Graph (IV)

Gabriel Liu {gabrielliu2001}

2023-04-15

Introduction

(From Wikipedia) In mathematics and computer science, **connectivity** is one of the basic concepts of graph theory.

It asks for the minimum number of elements (nodes or edges) that need to be removed to separate the remaining nodes into two or more isolated subgraphs.

It is closely related to the theory of network flow problems. The connectivity of a graph is an important measure of its resilience as a network.

Our Powerful Weapons

- DFS
- DFS
- DFS
- ...

- Tree
- DAG
- Stack
- Array
- ~~For loop, fundamental programming knowledge~~

Agenda

- Undirected Graph:
  - Bridge (Cut Edge)
  - Articulation Point (Cut Vertex)
  - Bridge-connected Component (2-edge-connected Component)
  - Biconnected Component
  - Cactus
- Directed Graph:
  - Strongly Connected Component

Practice Tasks

For those who have already learnt everything in this lecture:

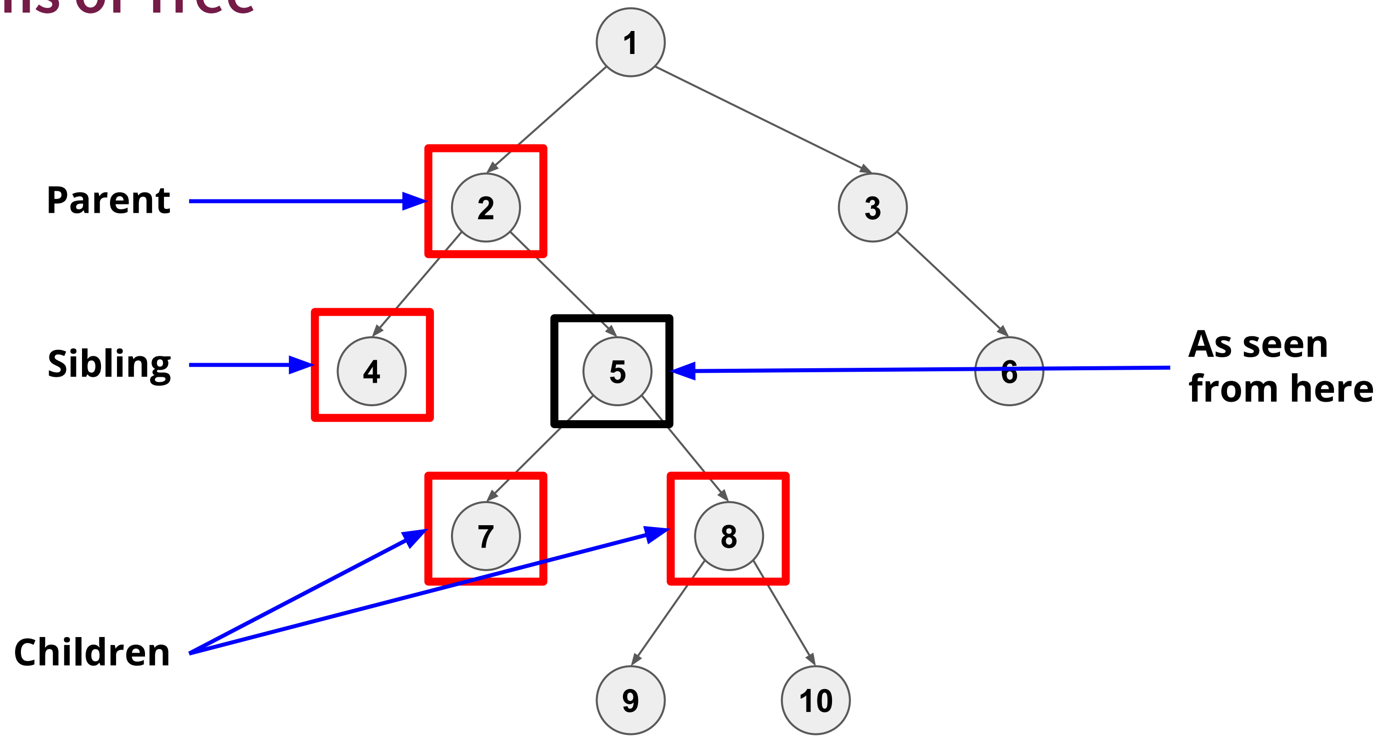
- https://codeforces.com/gym/102835/problem/I
- https://oj.uz/problem/view/IOI19_split
- https://oj.uz/problem/view/CEOI17_oneway
- https://codeforces.com/problemset/problem/487/E
- https://codeforces.com/contest/856/problem/D
- https://codeforces.com/contest/1239/problem/D
- https://codeforces.com/problemset/problem/1215/F

