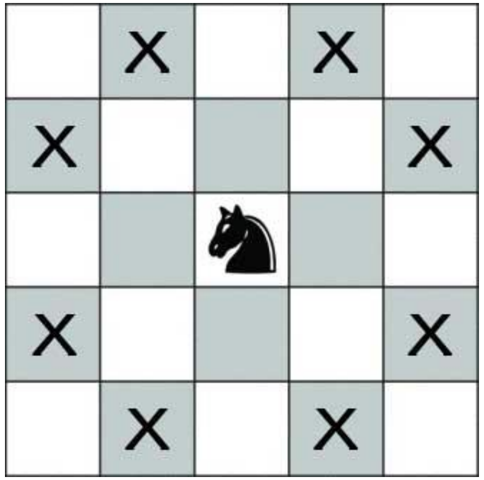
GRAPHS: DFS AND BFS BASED ALGORITHMS

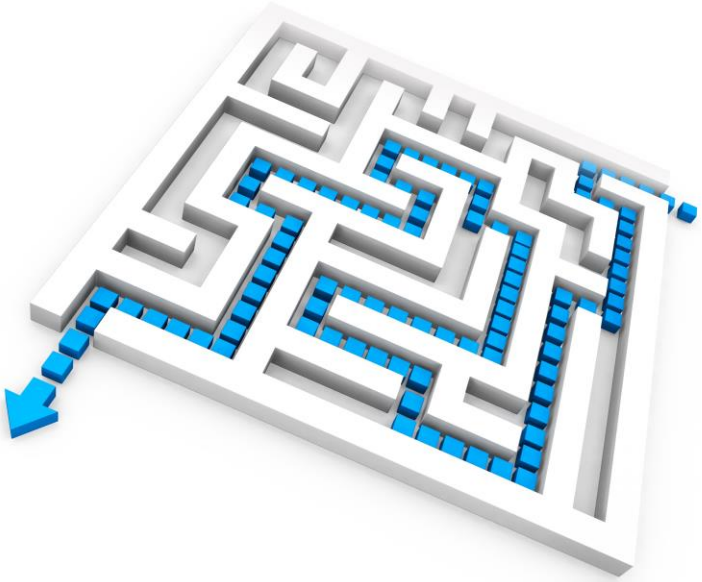
DATA STRUCTURES AND ALGORITHMS

GRAPHS DATA STRUCTURE AND ALGORITHMS

DFS and BFS based Algorithms

- Connected Components
- Cycle finding
  - For Directed
  - Undirected
- Connected Component
- Topological sorting
- All paths finding
- Shortest path (not weighted graph)
- Bipartite graph finding
- Edges Classification
- Articulation points finding
- Bridges Finding





CONNECTED COMPONENTS

DATA STRUCTURES AND ALGORITHMS

CONNECTED COMPONENTS

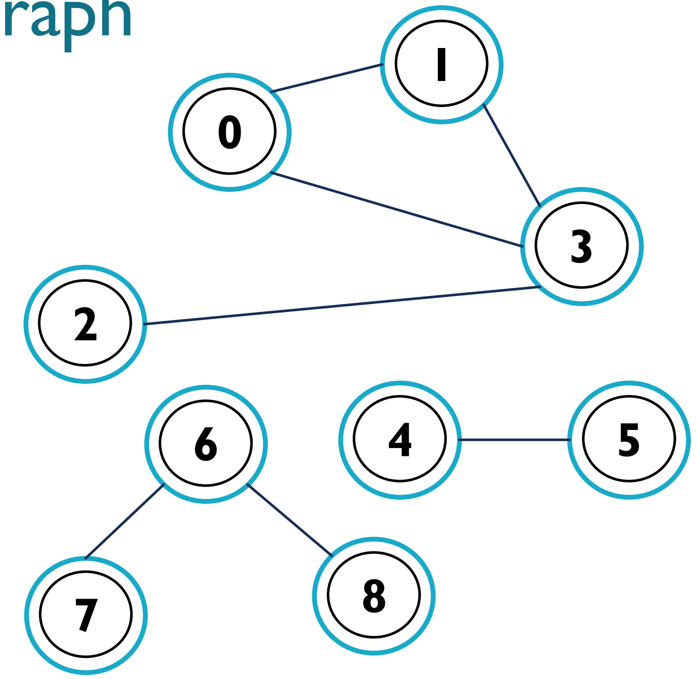
Definition

Connected component of an undirected graph is a subarray in which any two vertices are connected to each other by paths

restart DFS (or BFS) from one of the remaining unvisited vertices to find the next connected component. This process is repeated until all vertices have been visited and has an overall time complexity of O(V + E).

```
int cc_num = 0; //connected components
visited.assign(N, false);
for (int i = 0; i < N; i++){
    if(!visited[i]){
        cout<<"connected component "<<
++cc_num<<":";
        dfs(i);
        cout<<endl;
    }
}
```

Graph



Output

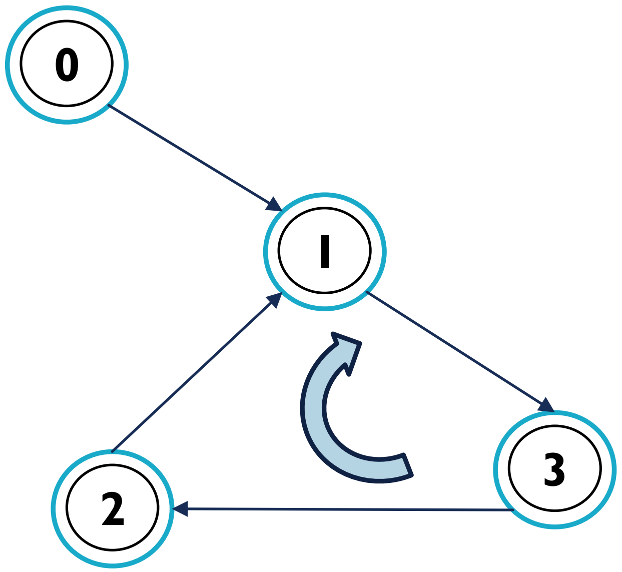
```
connected component 1: 0 1 3 2
connected component 2: 4 5
connected component 3: 6 7 8
```

