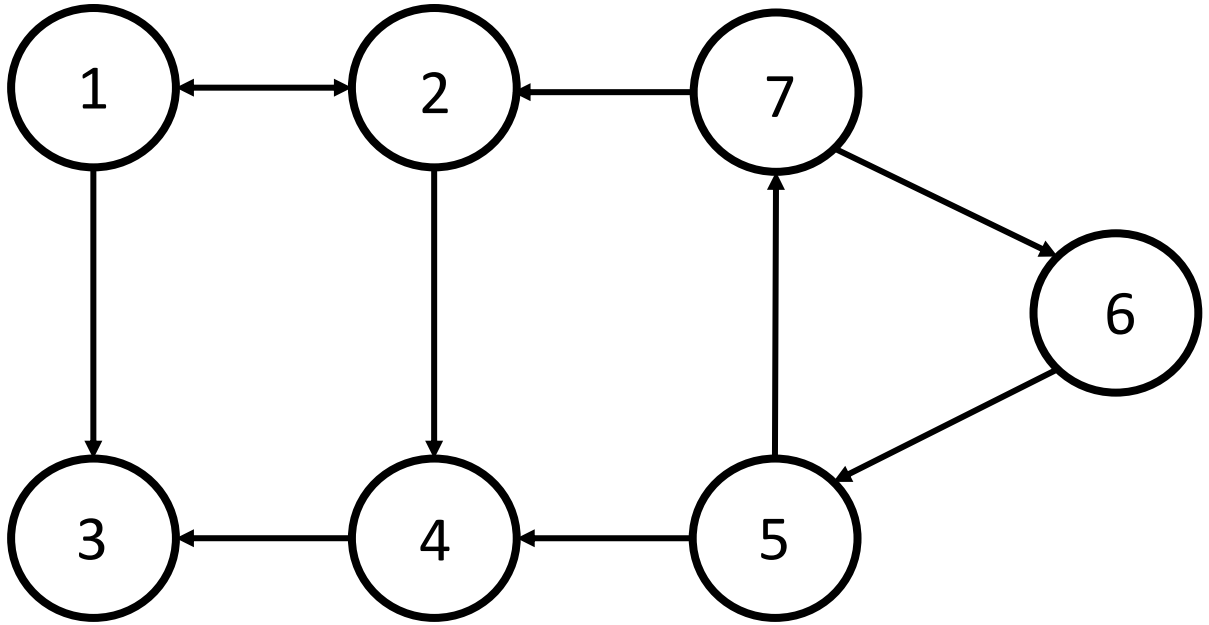
KOSARAJU'S ALGORITHM

(in pseudo)

1. Construct adjacency list
2. Perform DFS
    1. Flag entry time
    2. Push to children
    3. Flag exit time
    4. Add node timing object to list
3. Order list by descending exit time
4. Reverse all edges in the graph
5. Perform DFS from first list element
    1. Push nodes to component lists

EXAMPLE:

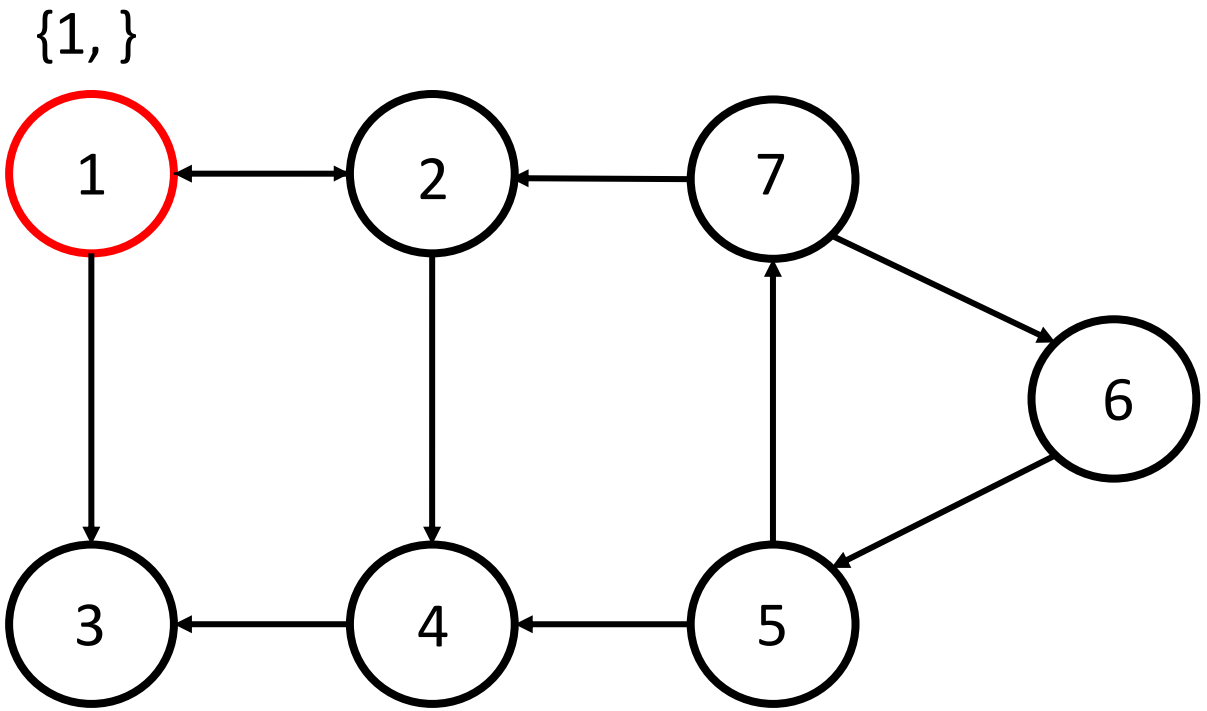
1. construct adjacency list



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

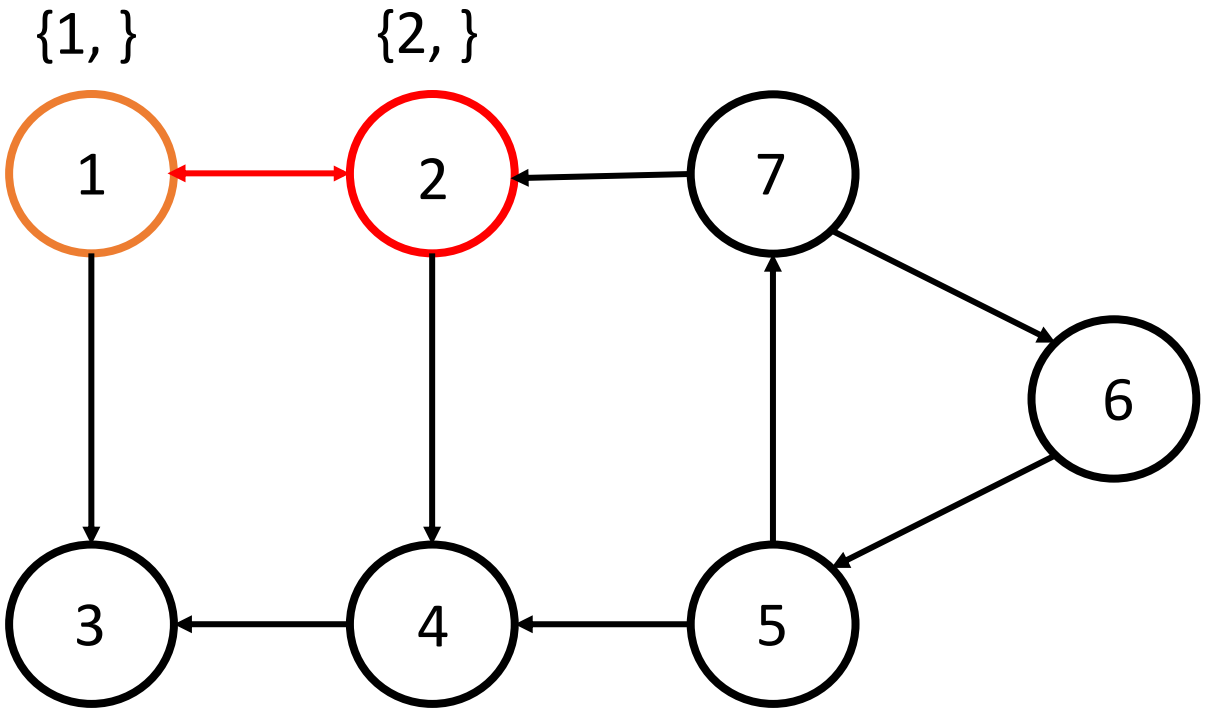
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