Revision: Terms of Tree

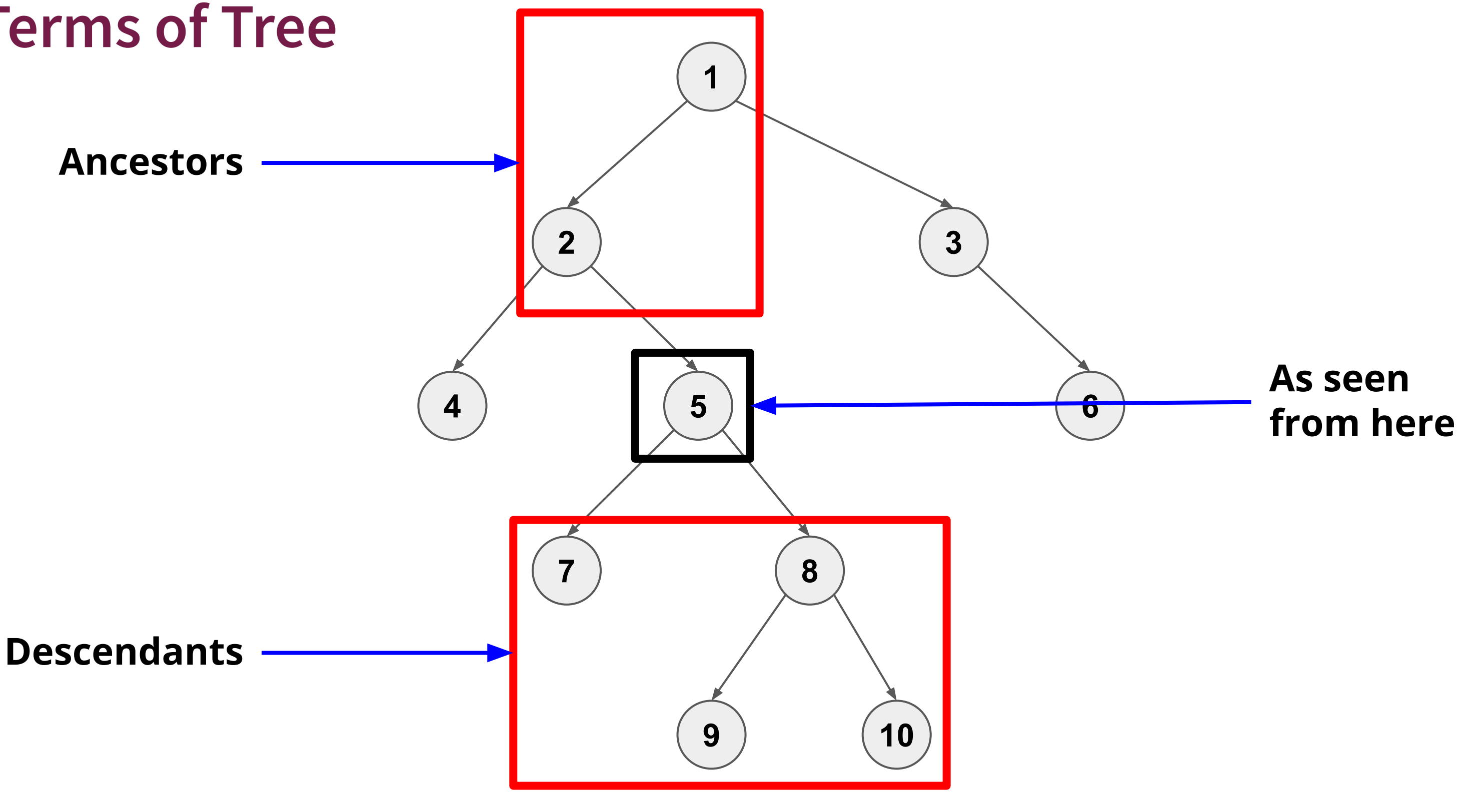
Adapted from Graph (III) (2023)