CYCLE FINDING ALGORITHM

DATA STRUCTURES AND ALGORITHMS

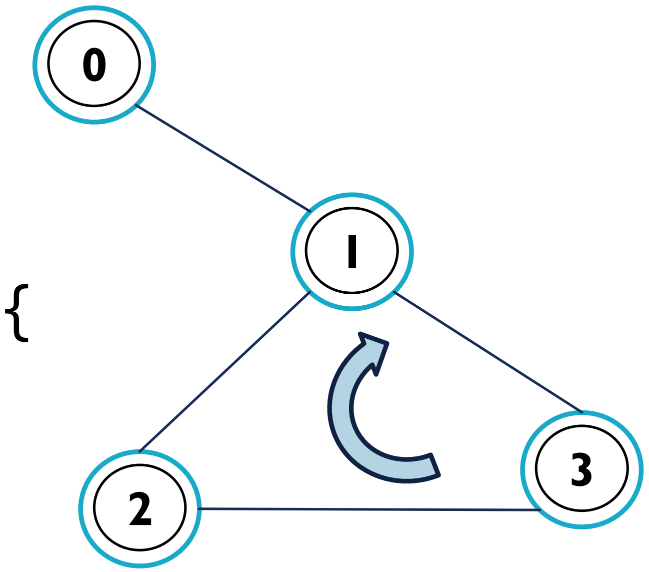
CYCLE FINDING IN DIRECTED GRAPH

```
vector<vector<int>> graph;
vector<bool> visited;

bool dfs(int u){
   visited[u] = true;
   for (int i = 0; i < graph[u].size(); i++){

      int v = graph[u][i];
      if(visited[v])
         return true;
      else
         dfs(v);
   }
   return false;
}
```



CYCLE FINDING IN UNDIRECTED GRAPH

```
vector<vector<int>> graph;
vector<bool> visited;

bool dfs(int u, int p = -1){
   visited[u] = true;
   for (int i = 0; i < graph[u].size(); i++){
      int v = graph[u][i];
      if(visited[v] && p != u)
         return true;
      else
         dfs(v, u);
   }
   return false;
}
```