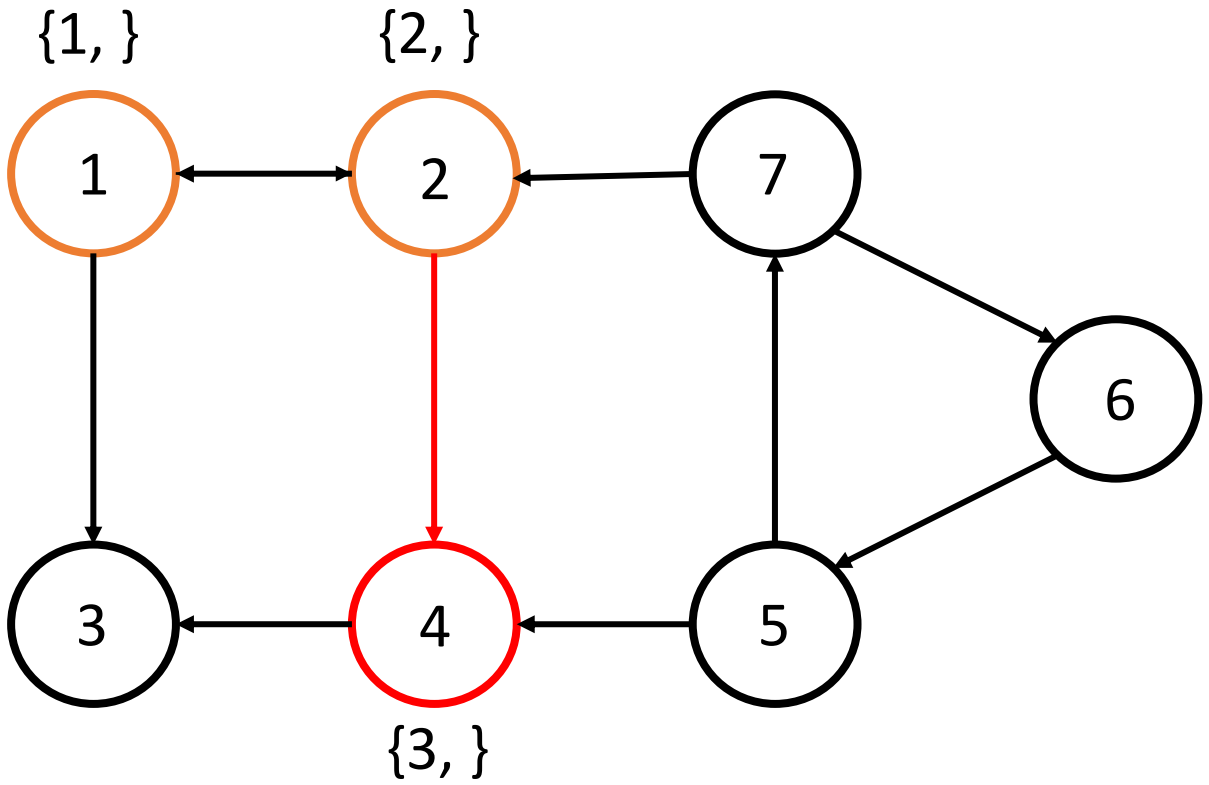
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

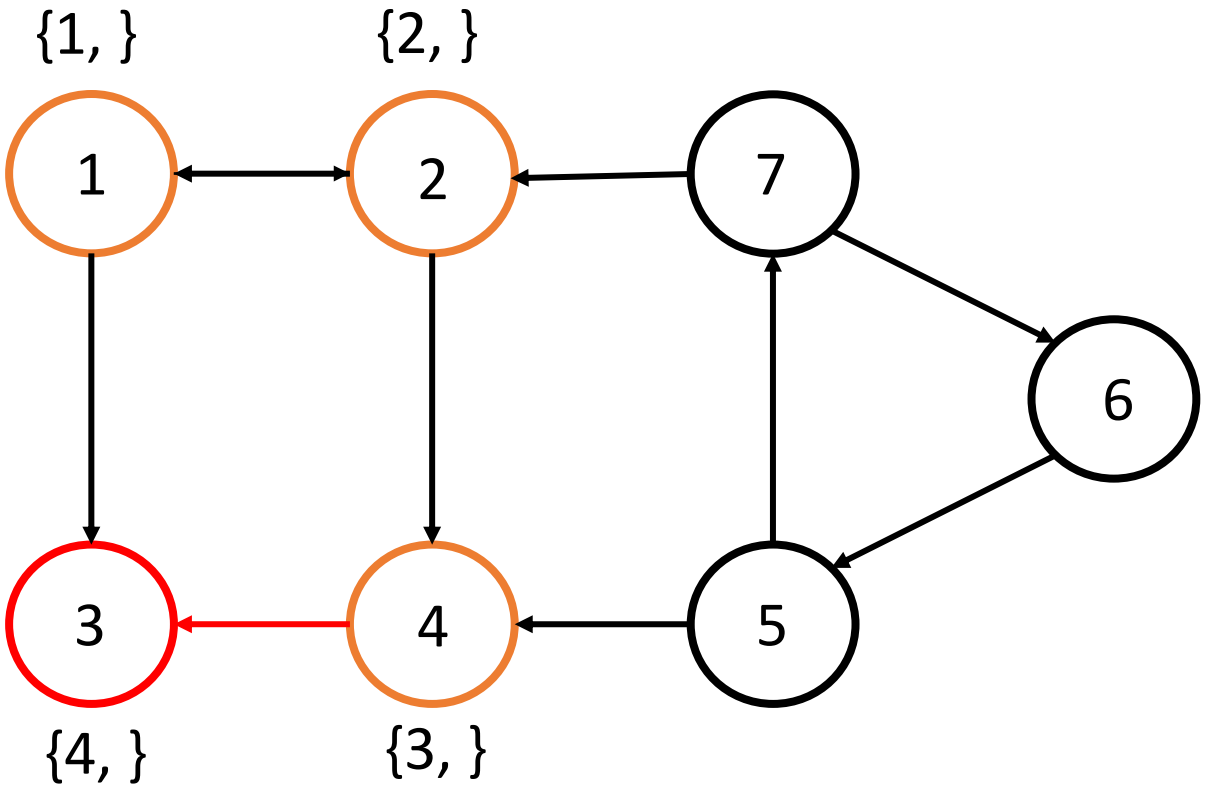
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

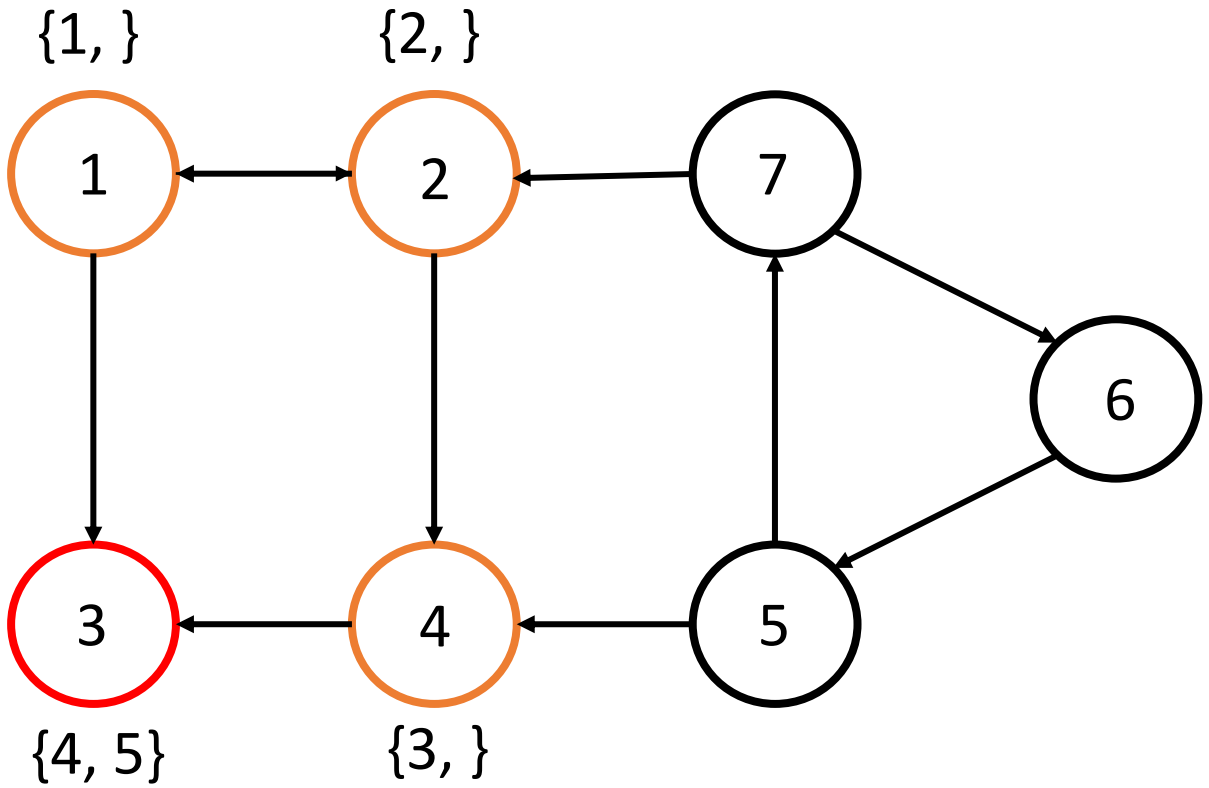
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