https://assets.hkoi.org/training2023/g-iii.pdf

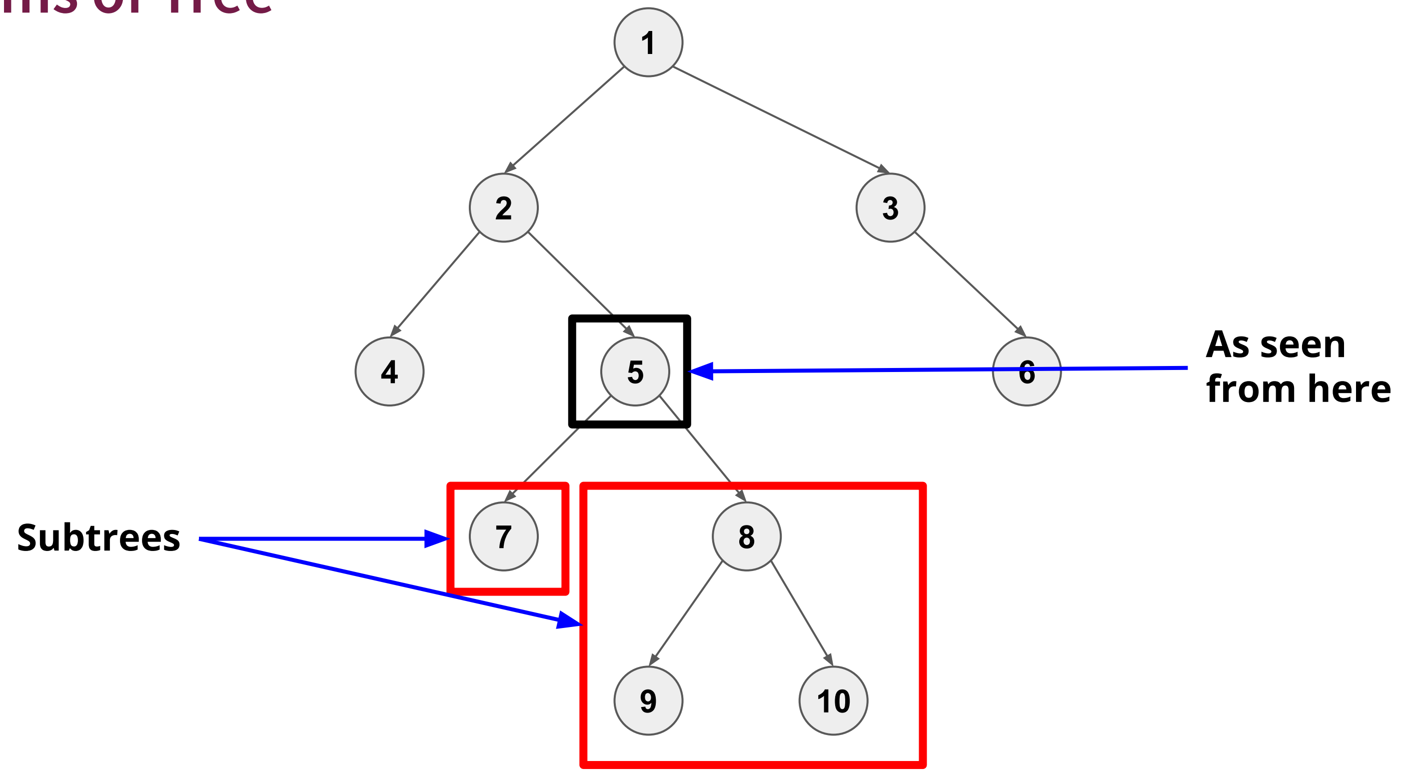
Revision: Terms of Tree



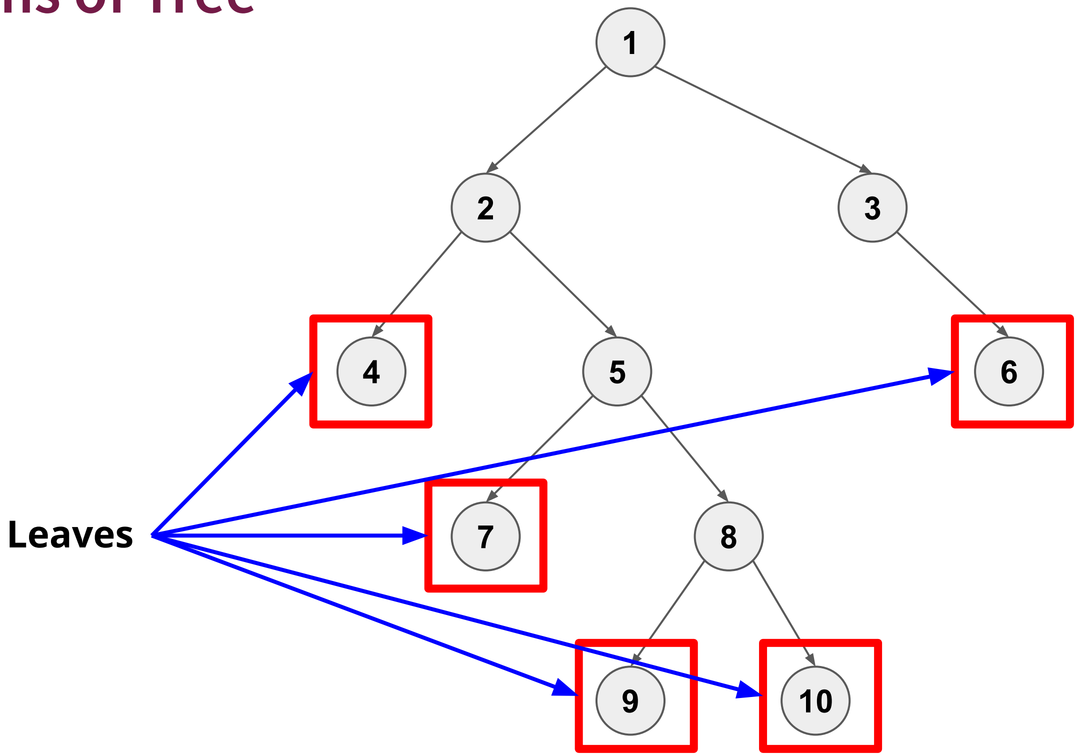
Revision: Terms of Tree



Revision: Terms of Tree



Revision: Terms of Tree



Revision: DFS

```
vector<vector<int>> G;  // Adjacency List
vector<bool> vis;

void dfs(int u) {
  vis[u] = true;
  for (int v : G[u])
    if (!vis[v])
      dfs(v);
}
```

```
vector<vector<int>> G;  // Adjacency List
vector<int> st, ed;
int cnt = 0;

void dfs(int u) {
  st[u] = ++cnt;
  for (int v : G[u])
    if (!st[v])
      dfs(v);
  ed[u] = cnt;
}
```

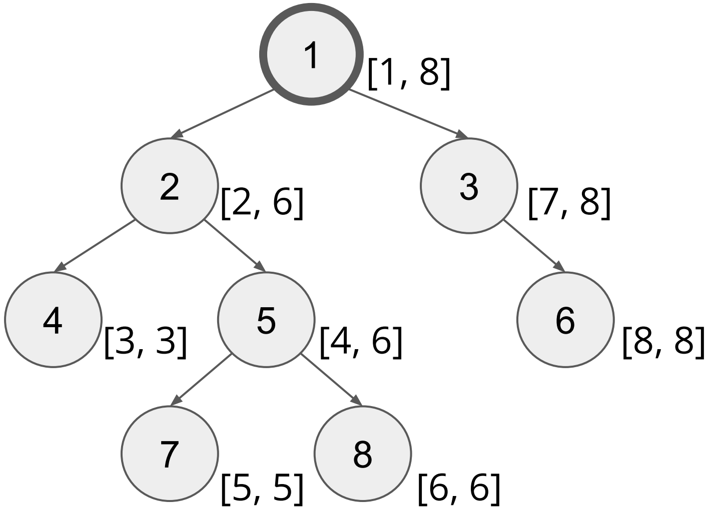
Some tricks

O(1) check if u is ancestor of v:

st[u] < st[v]

ed[u] >= ed[v]