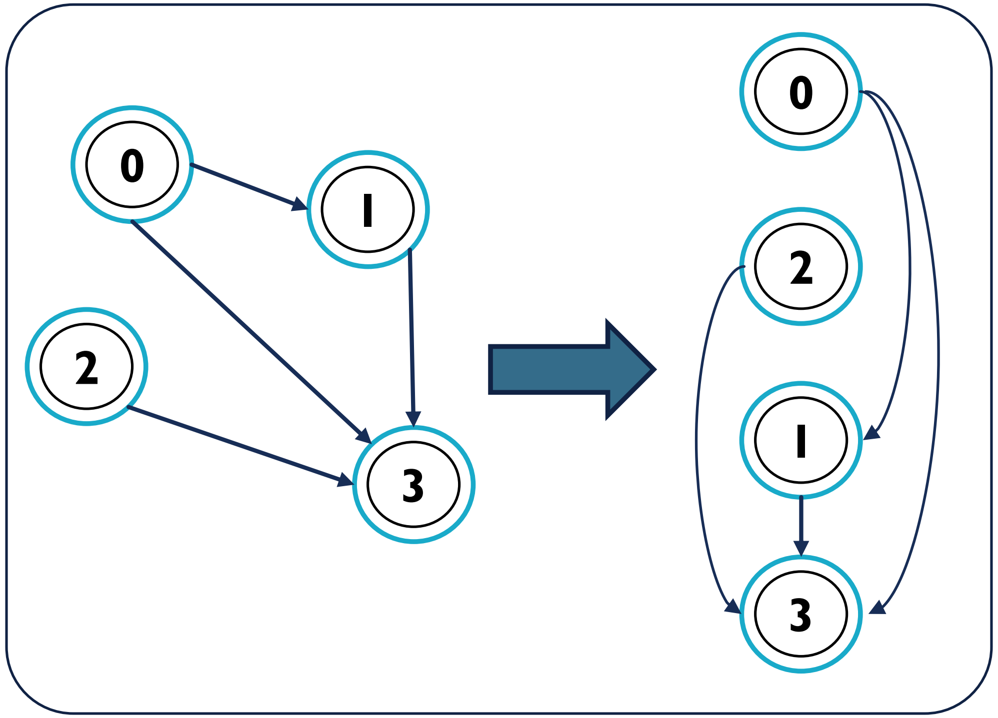
TOPOLOGICAL SORTING

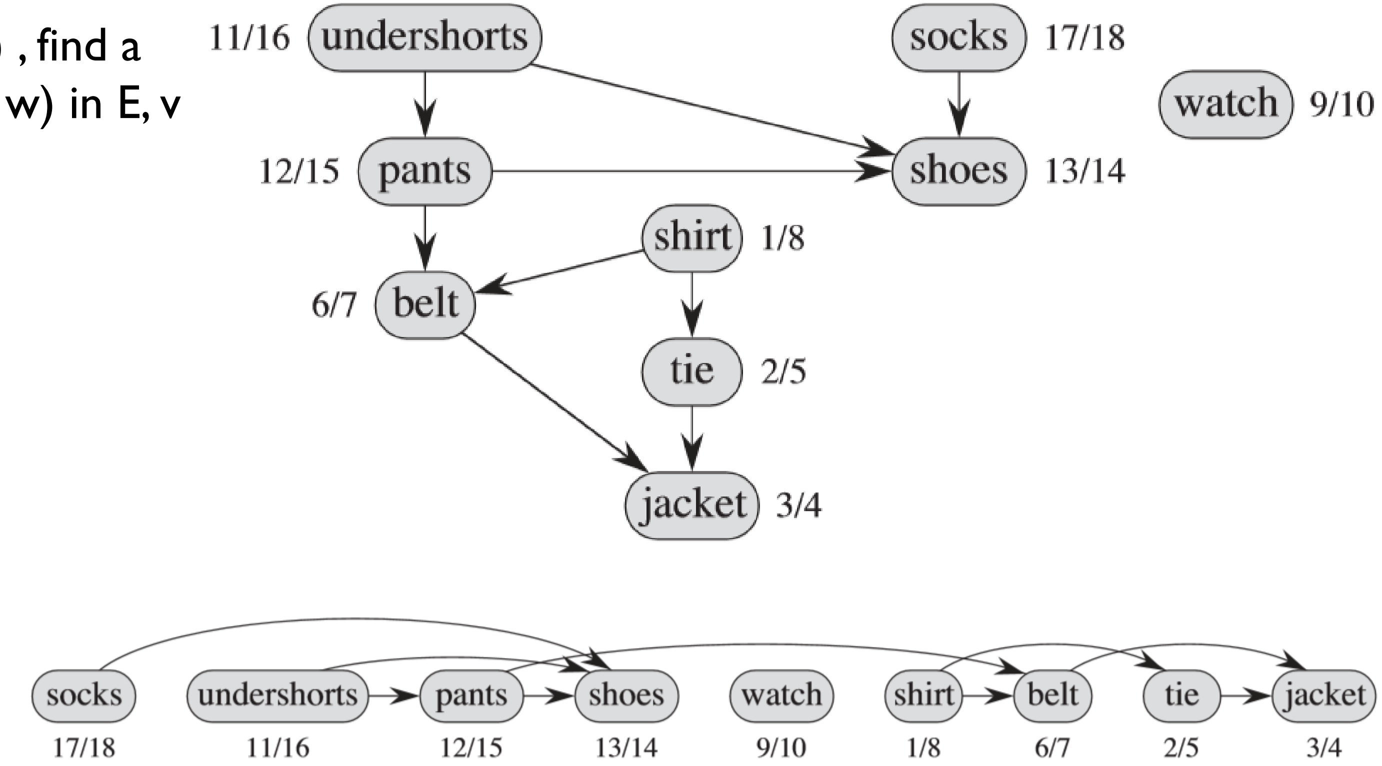
DATA STRUCTURES AND ALGORITHMS

TOPOLOGICAL SORTING

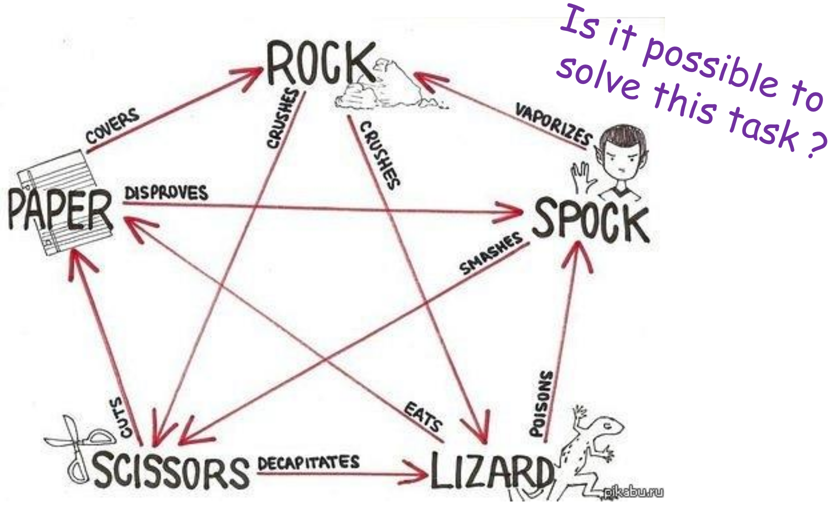
Definition

Topological sorting problem: given digraph G = (V, E) , find a linear ordering of vertices such that: for any edge (v, w) in E, v precedes w in the ordering





TOPOLOGICAL SORTING



Algorithm

A topological sorting is possible if and only if the graph has no directed cycles i.e., it is a Directed Acyclic Graph (DAG). It is always possible to find a topological order for a DAG and this can be done in linear time.

Topological sorting of a DAG can be found with DFS. During the depth-first traversal of the graph, just when a vertex finishes expanding (i.e., all its outlinks have been visited), add it to a stack. The order of vertices in the stack represents the topological order of the DAG.

```
Let L be an empty list

Let all vertices be initially unmarked

while there are unmarked vertices:
  select an unmarked vertex u
     dfs(u)


dfs(vertex u):
  mark u
  foreach edge u -> v
    if v is unmarked:
      dfs(v)
  add u to head of L

L represents the topological order of the DAG
```

TOPOLOGICAL SORTING IMPLEMENTATION

```
#include<iostream>
#include<vector>
using namespace std;

vector<vector<int>> g;
vector<bool> visited;
vector<int> ans;

void dfs(int v){
  visited[v] = true;
  for (int i=0; i<g[v].size(); ++i) {
     int to = g[v][i];
     if (!visited[to])
        dfs (to);
  }
  ans.push_back (v); // reverse path
}
```

```
void topological_sort(){
  for (int i=0; i<visited.size(); ++i)
    visited[i] = false;
  ans.clear();
  for (int i=0; i<visited.size(); ++i)
     if (!visited[i])
          dfs (i);
  reverse (ans.begin(), ans.end());
}
```

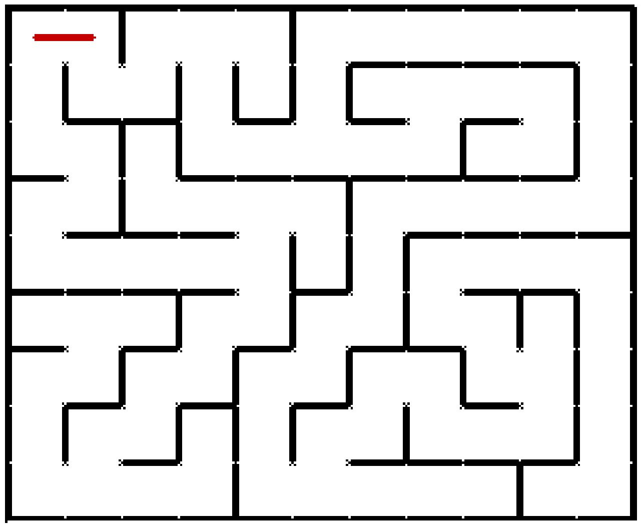
ALL PATHS FINDING

DATA STRUCTURES AND ALGORITHMS

DFS ALL PATHS FINDING

Algorithm

- Additional parameter for storing the path
- Use the DFS with two additional parts
  - Source – Destination producing (printing or storing)
  - Backtracking



```
dfs( curr, dest)

    If curr == dest
     print path from curr to dest

DFS visitied [ cur ] = true;
    from cur vertex to all adjusts
     if not visited[ adjust ]
            dfs(curr,  dest)

    backtrack the path.
```

DFS ALL PATHS FINDING

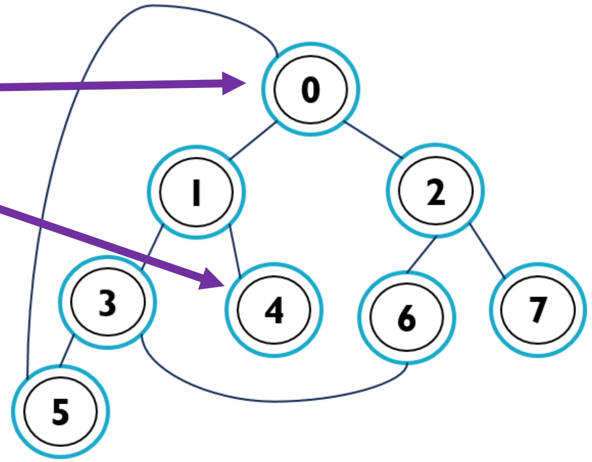
```cpp
#include<iostream>
#include<vector>
using namespace std;
int n,m, a,b;
vector <vector<int>> adj;
vector <int> visited, path;

void dfs(int curr, int dest){
    if(curr == dest) {
      for(int i = 0; i < path.size(); i++) {
        cout << path[i] << ' ';
      }
    cout << dest << '\n';
    return;
  }
  path.push_back(curr);
  visited[curr] = true;
  for (int i = 0; i < adj[curr].size(); ++i) {
    int now = adj[curr][i];
    if(!visited[now])
      dfs(now, dest);
  }
  path.pop_back();
  visited[curr] = false;
}
```

```cpp
int main(){

  cin >> n >> m;
  adj = vector <vector <int>>(n);
  visited = vector <int> (n, 0);
  for (int i = 0; i < m; ++i) {
    cin >> a >> b;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }
  cout<<endl;
  dfs(0, 4);
  system("pause");
  return 0;
}
```

c:\users\sul\documents\vis...