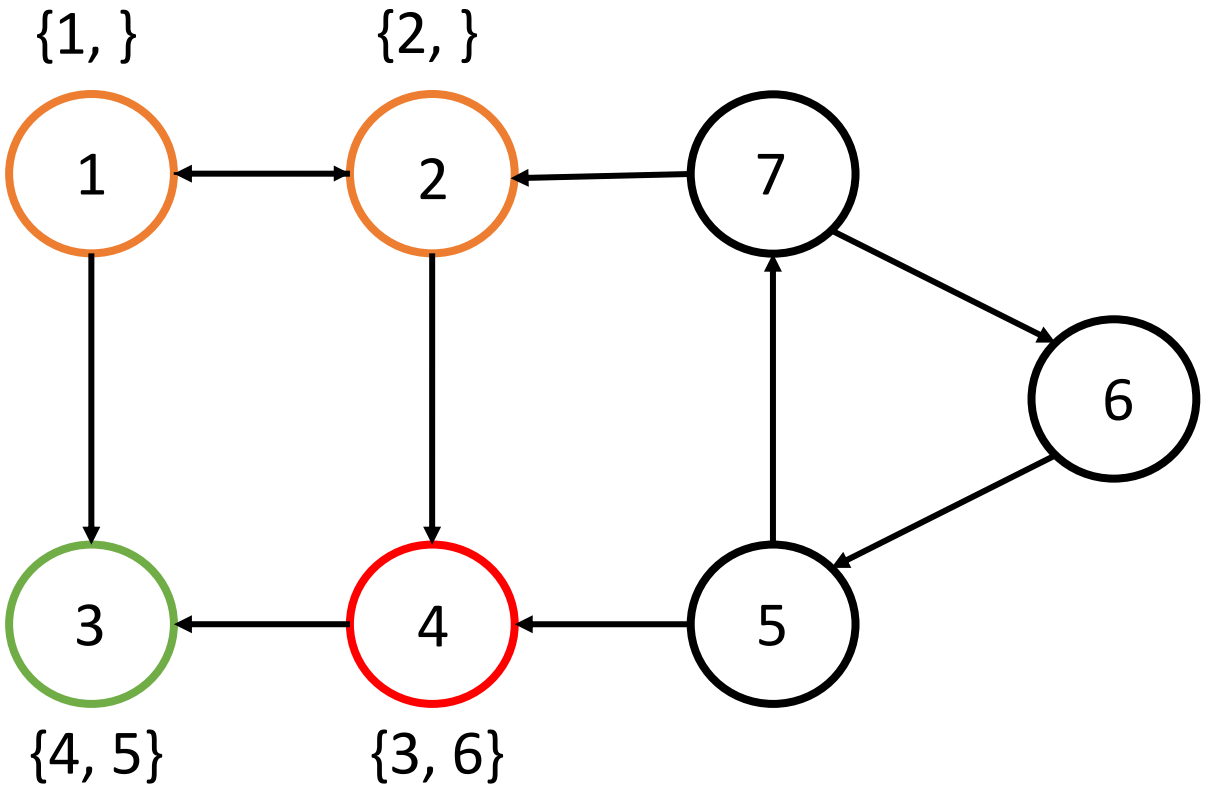
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

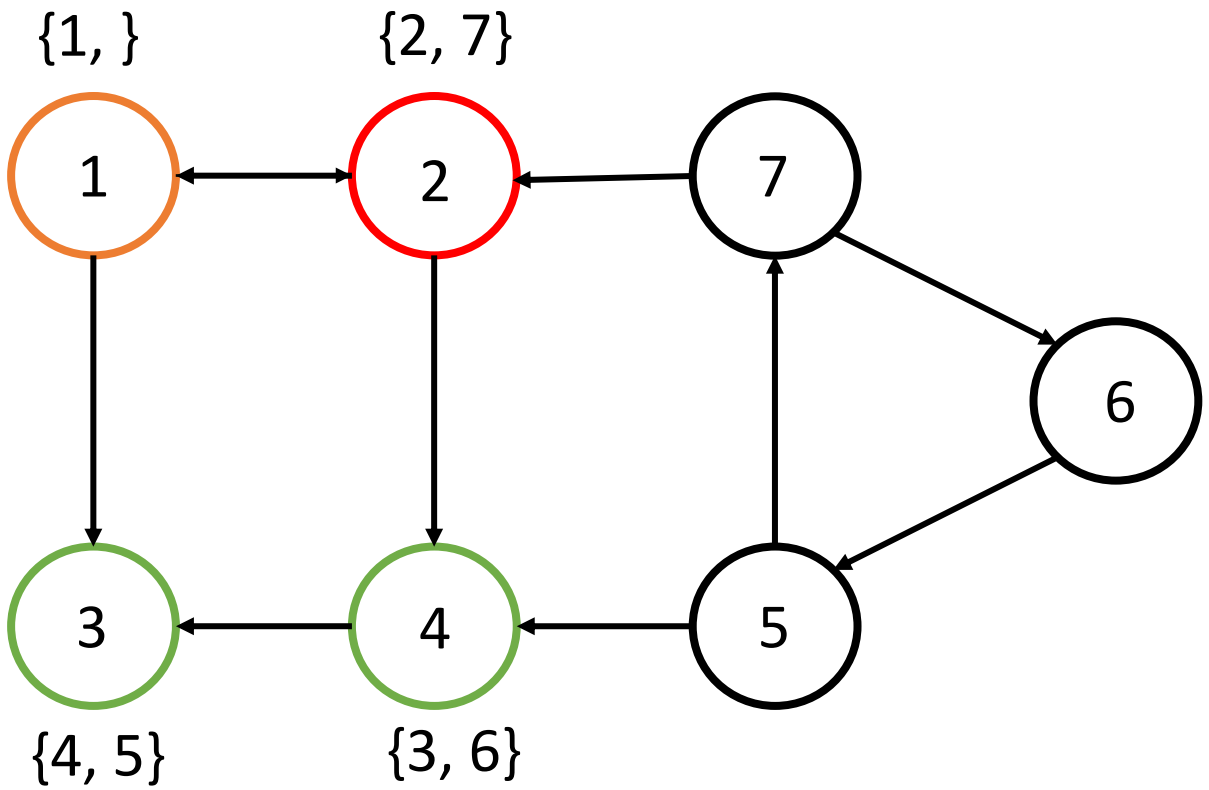
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

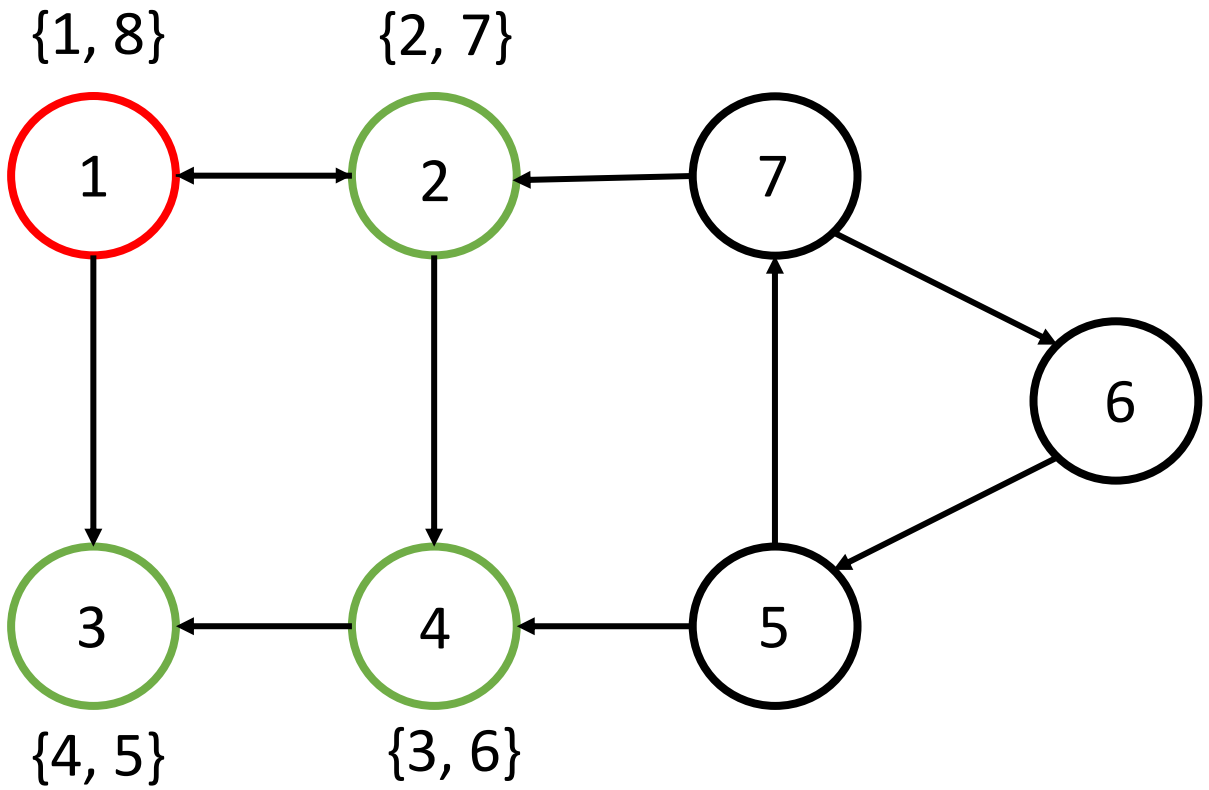
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