More: refer to 2019 Graph (IV) slide



DFS Tree

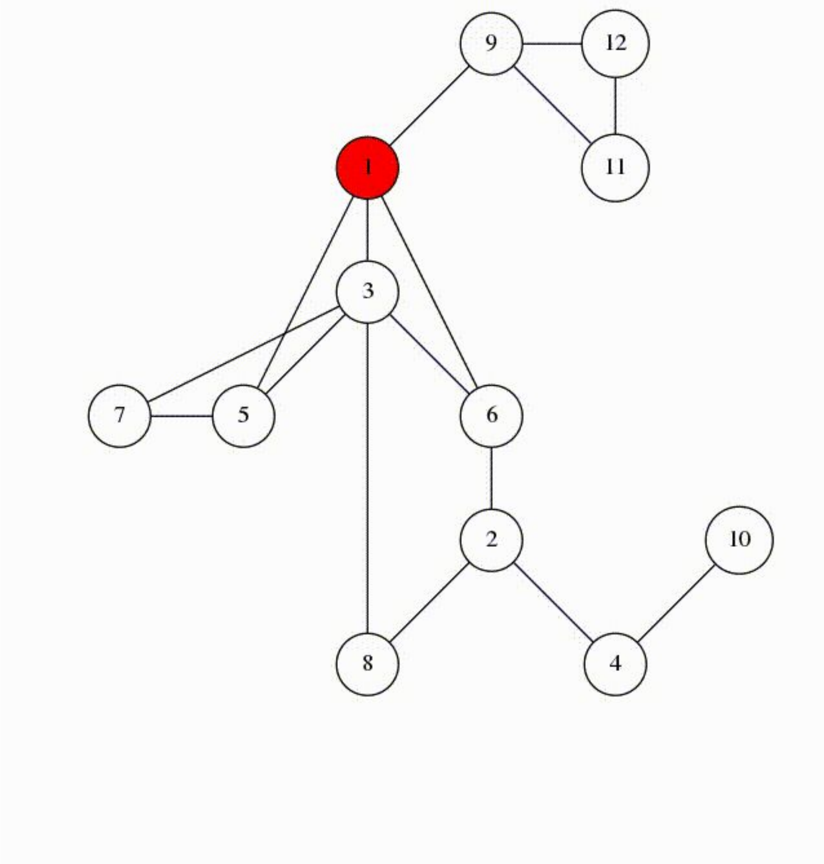
DFS Tree

- When we DFS on a graph, we get a DFS forest
- For simplicity, we will focus on DFS tree, as forest is just many trees
- **Reminder**: the following graphs are all **undirected graphs**

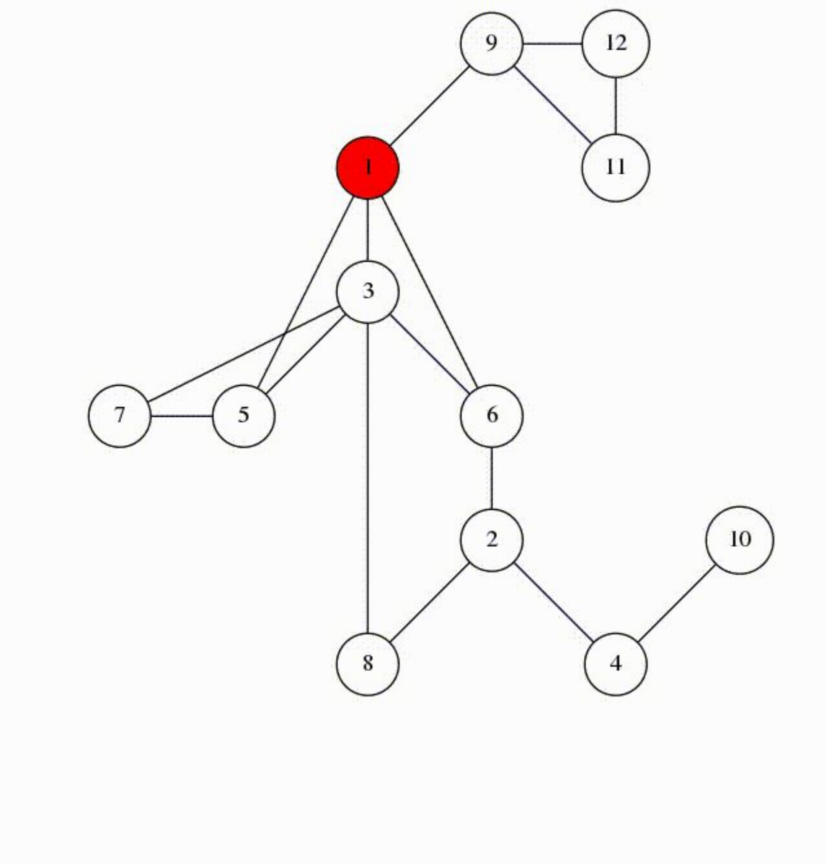
DFS Tree

Consider the graph and the function on the next page:

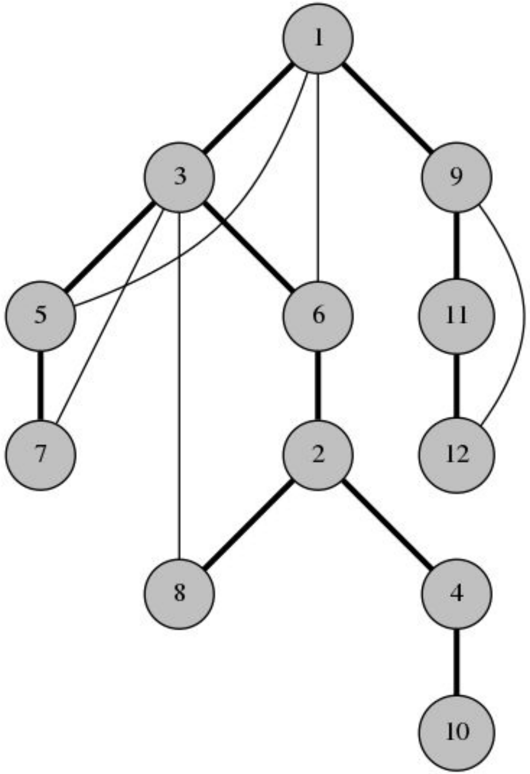
what edges will be marked in line 5 when calling `visit(1)`?




```
function visit(u):
  mark u as visited
  for each vertex v among the neighbours of u:
    if v is not visited:
      mark the edge u-v
      call visit(v)
```




```
function visit(u):
  mark u as visited
  for each vertex v among the neighbours of u:
    if v is not visited:
      mark the edge u-v
      call visit(v)
```




```
function visit(u):
  mark u as visited
  for each vertex v among the neighbours of u:
    if v is not visited:
      mark the edge u-v
      call visit(v)
```

DFS Tree

We mark some edges in line 5, let's call them **tree edges**.

Let's call other edges **back edges**.

```
1 function visit(u):
2   mark u as visited
3   for each vertex v among the neighbours of u:
4     if v is not visited:
5       mark the edge u-v
6       call visit(v)
```

Observation 1

Every **back edge** connects an **ancestor** and a **descendant**.

In other words:

- Tree edge (u - v): u is parent of v (st[u] < st[v], ed[u] >= ed[v])
- Back edge (u - v): st[u] > st[v], ed[u] <= ed[v]

P.S. without other specification, when we use the notation “u and v”, assume that we are now at node u.