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

0 1 4
0 2 6 3 1 4
0 5 3 1 4
Press any key to continue . . .
```

All paths from 0 to 4



EDGES CLASSIFICATION

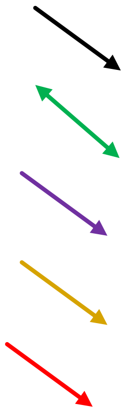
DATA STRUCTURES AND ALGORITHMS

EDGES CLASSIFICATION

Definition

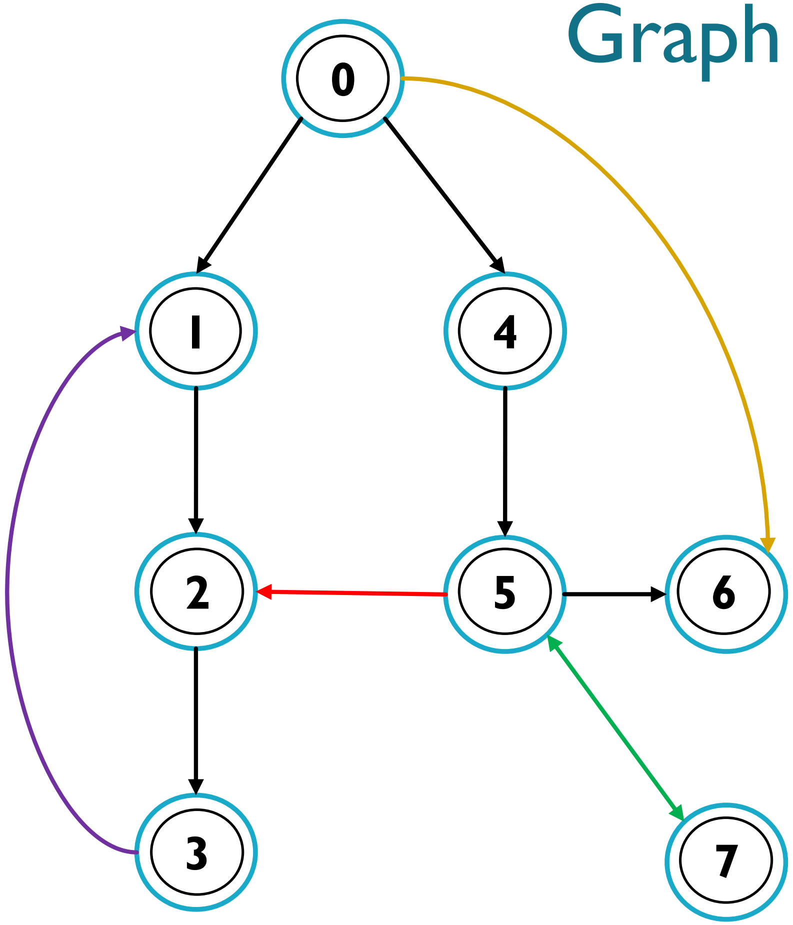
- If v is visited for the first time as we traverse the edge (u, v), then the edge is a tree edge
- Else, v has been visited
  - If v is an ancestor of u , then edge (u, v) is a back edge
  - Else, if v is a descendant of u , then edge (u, v) is a forward edge
  - Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a cross edge

Notation



- Tree edge
- Bidirectional edge
- Back Edge
- Forward Edge
- Cross Edge

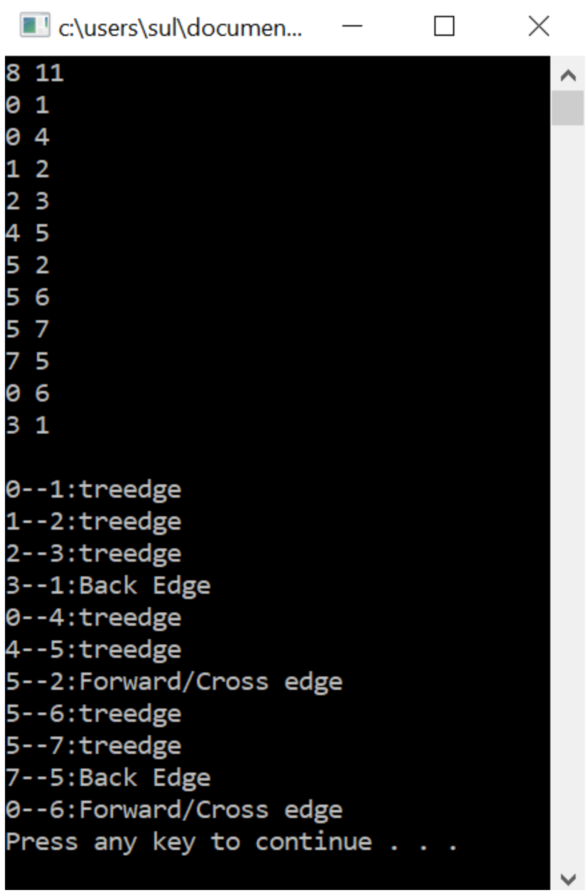
Graph



EDGES CLASSIFICATION

```cpp
#include<iostream>
#include<vector>
#define white 0
#define gray 1
#define black 2
using namespace std;
int n,m, a,b;
vector <vector<int>> adj;
vector <int> color;
vector<int> parent;

void dfs(int curr);

int main(){
  cin >> n >> m;
  adj = vector < vector <int> > (n);
  color = vector <int> (n, 0);
  parent = vector<int> (n, -1);
  for (int i = 0; i < m; ++i) {
    cin >> a >> b;
    adj[a].push_back(b);
  }
  cout<<endl;
  dfs(0);
  system("pause");
  return 0;
}
```