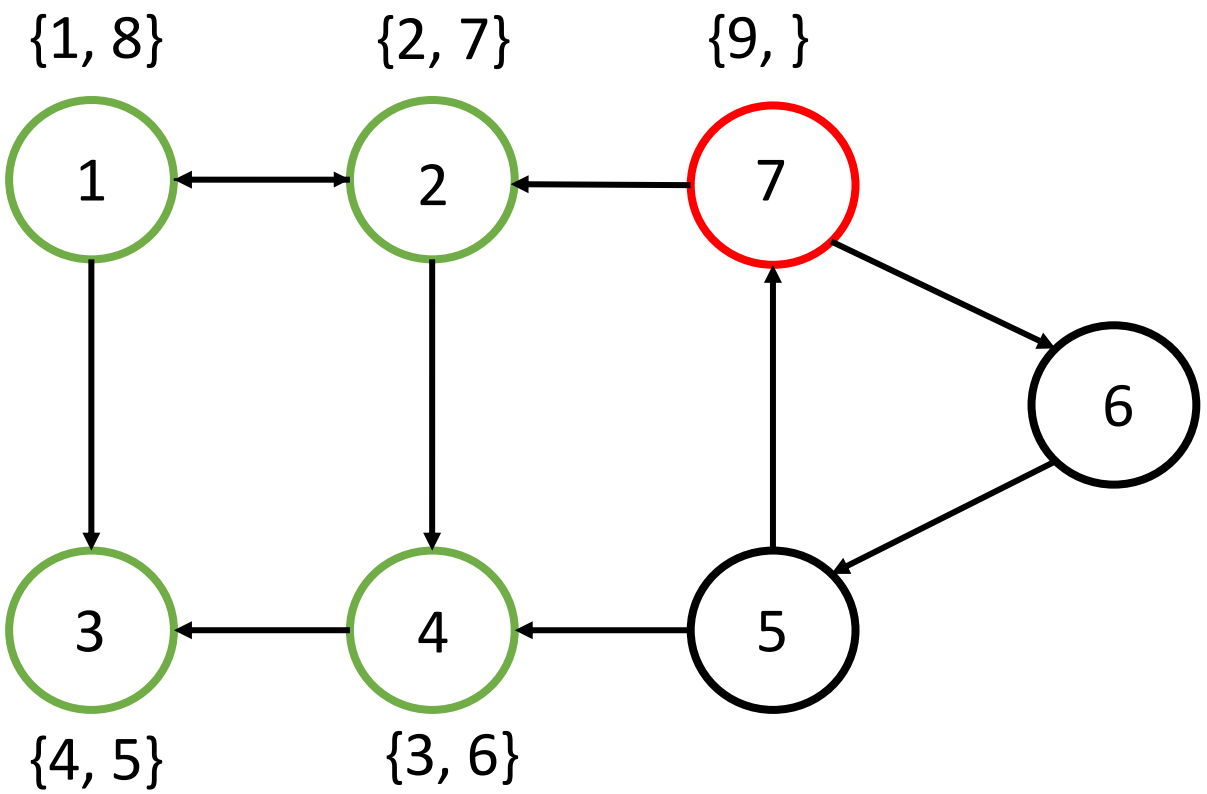
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

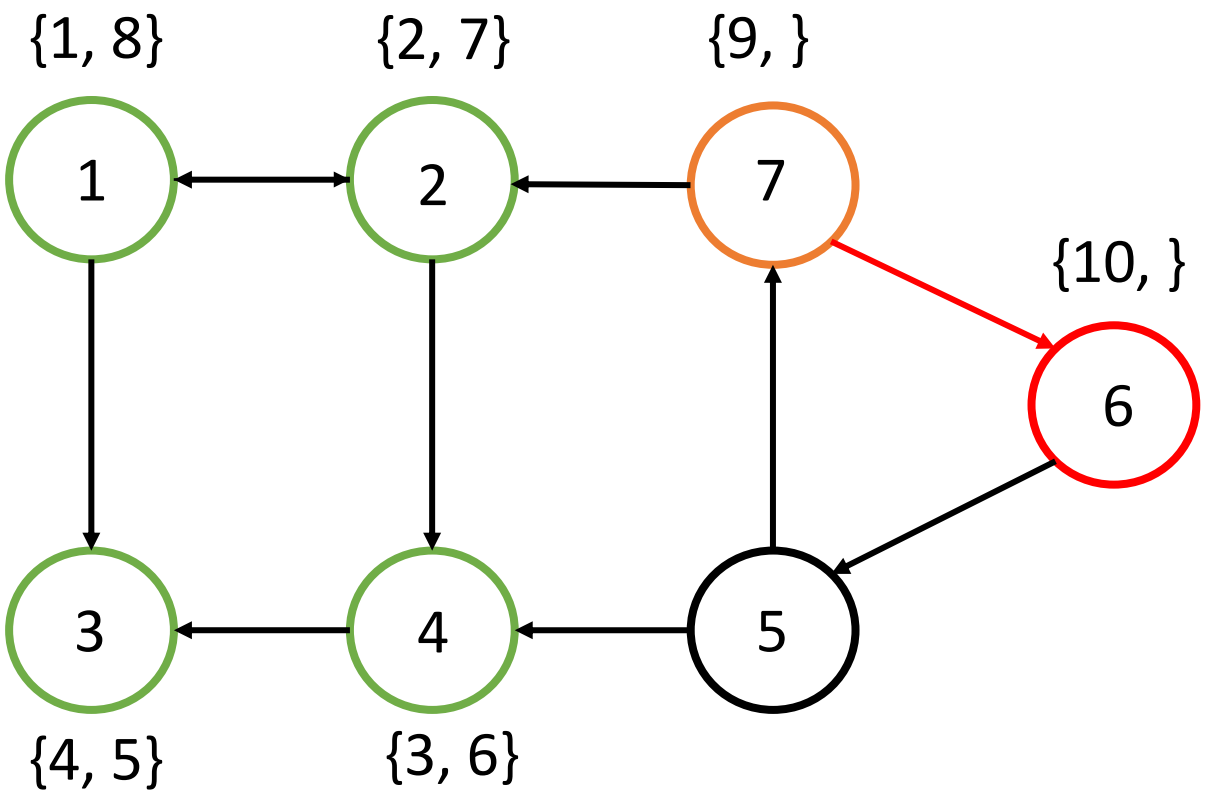
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

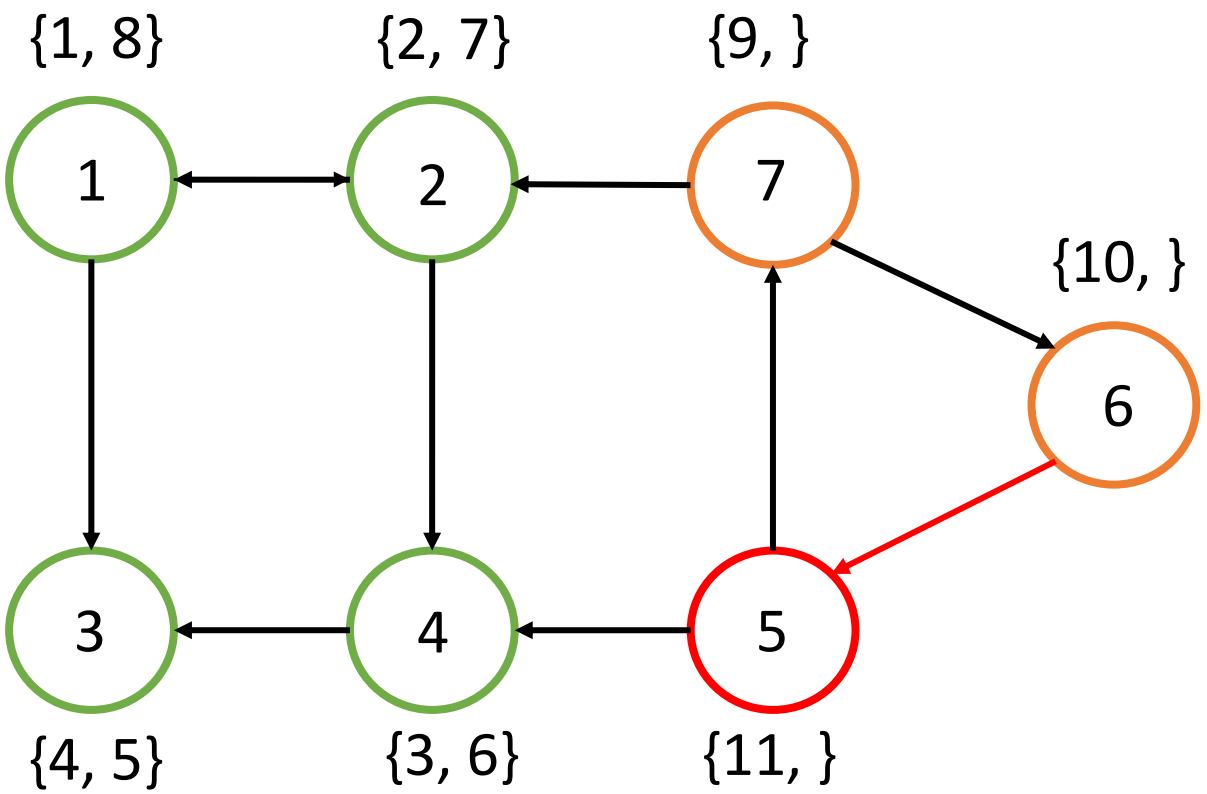
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

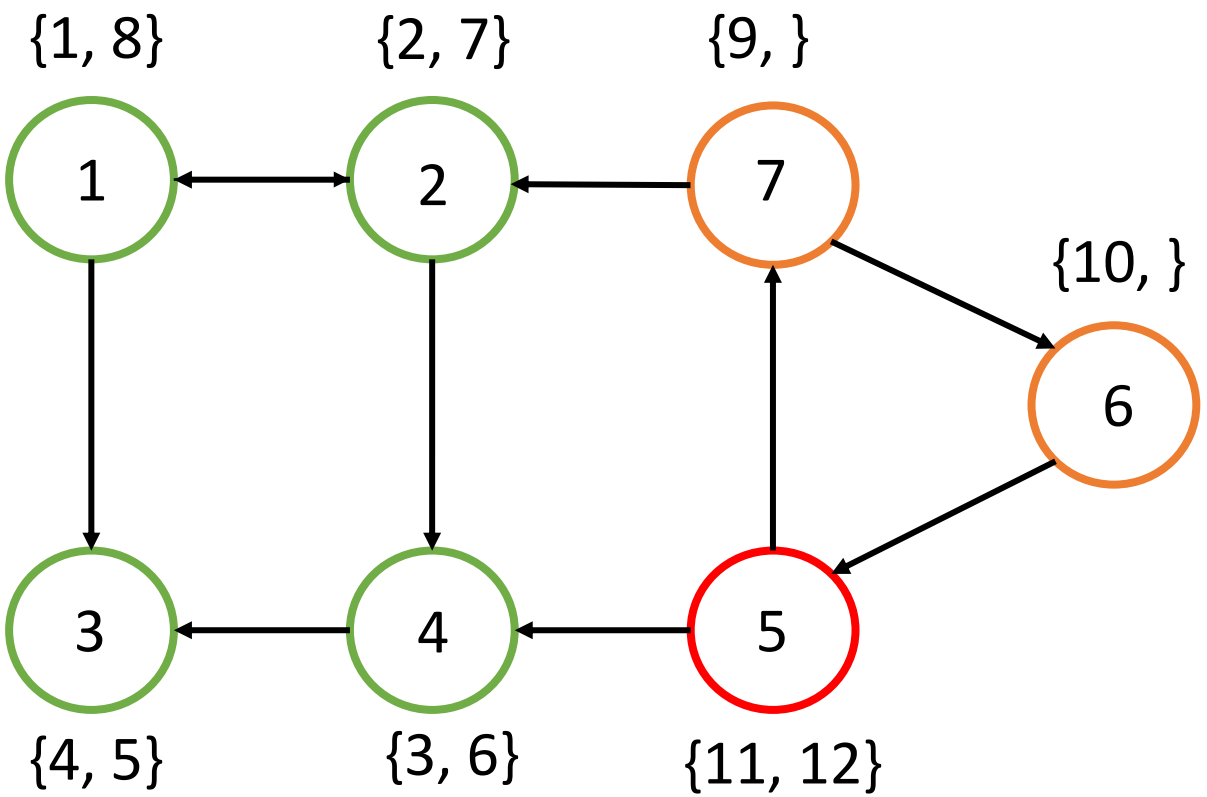
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