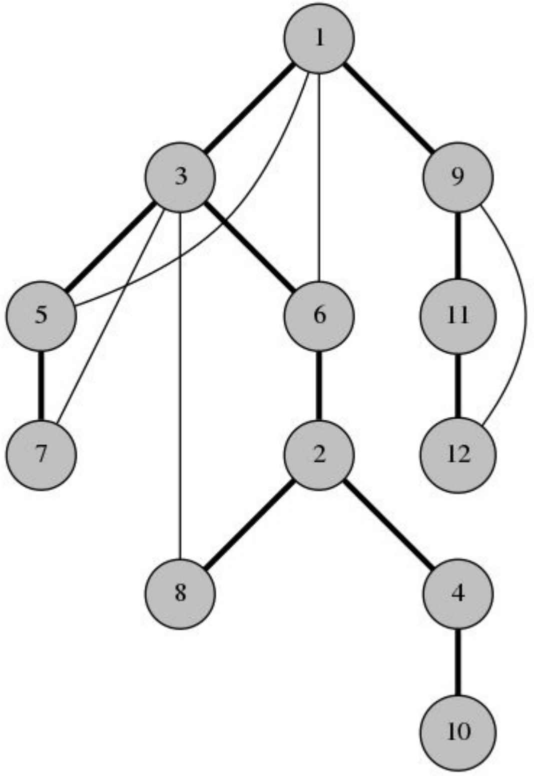
Observation 1

Wait... how about if u is v and v is u?

(u, v) = (8, 3) vs (u, v) = (3, 8)

Actually, every back edge is also a forward edge.

- forward edge (u - v): st[u] < st[v], ed[u] >= ed[v]



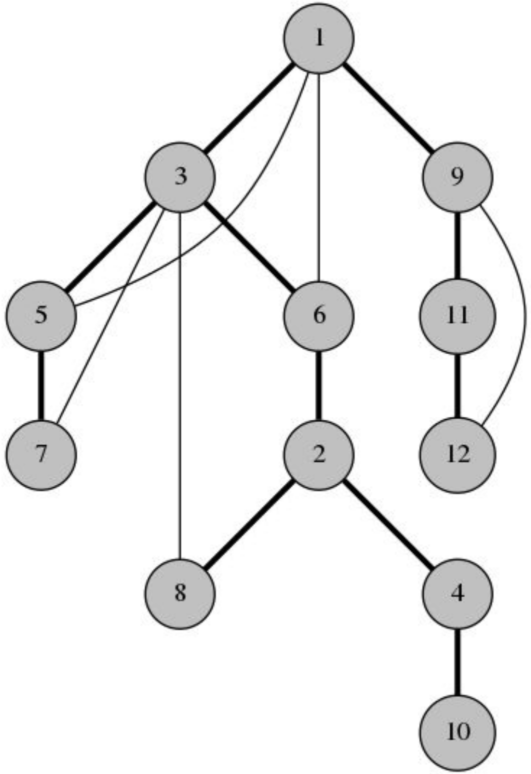
Observation 1

Wait... how about if u is v and v is u?

(u, v) = (8, 3) vs (u, v) = (3, 8)

Actually, every back edge is also a forward edge.

- Forward edge (u - v): st[u] < st[v], ed[u] >= ed[v]
- u is NOT the parent of v in DFS tree!



When solving tasks, sometimes you don't need to care about visiting forward edge (e.g. finding bridge), but sometimes you need (e.g. CF118E, practice sample).

Bridge (Cut Edge)

Definition: an edge of a graph whose deletion increases the graph's number of connected components.

We can use the idea of DFS tree to find all bridges in $O(V + E)$.

Bridge (Cut Edge)

Recall our DFS tree, we have **tree edges** and **back edges**.

How can we determine if an edge(u - v) is a bridge?

Observation 2

A back edge is never a bridge.

Why? Because tree edges already connect the whole graph!

Observation 3

A tree edge(u - v) is a bridge if and only if there is **NO back edge** connecting a descendant of v (including v) with an ancestor of u (including u).

In other words, a tree edge(u - v) is a bridge if and only if there is no back edge "passing over" edge(u - v).

Bridge (Cut Edge)

OK now we have an algorithm to find the bridges:

1. Find the DFS tree of the graph
2. For each tree edge(u - v), check if there is a back edge “passing over” (u - v)

Time for implementation. The algorithm is usually called “Tarjan’s Algorithm”.

Tarjan’s Algorithm

Two important arrays:

- st[u]: starting time (preorder) of node u
- low[u]: min{ st[x] | node u can reach node x using at most one back edge }
  - From **node u**, you can reach all **descendants** of node u by **tree edges**
  - Since we can use **at most one back edge** among node u and its descendants, we would like to use the one to reach a **node x such that st[x] is the smallest**
  - low[u] = min{ low[v] (v is a child of u), st[t] ((u-t) is a back edge) }

Tarjan's Algorithm (Bridge)

1. Find the DFS tree of the graph
    a. Already done when we DFS (just we didn't mark the edges before)

2. For each tree edge(u - v), check if there is a back edge "passing over" (u - v)
    a. How?

Let's modify from a basic DFS!

Tarjan’s Algorithm (Bridge): Step-by-step

Step 1: same as a basic DFS, we record the preorder of node u.

Initially, low[u] = st[u].

(because we only know node u and of course node u can reach itself)

```
vector<pair<int, int>> bridge;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
}
```

Tarjan's Algorithm (Bridge): Step-by-step

Step 2: same as a basic DFS, if node v is not visited, we dfs(v, u) recursively.

This implies (u - v) is a tree edge.

```
vector<pair<int, int>> bridge;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);



    }
    else if (v != par) {

    }
  }
}
```

Tarjan’s Algorithm (Bridge): Step-by-step

Step 3: after dfs(v, u), we already finish processing node v and its descendants.

Since (u - v) is a tree edge, we can update the value of low[u] with low[v].

```
vector<pair<int, int>> bridge;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);


    }
    else if (v != par) {

    }
  }
}
```

Tarjan’s Algorithm (Bridge): Step-by-step

Step 4: now this is the checking: since node v and its descendants have finished processing, if (low[v] > st[u]), it means that node v and all descendants of node v cannot reach node u or ancestor of node u.