```cpp
void dfs(int curr){

  color[curr] = gray;
  for (int i = 0; i < adj[curr].size(); ++i) {
    int now = adj[curr][i];
    if(color[now] == white){
      parent[now] = curr;
      cout<<curr<<"--"<<now<<":treedge\n";
      dfs(now);
    }
    else if(color[now] == gray){
       if(now == parent[now])
          cout<<curr<<"--"<<now<<":Bidirectional\n";
        else
          cout<<curr<<"--"<<now<<":Back Edge\n";
    }
    else if(color[now] == black)
       cout<<curr<<"--"<<now<<":Forward/Cross edge\n";
  }
  color[curr] = black;
}
```

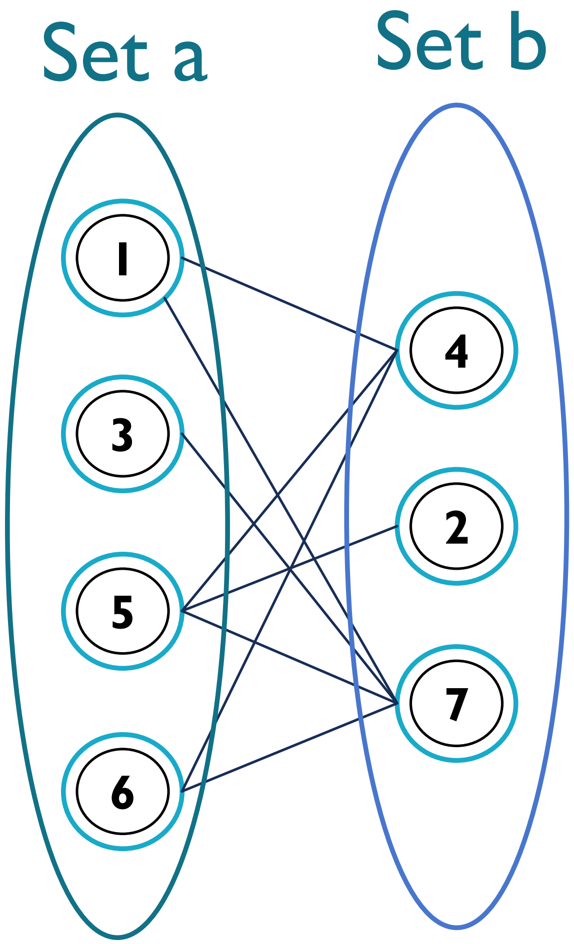


BIPARTED GRAPH FINDING

DATA STRUCTURES AND ALGORITHMS

BIPARTED GRAPH IMPLEMENTATION

```
vector<vector<int>> graph;
vector<int> color;
```




```
bool bfs(int s){
  fill(color.begin(), color.end(), -1);
  color[s] = 1;
  queue<int> q;
  q.push(s);
  while (!q.empty()){
     int u = q.front(); q.pop();
     for (int i = 0; i < graph[u].size(); i++){
       int v = graph[u][i];