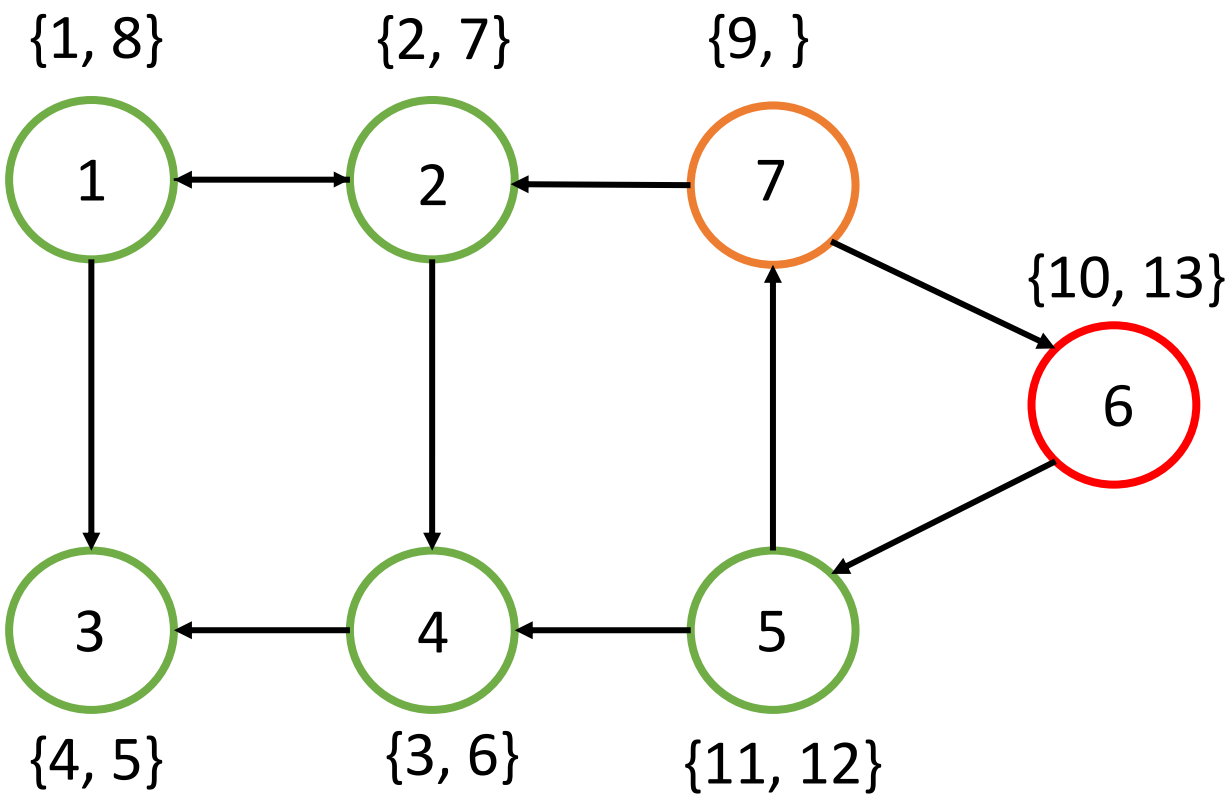
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

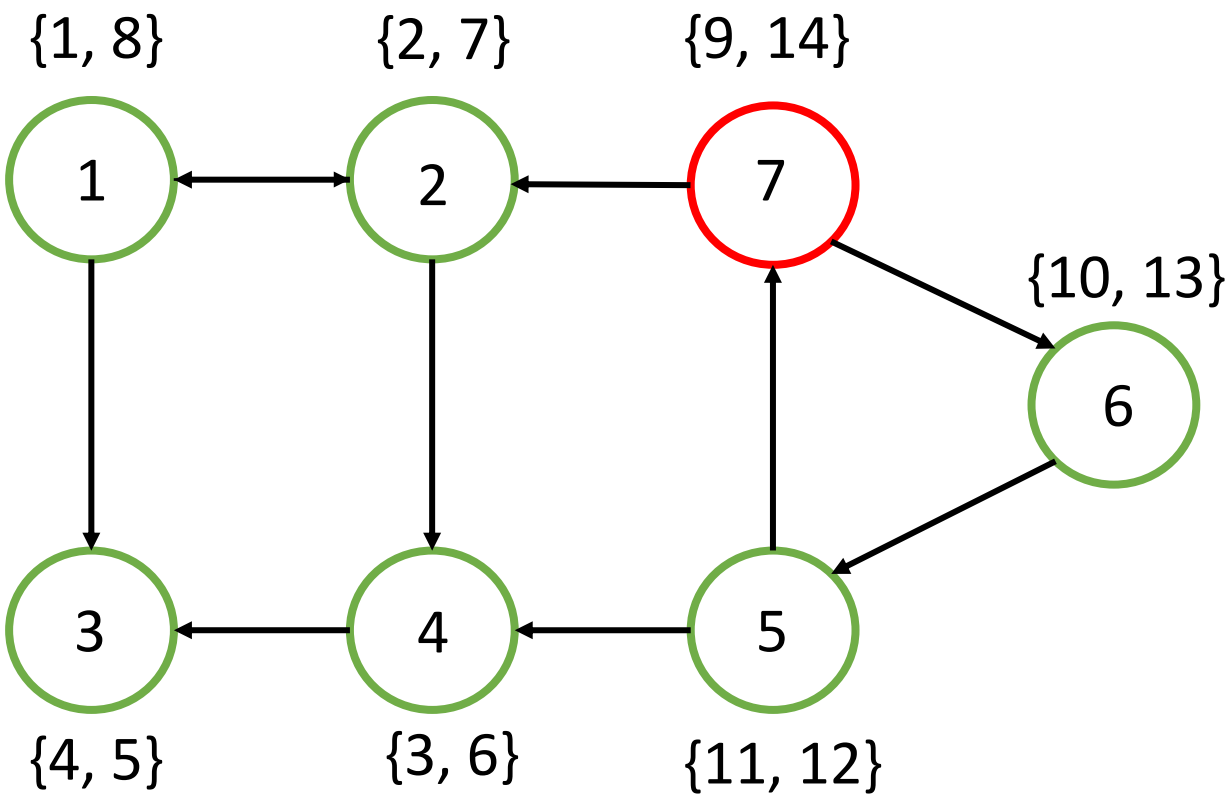
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

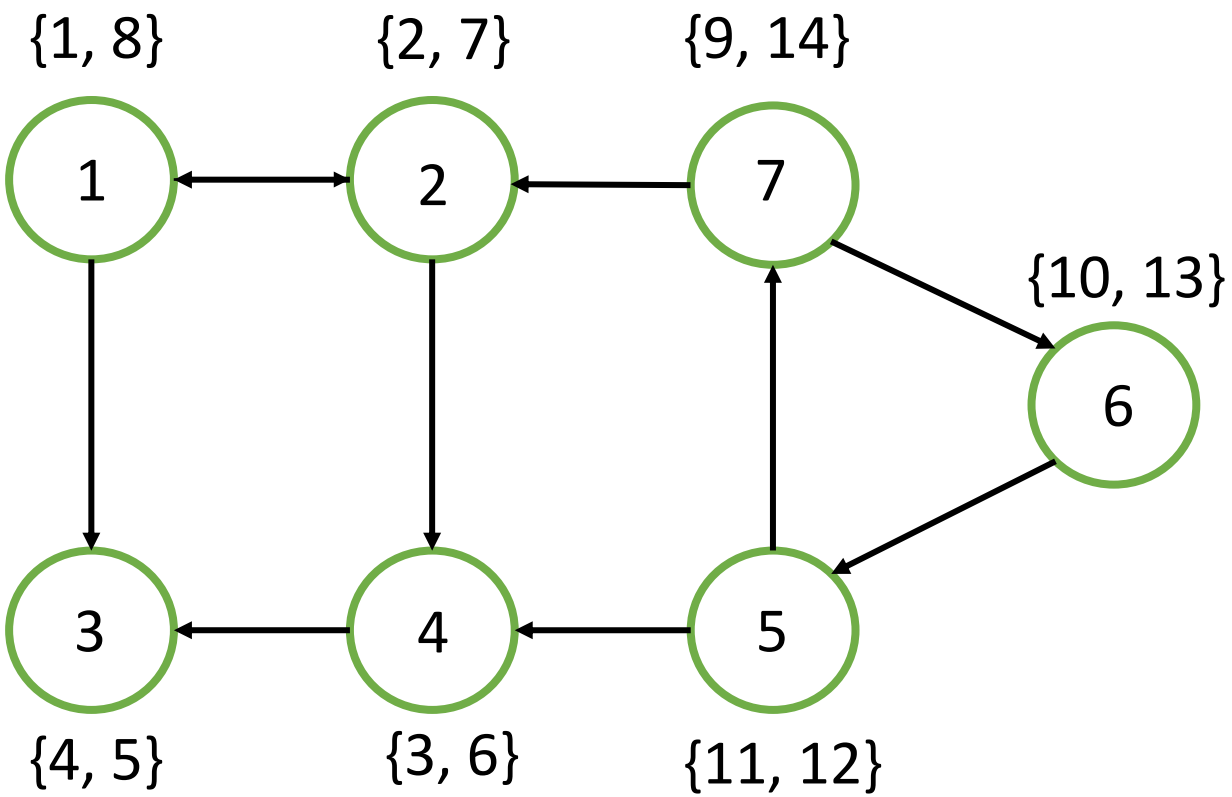
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