This implies (u - v) is a bridge!

```
vector<pair<int, int>> bridge;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > st[u])
        bridge.emplace_back(u, v);
    }
    else if (v != par) {

    }
  }
}
```

Tarjan’s Algorithm (Bridge): Step-by-step

Step 5: now the else part. If v is visited and v isn’t the parent of u, then (u - v) is either a back edge or a forward edge.

low[u] can be updated with st[v].

(forward edge doesn’t affect the result since low[u] <= st[u] < st[v])

Note: NOT low[v] because we are using (u - v), a back edge here. low[v] may already use a back edge. Using more than one back edge violates the definition.

```
vector<pair<int, int>> bridge;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > st[u])
        bridge.emplace_back(u, v);
    }
    else if (v != par) {
      low[u] = min(low[u], st[v]);
    }
  }
}
```

Tarjan's Algorithm (Bridge)

That's it!

Try the algorithm with different graphs to reinforce your understanding.

P.S. some tasks may involve multiple edges, the definition of bridge may depends on the statement. Handle such cases according to the statement ~~so you won't sit until the contest ends and still can't figure out what is going on~~.

Articulation Point (Cut Vertex)

Definition: a node of a graph whose deletion increases the graph's number of connected components.

Understanding how to find bridges by Tarjan’s Algorithm, finding articulation points is more or less the same.

Main question again: how to determine if node u is an articulation point?

Observation 4

Node u is an articulation point if and only if there is **NO back edge** connecting a descendant of v (including v) with an ancestor of u (**excluding** u)[1].

In other words, node u is an articulation point if and only if there is no back edge "passing over" node u.

[1] Why exclude node u? Because even if it can reach node u, deleting node u still splits the graph into two connected components.

Tarjan’s Algorithm (Articulation Point)

1. Find the DFS tree of the graph
2. For each node v, check if there is a back edge “passing over” node u

Let’s modify from previous DFS!

Tarjan's Algorithm (Articulation Point): Step-by-step

Step 1: copy & paste.

```
vector<bool> ap;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;

  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);

      low[u] = min(low[u], low[v]);
      if (low[v] >= st[u] &&           )
        ap[u] = true;
    }
    else if (v != par) {
      low[u] = min(low[u], st[v]);
    }
  }


}
```

Tarjan's Algorithm (Articulation Point): Step-by-step

Step 2: special handle root case.

Why? Because low[v] >= st[u] is always true for u being the root. But then u isn't always an articulation point.

How? Simply check if the root has more than one child or not.

```
vector<bool> ap;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
int tin = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  int root_child_cnt = 0;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      ++root_child_cnt;
      low[u] = min(low[u], low[v]);
      if (low[v] >= st[u] && par != -1)
        ap[u] = true;
    }
    else if (v != par) {
      low[u] = min(low[u], st[v]);
    }
  }
  if (par == -1 && root_child_cnt > 1)
    ap[u] = true;
}
```

Tarjan’s Algorithm (Articulation Point)

That’s it!

Again, try the algorithm with different graphs to reinforce your understanding.

Practice Samples

Bridge:

- https://codeforces.com/contest/118/problem/E

Articulation Point:

- https://www.spoj.com/problems/SUBMERGE/

Practice Samples

Bridge:

- https://codeforces.com/contest/118/problem/E
- Lesson: handle forward edge carefully

Articulation Point:

- https://www.spoj.com/problems/SUBMERGE/
- Lesson: count articulation point carefully without duplication

Practice Tasks

Bridge:

- https://www.spoj.com/problems/EC_P/
- https://codeforces.com/gym/100712/problem/H
- https://codeforces.com/gym/103427/problem/H
- https://codeforces.com/contest/700/problem/C

Articulation Point:

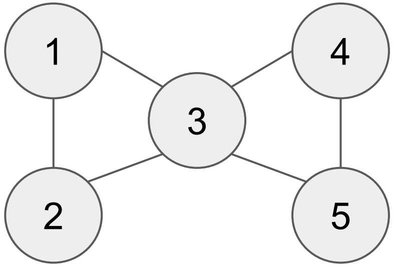
- https://codeforces.com/problemset/problem/193/A
- http://poj.org/problem?id=1523
- https://www.luogu.com.cn/problem/P3469

BCC vs BCC

Bridge-connected Component (2-edge-connected Component) is **DIFFERENT** from Biconnected Component.

[1, 2, 3, 4, 5] is a bridge-connected component.

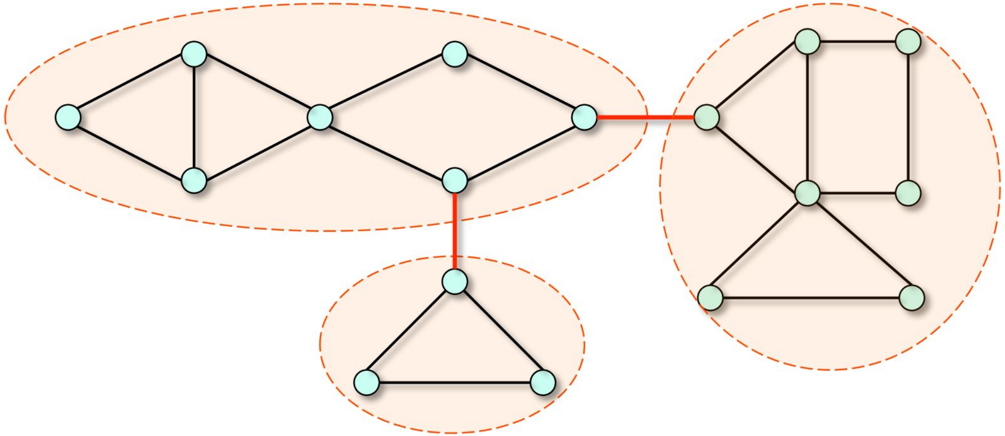
[1, 2, 3] & [3, 4, 5] are biconnected components.