       if(v == u) // self loop
           return false;
       if(color[v] == -1){
            color[v] = 1 - color[u];
            q.push(v);
       }else if(color[v] == color[u])
                  return false;
     }
  }
  return true;
}
```

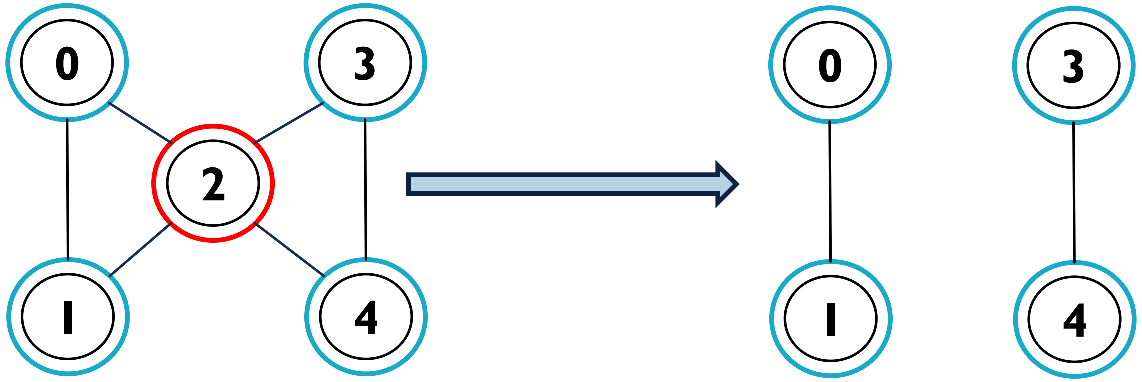
ARTICULATION POINT FINDING

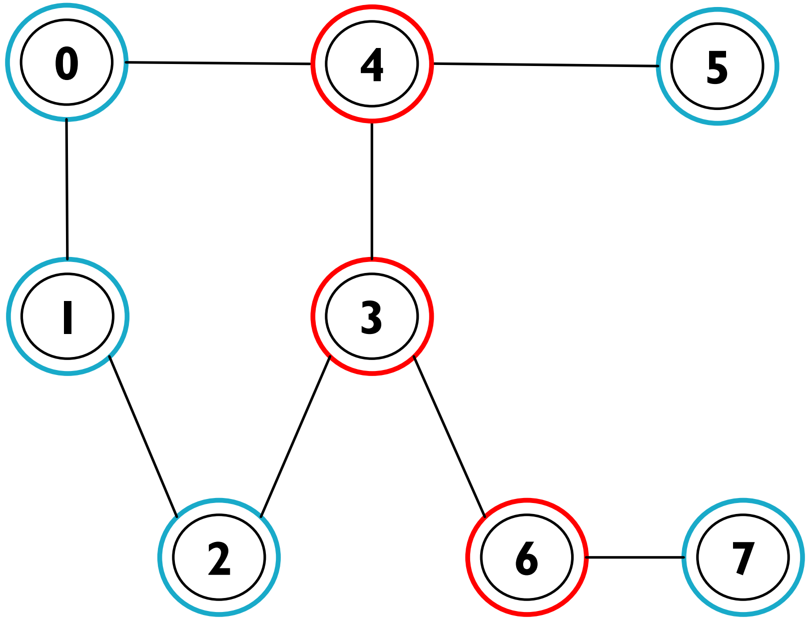
DATA STRUCTURES AND ALGORITHMS

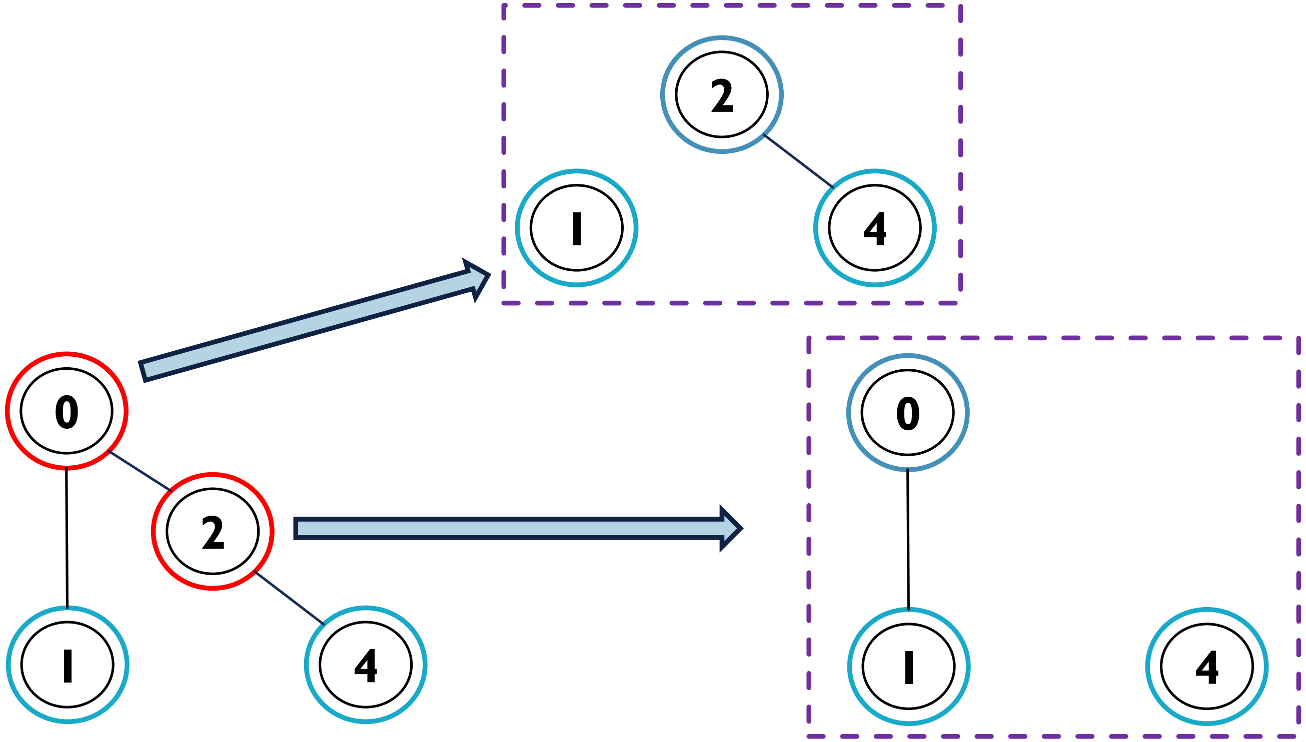
ARTICULATION POINT

Definition

A vertex in an undirected connected graph is an articulation point iff removing it disconnects the graph





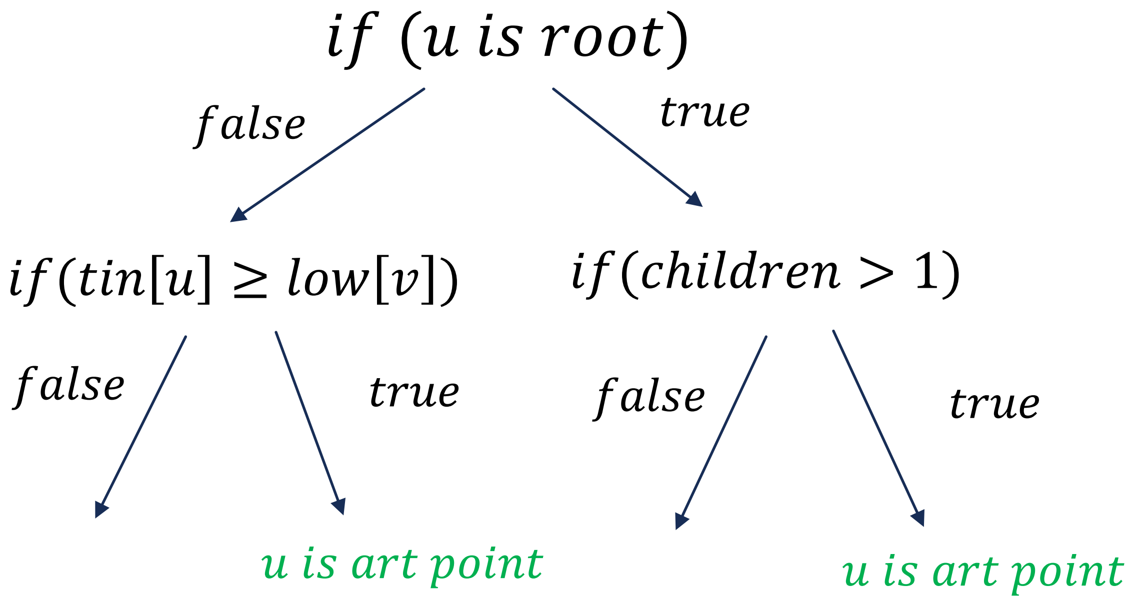


ARTICULATION POINT

Algorithm

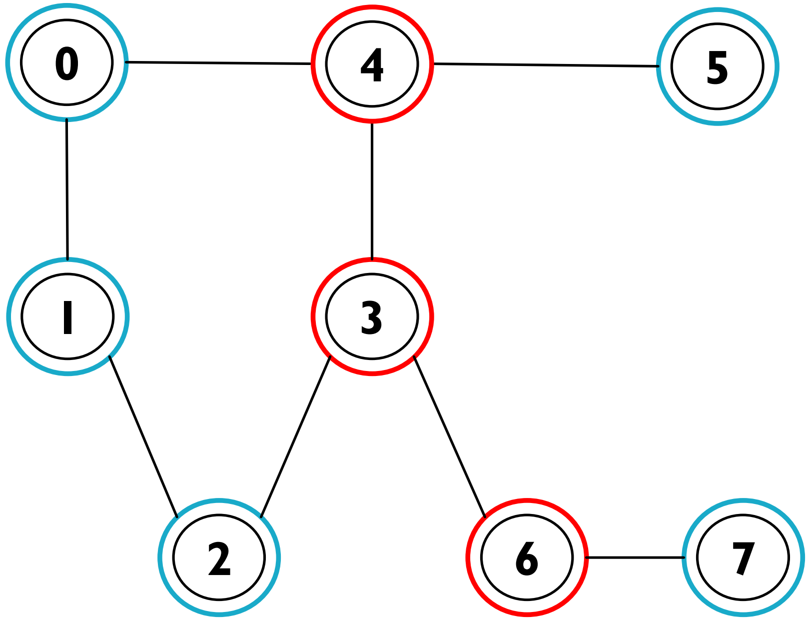
$$\text{low}[u] = \begin{cases} \text{u} \to \text{v tree edge: } \min(\,low[u], low[v]) \\ u \to v\ back\ edge\text{: } \min(low[u], tin[v]) \end{cases}$$

- u is root of DFS tree and it has more than one children
- u is not the root
  - If it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors in of u



ARTICULATION POINT IMPLEMENTATION

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
vector<vector<int>> graph;
vector<bool> visited;
vector<int> tin, low;
int timer;
```




```
void dfs(int u, int p = -1){
    low[u] = tin[u] = timer++;
    visited[u] = true;
    int child = 0;
    for (int i = 0; i < graph[u].size(); i++)
    {
            int v = graph[u][i];
            if(v == p)
                continue;
            if(visited[v])
                low[u] = min(low[u], tin[v]); // back edge
            else{
                dfs(v, u);
                low[u] = min(low[u], low[v]); // tree edge
                if( low[v] >= tin[u] && p != -1)
                    cout<<"art point: "<<u<<endl;
                child++;
            }
    }
    if(child > 1 && p == -1)
    cout<<"art point: "<<u<<endl;
}
```