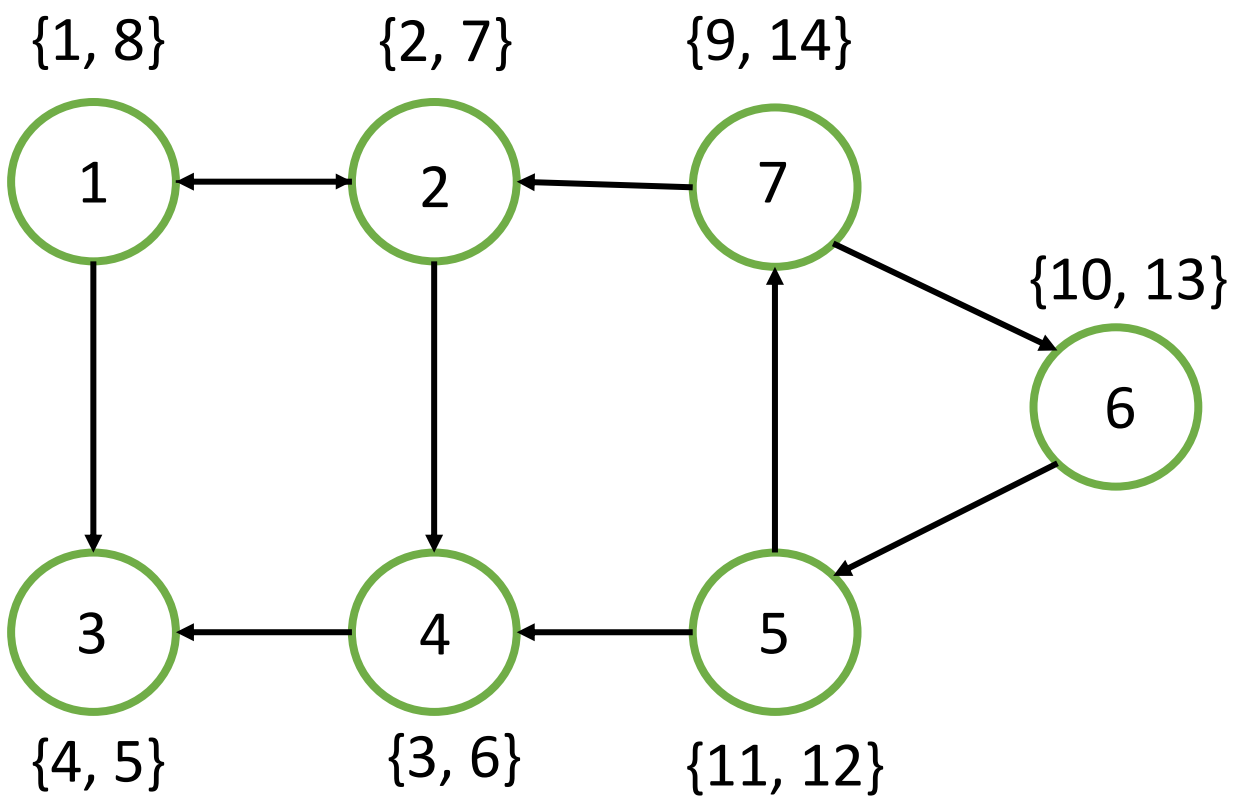
2. perform dfs



NODE	CHILDREN
1	{2,3}
2	{1,4}
3	{}
4	{3}
5	{4,7}
6	{5}
7	{2,6}

EXAMPLE:

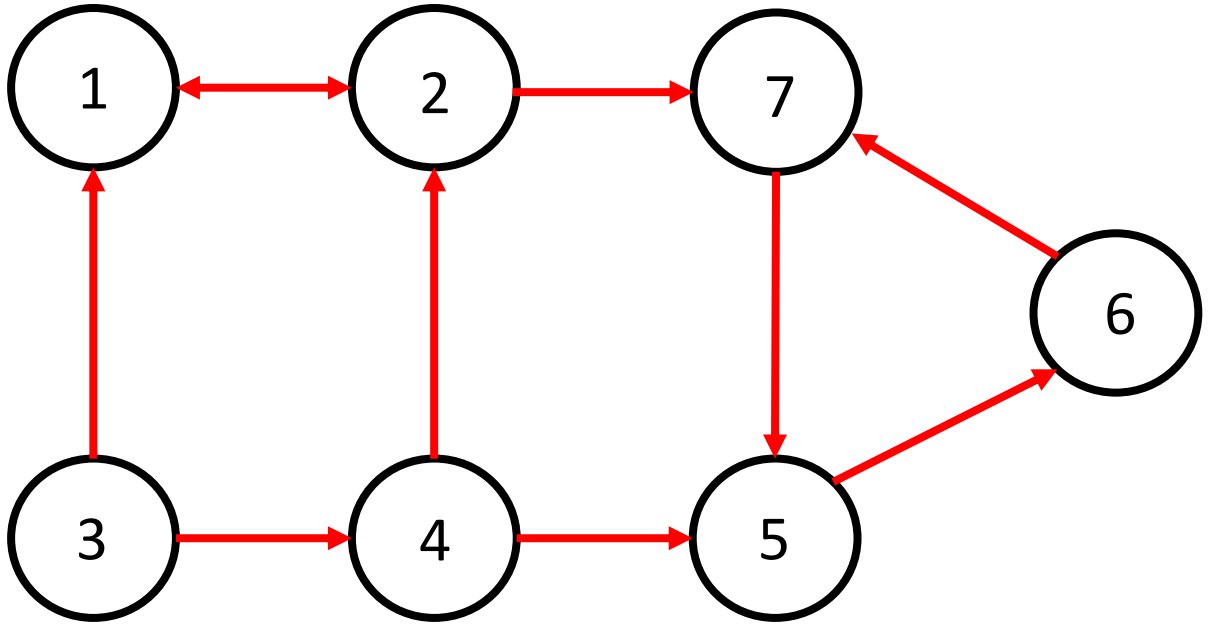
3. order list by descending time



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

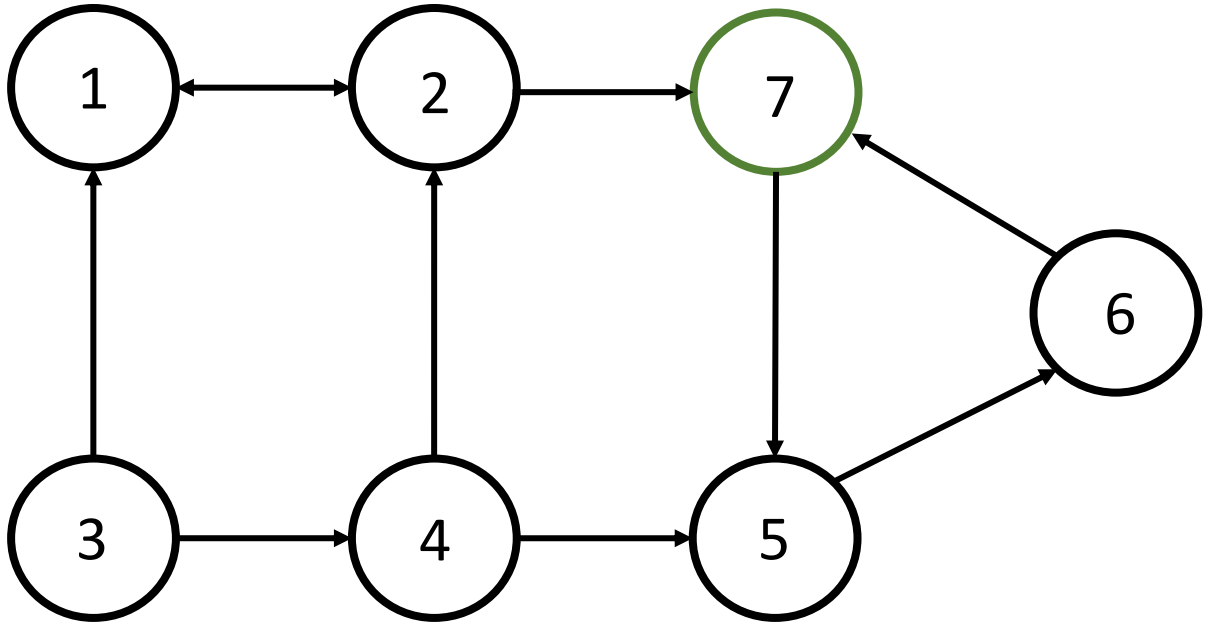
4. REVERSE EDGES



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

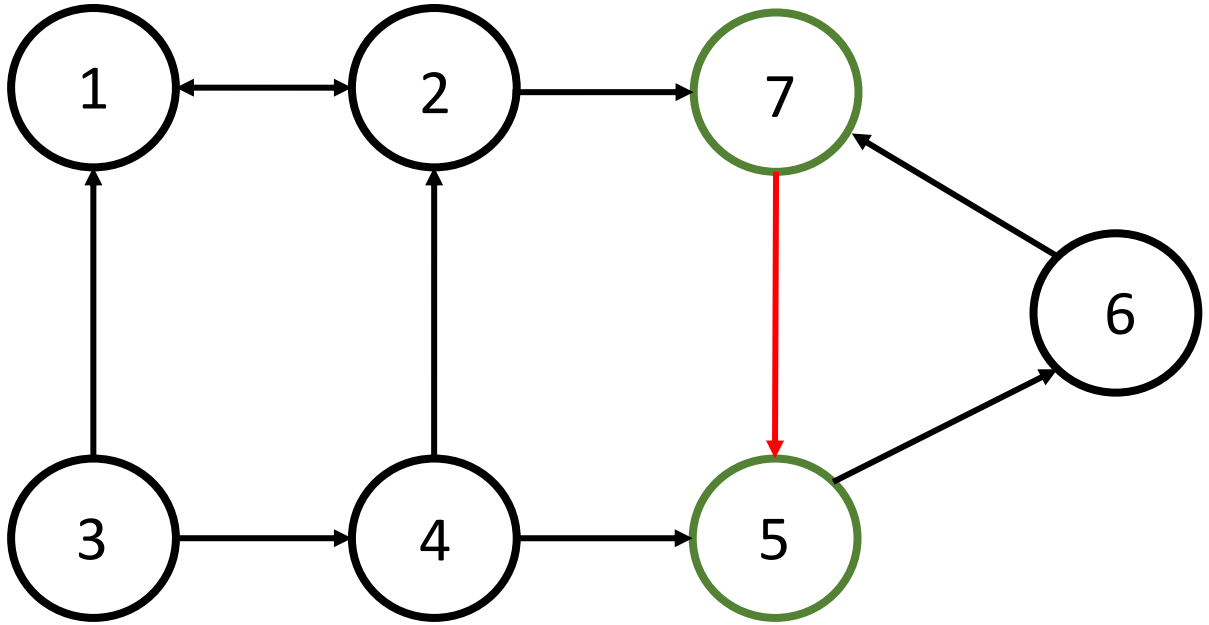
5. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

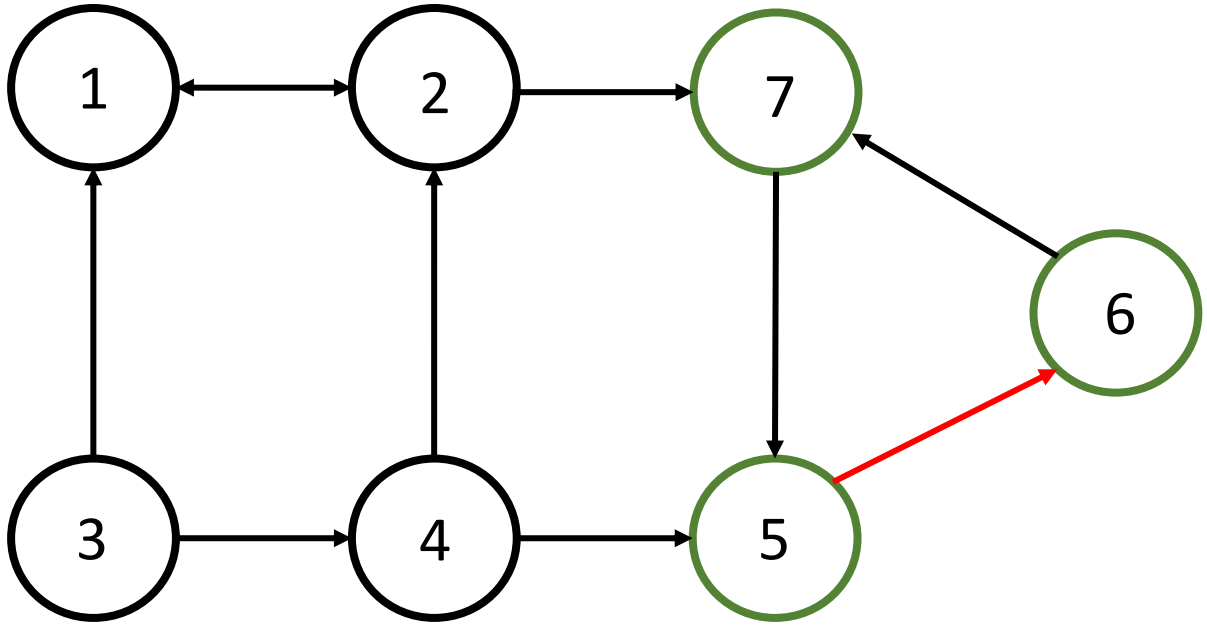
4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

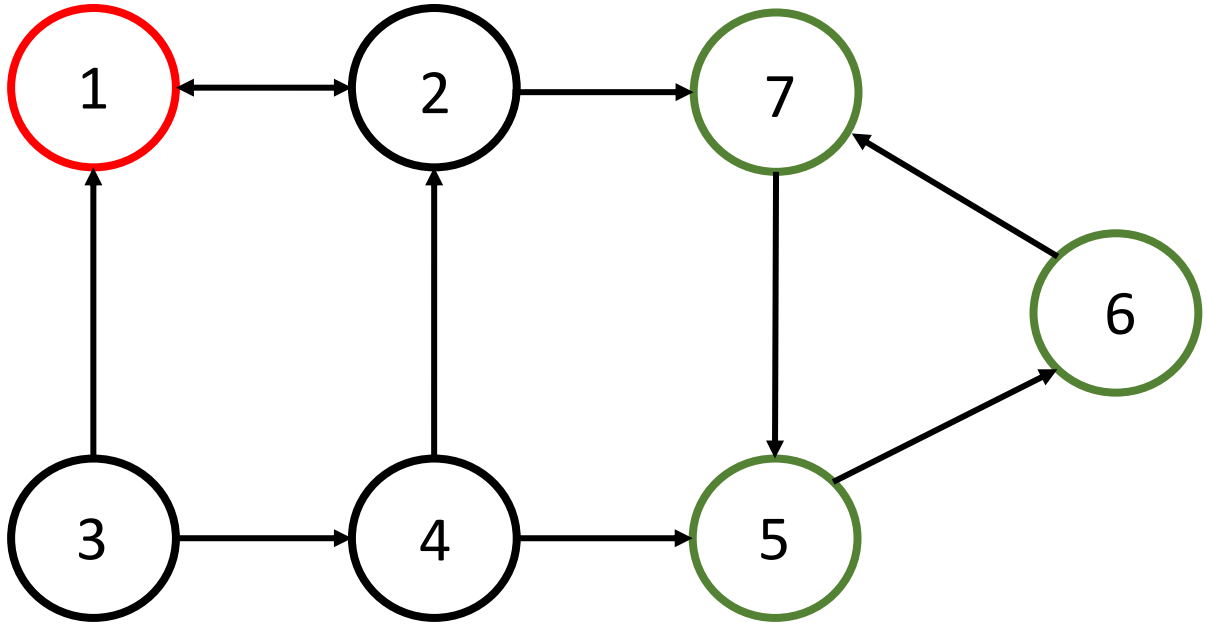
4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

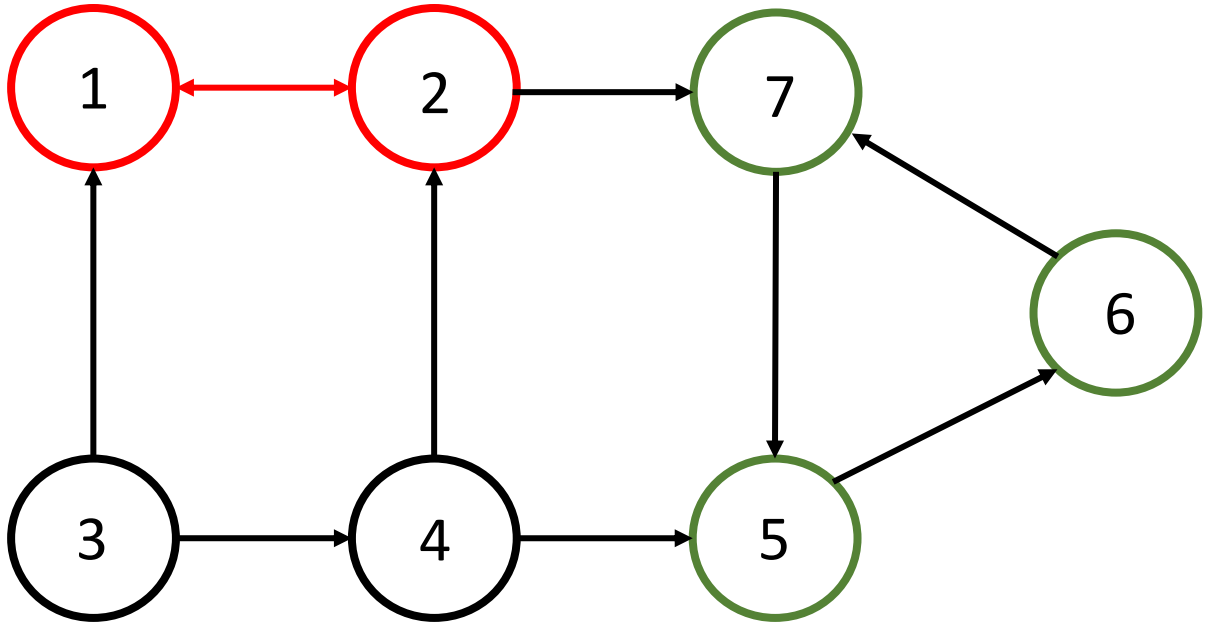
4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