Bridge-connected Component (2-edge-connected Component)

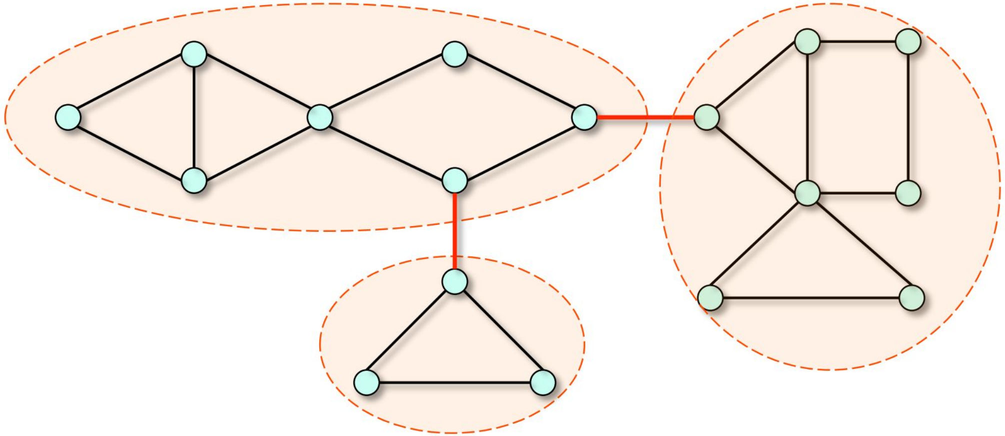
(From Wikipedia) In graph theory, a connected graph is **k-edge-connected** if it remains connected whenever fewer than k edges are removed.

- Shrinking 2-edge-cc gives you a tree
- The graph structure is simpler~~, but the task difficulty may not :(~~



Properties of 2-edge-connected Component

- Within a 2-edge-cc, every pair of nodes has at least two different paths without duplicate edges
- 2-edge-cc partitions nodes & tree edges
    - Hence, the number of edges within a 2-edge-cc can be counted easily during first DFS



Two Phases Algorithm for Finding 2-Edge-CC

1. Mark all bridges
2. DFS the whole graph again but not using bridge
3. Each DFS produces a BCC

Observation 5

If node u is the first node reached of a particular 2-edge-cc in the DFS tree, all other nodes of this particular 2-edge-cc must be the descendants of node u.

This implies that st[u] == low[u].

One Phase Algorithm for Finding 2-Edge-CC

Using observation 5, we can deduce all 2-edge-cc during first DFS.

Practice Sample:
https://codeforces.com/contest/1000/problem/E

```
vector<vector<int>> G;  // Adjacency List
vector<int> st, low, ecc, s;
int tin = 0, k = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
    }
    else if (v != par) {
      low[u] = min(low[u], st[v]);
    }
  }
  if (st[u] == low[u]) {
    ++k;
    for (int x = -1; x != u; s.pop_back()) {
      x = s.back();
      ecc[x] = k;
    }
  }
}
```

Practice Tasks

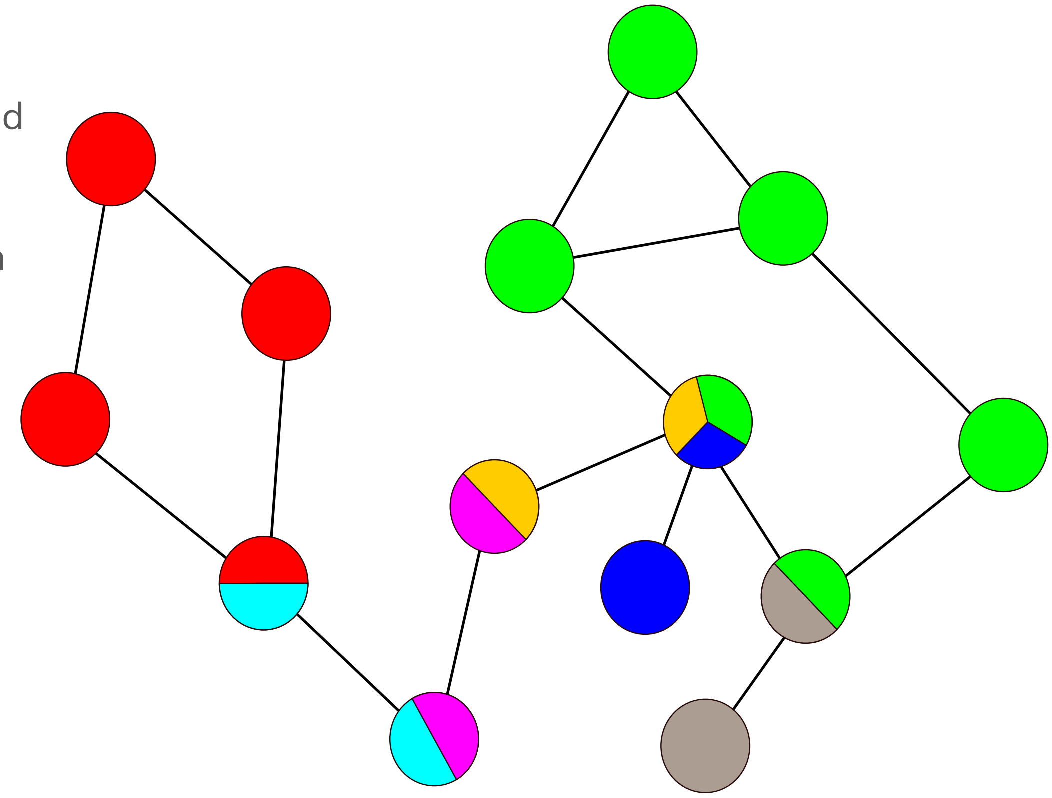
- https://www.spoj.com/problems/GRAFFDEF/
- https://codeforces.com/contest/732/problem/F
- https://codeforces.com/gym/100676/problem/H
- https://codeforces.com/contest/555/problem/E
- https://codeforces.com/contest/652/problem/E

Biconnected Component

(From Wikipedia) In graph theory, a connected graph G is said to be **k-vertex-connected** (or k-connected) if it has **more than k vertices** and remains connected whenever fewer than k vertices are removed.