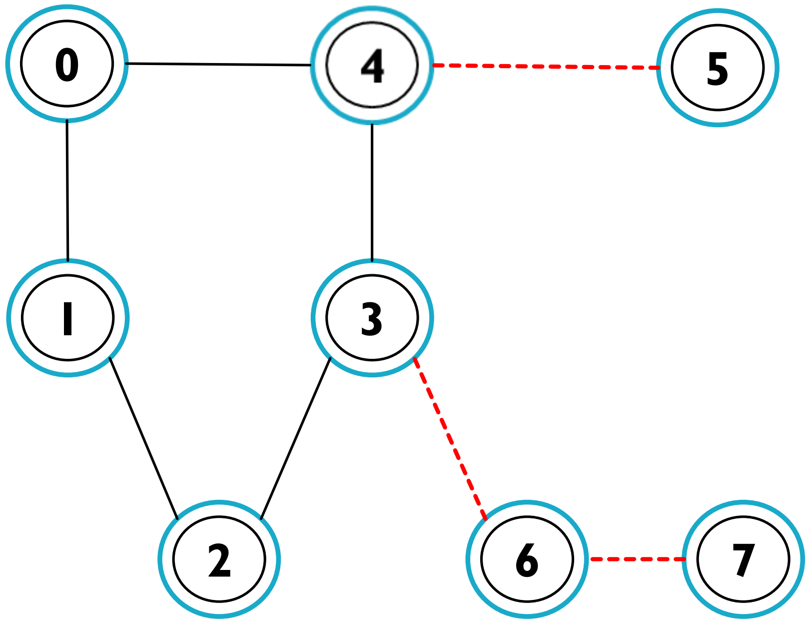
BRIDGES FINDING

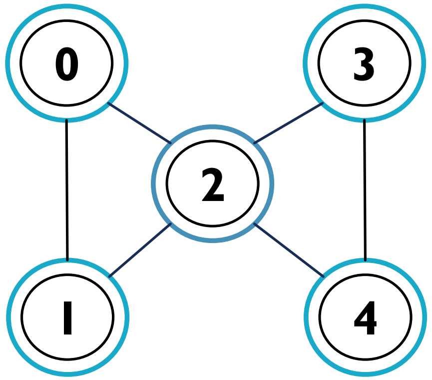
DATA STRUCTURES AND ALGORITHMS

BRIDGES

Definition

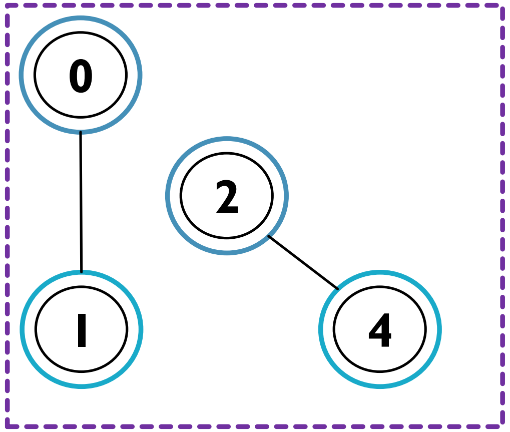
An edge in an undirected connected graph is the bridge iff removing it disconnects the graph

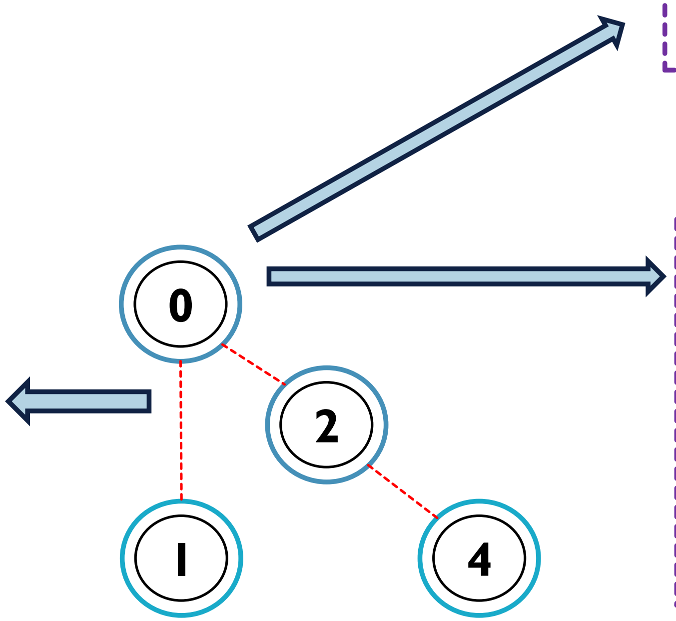




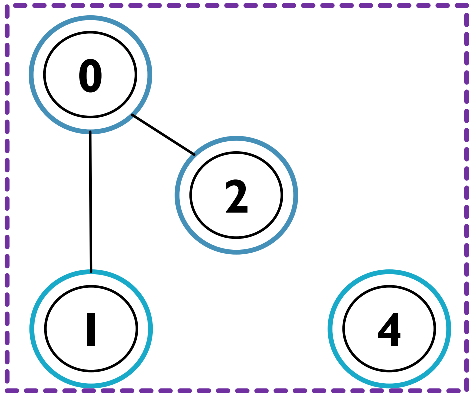
No Bridges,
but has an articulation point