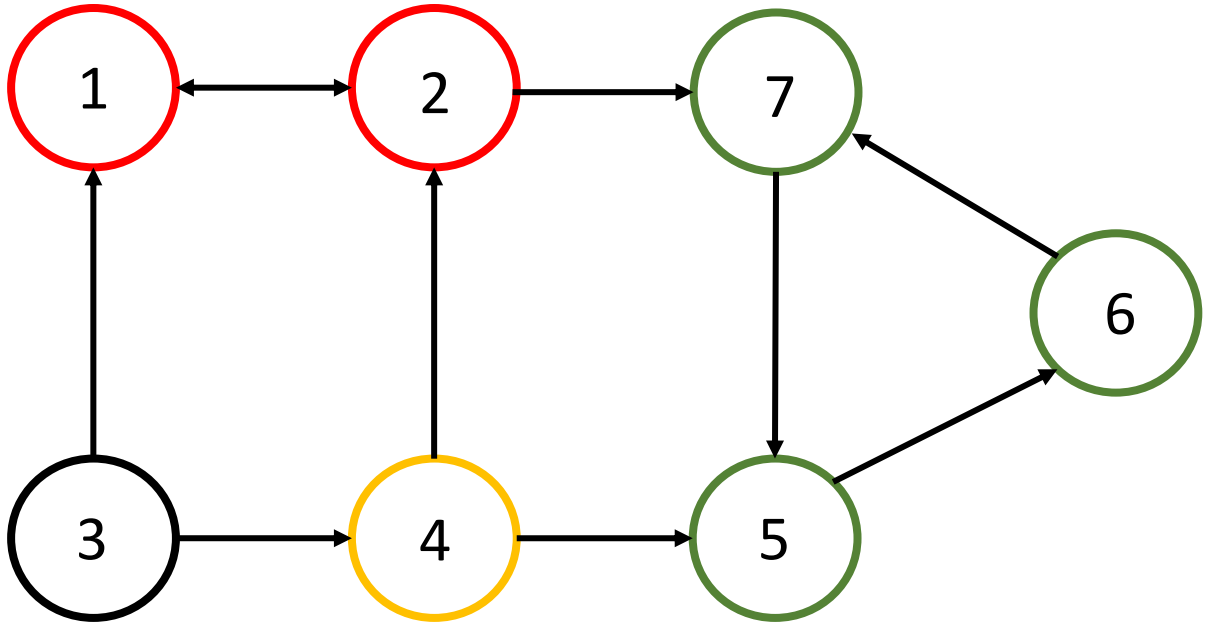
4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

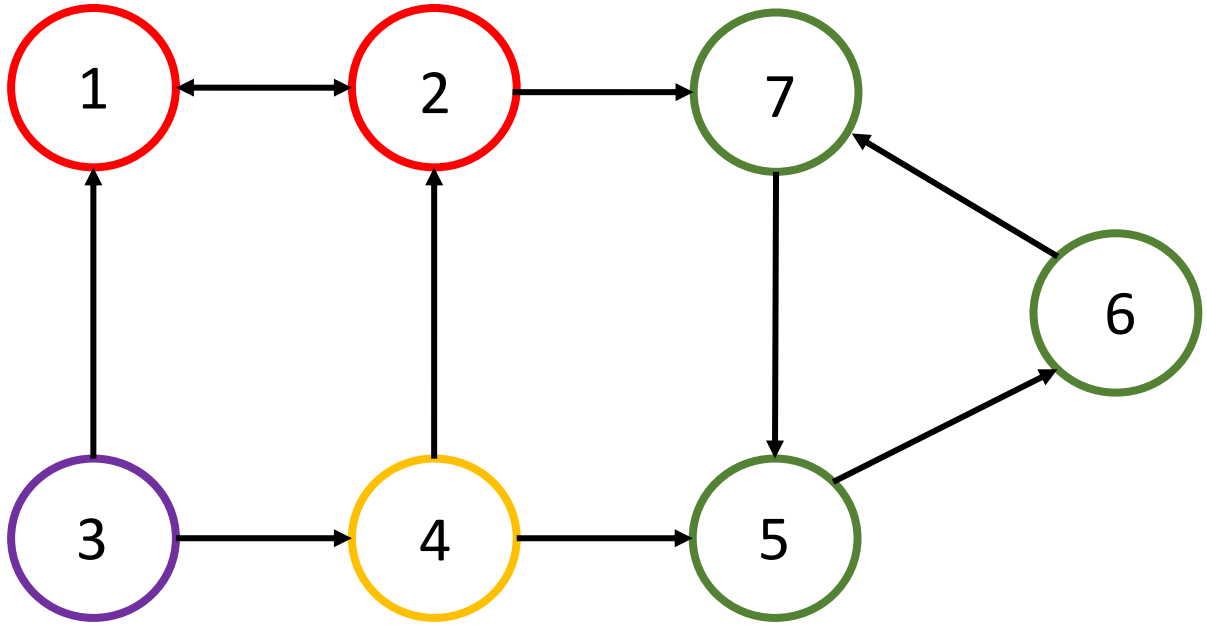
4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

4. 2nd DFS



NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

EXAMPLE:

NODE	FINISH TIME
7	14
6	13
5	12
1	8
2	7
4	6
3	5

DONE!

These components are now available for any other query

TIME AND SPACE COMPLEXITY

Time Complexity = O(n + m)

// due to the implementation of 2 depth-first searches

Space Complexity = O(n + m)

// accounts for the implementation of an adjacency list with n nodes and m edges

KOSARAJU'S ALGORITHM

(in pseudo)

1. Construct adjacency list
2. Perform DFS
    1. Flag entry time
    2. Push to children
    3. Flag exit time
    4. Add node timing object to list
3. Order list by descending exit time
4. Reverse all edges in the graph
5. Perform DFS from first list element
    1. Push nodes to component lists

WHY DOES THIS WORK THOUGH?

DETAILED PROOF HERE:

EXAMPLE PROBLEM

CSES 1683 : Planets and Kingdoms

Time limit: 1.00 s Memory limit: 512 MB

A game has n planets, connected by m teleporters. Two planets a and b belong to the same kingdom exactly when there is a route both from a to b and from b to a. Your task is to determine for each planet its kingdom.

ANSWER: FIND THE SCC (Strongly Connected Component) WHERE EACH NODE IS FOUND

SCC IMPLEMENTATION

INITIALIZATIONS

```
vector<pair<int, int>> adj[maxn]; // pair<int a <- target node, int b <- edge type>

vector<int> vect; // store node id in order of finish times

bool vis[maxn] = {0}; // visited array for DFS

int comp[maxn] = {0}; // comp[i] = ID of SCC

int clvl = 1; // current SCC ID
```

SCC = Strongly Connected Component

SCC IMPLEMENTATION

```
for (int i = 0; i < m; i++)
{
  int a, b; cin >> a >> b;
  a--; b--; // ZERO INDEXING OR NO INDEXING!

  adj[a].push_back(make_pair(b, 0)); // edges of type 0 are used in the first run
  adj[b].push_back(make_pair(a, 1)); // edges of type 1 are used in the second run
}

// run the first dfs
for (int i = 0; i < n; i++) if (!vis[i]) dfs(i, 0, -1);

reverse(vect.begin(), vect.end()); // reverse edges to find ordering by descending finish time

// run second dfs based on vect ordering
for (int i = 0; i < vect.size(); i++)
{
  if (!vis[vect[i].first])
  {
    dfs(vect[i].first, 1, clvl); // notice the second parameter!
    clvl++;
  }
}
```

SCC IMPLEMENTATION

```
void dfs(int a, int type, int cid)
{
  if (vis[a]) return;

  vis[a] = true;
  if (type == 1) comp[a] = cid; // set component id only in second run

  for (pair<int, int> child : adj[a])
  {
    if (child.second != type) continue;
    dfs(child.first, type, cid);
  }

  // append processed nodes to list
  vect.push_back(a);
}
```

Submission details

Task:	Planets and Kingdoms
Sender:	YOU IN THE FUTURE
Submission time:	SOONER THAN YOU THINK
Language:	C++17
Status:	READY
Result:	ACCEPTED

Test results

test	verdict	time	
#1	ACCEPTED	0.01 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.01 s	»
#4	ACCEPTED	0.01 s	»
#5	ACCEPTED	0.01 s	»
#6	ACCEPTED	0.15 s	»
#7	ACCEPTED	0.15 s	»
#8	ACCEPTED	0.14 s	»
#9	ACCEPTED	0.14 s	»
#10	ACCEPTED	0.14 s	»

APPLICATIONS

1. **Finding Strongly Connected Components** **(as shown in the example problem)**

2. **Condensed Graphs formed with SCC's are always acyclic.** **(We can use this fact to..)**

    - **Generate the topological ordering to apply Dynamic Programming techniques that tell us**

        - how many different paths there are
        - what the shortest/longest path is
        - what the minimum/maximum number of edges in a path is
        - which nodes certainly appear in any path

MORE SCC PROBLEMS

- SPOJ - True Friends
- SPOJ - Capital City
- Codeforces - Scheme
- SPOJ - Ada and Panels
- CSES - Flight Routes Check
- CSES -Coin Collector
- Codeforces - Checkposts