Remove {0, 2}

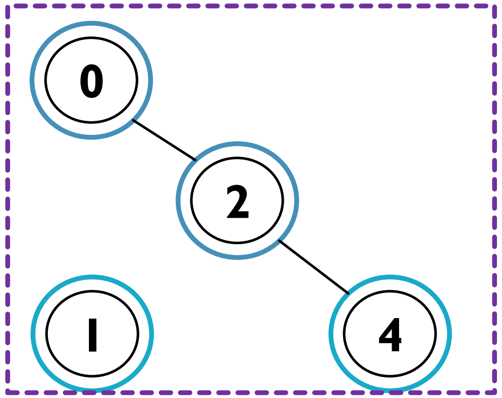




Remove {2, 4}

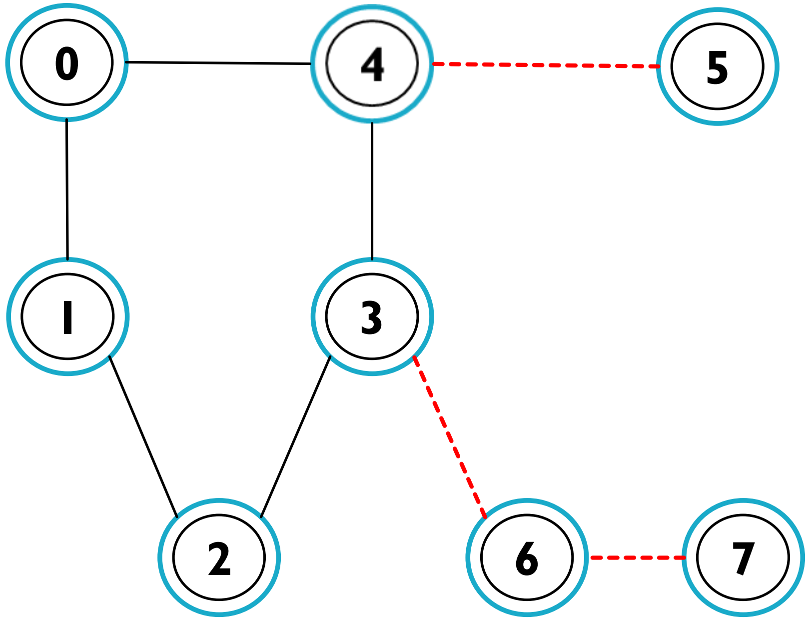


Remove {1, 0}



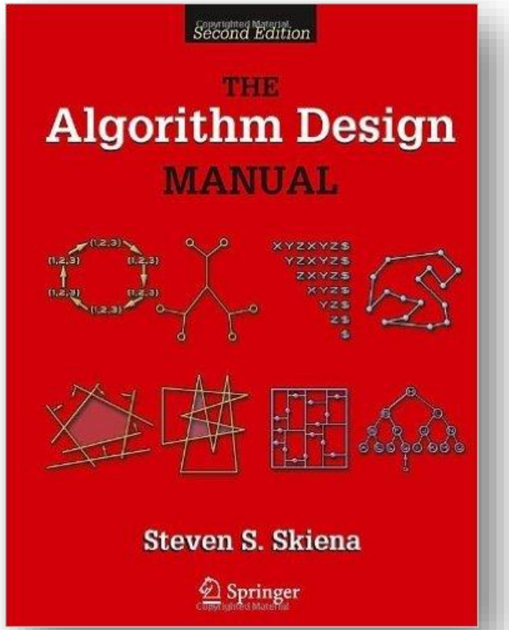
BRIDGE FINDING IMPLEMENTATION

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
vector<vector<int>> graph;
vector<bool> visited;
vector<int> tin, low;
int timer;
```

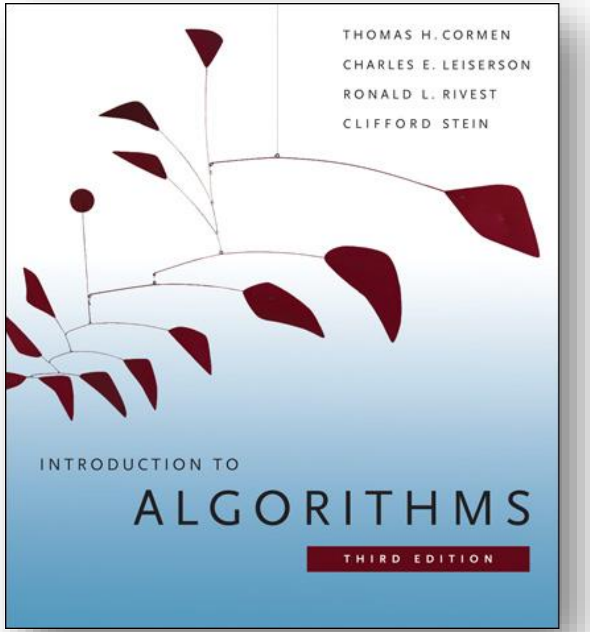



```
void dfs(int u, int p = -1){
    low[u] = tin[u] = timer++;
    visited[u] = true;
    for (int i = 0; i < graph[u].size(); i++)
    {
        int v = graph[u][i];
        if(v == p)
            continue;
        if(visited[v])
            low[u] = min(low[u], tin[v]); // back edge
        else{
            dfs(v, u);
            low[u] = min(low[u], low[v]); // tree edge
            if( low[v] > tin[u] && p != -1)
                cout<<u<<"---"<<v<<endl;
        }
    }
}
```

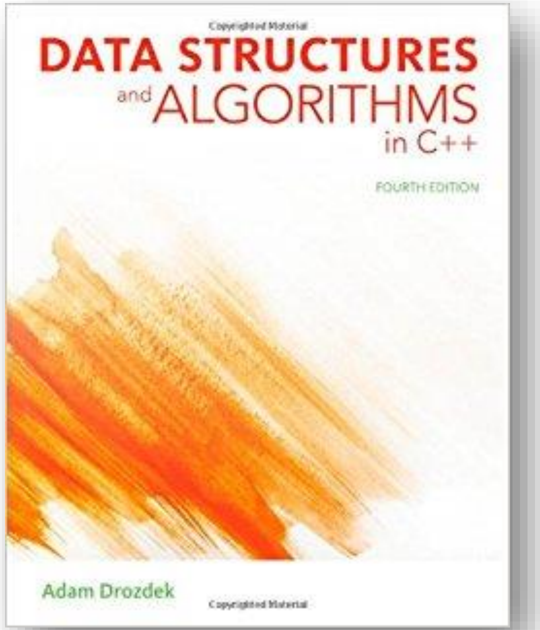
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 5: Graph Traversal
Page 145



Thomas H. Cormen
Introduction to Algorithms
Chapter VI Graph Algorithms
Page 587.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 8: Graphs
Page 391