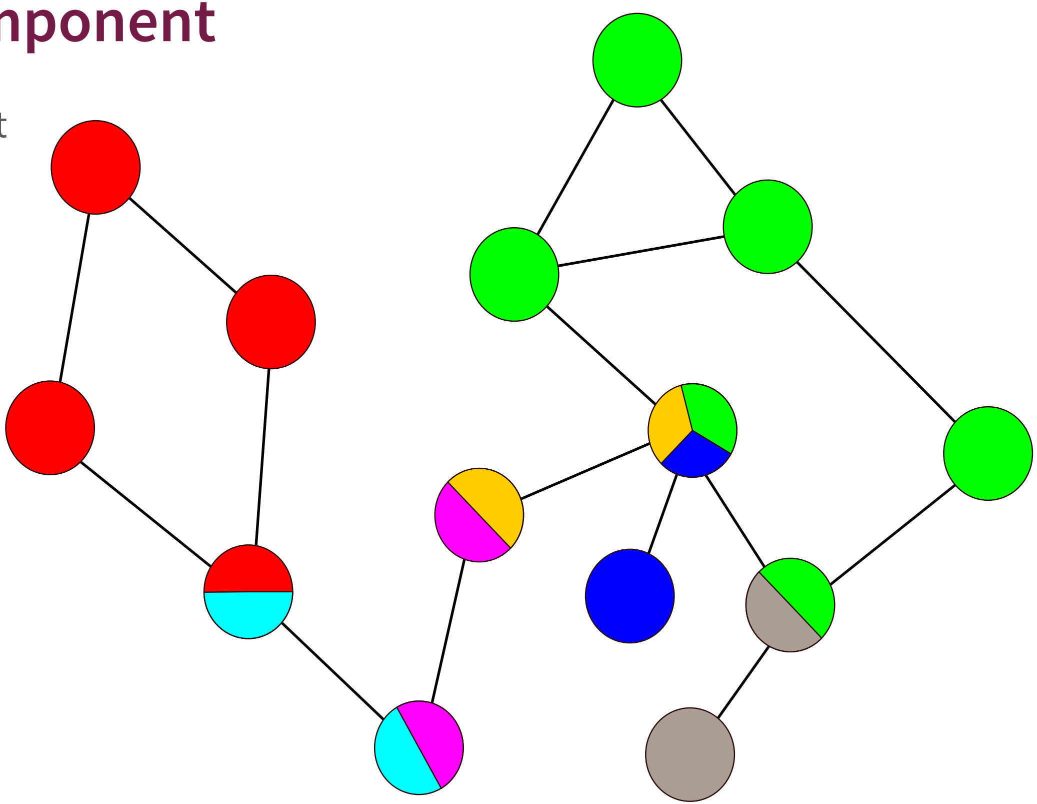
But in biconnected component, a bridge connecting two nodes is still valid.

Each color corresponds to a biconnected component. Multi-colored vertices are cut vertices, and thus belong to multiple biconnected components.



Properties of Biconnected Component

- Within a BCC, every pair of nodes has at least two different paths without duplicate nodes (excluding starting and ending nodes)
- BCC partitions edges
  - Hence, the number of edges within a BCC can be counted by storing edges instead of nodes during first DFS
  - Another way is to find all the BCCs first, then DFS again to count the number of edges
    - All articulation points will be visited at most O(N) times, and every other node will be visited exactly once



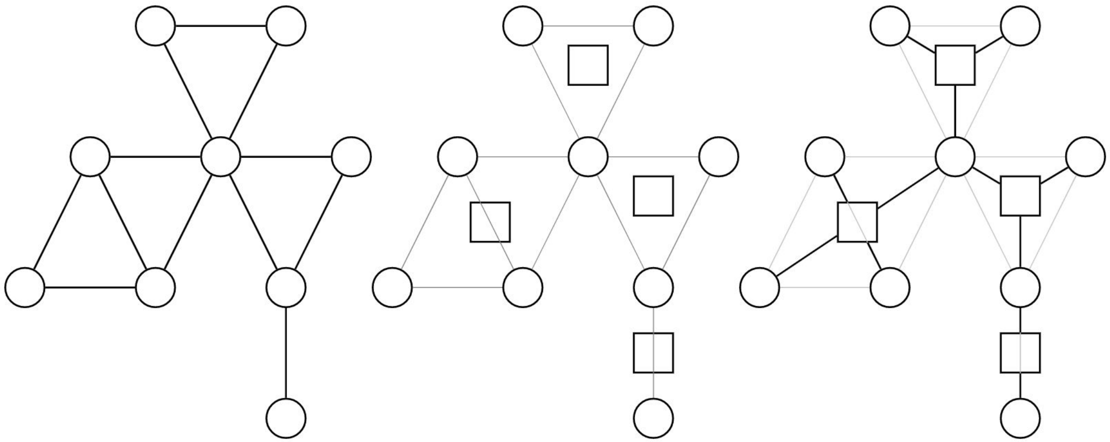
Finding Biconnected Component

```
vector<vector<int>> G, bcc;  // Adjacency List
vector<int> st, low, s;
int tin = 0, k = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] == st[u]) {
        ++k;
        for (int x = -1; x != v; s.pop_back()) {
          x = s.back();
          bcc[x].emplace_back(k);
        }
        bcc[u].emplace_back(k);
      }
    }
    low[u] = min(low[u], st[v]);
  }
}
```

```
vector<vector<int>> G;  // Adjacency List
vector<int> st, low;
vector<set<int>> bcc;
vector<pair<int, int>> s;
int tin = 0, k = 0;

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  for (int v : G[u]) {
    if (!st[v]) {
      s.emplace_back(u, v);
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] == st[u]) {
        ++k;
        for (auto x = pair(-1, -1); x != pair(u, v);) {
          x = s.back();
          bcc[x.first].emplace(k);
          bcc[x.second].emplace(k);
          s.pop_back();
        }
      }
    }
    low[u] = min(low[u], st[v]);
  }
}
```

Block-cut Tree

- For each BCC, create a new source node to connect all the original nodes within the same BCC
- Now you shrink it to a tree
- The graph structure is simpler, ~~but the task difficulty may not :(~~



Block-cut Tree Implementation

Practice Sample:

https://judge.hkoi.org/task/M223J

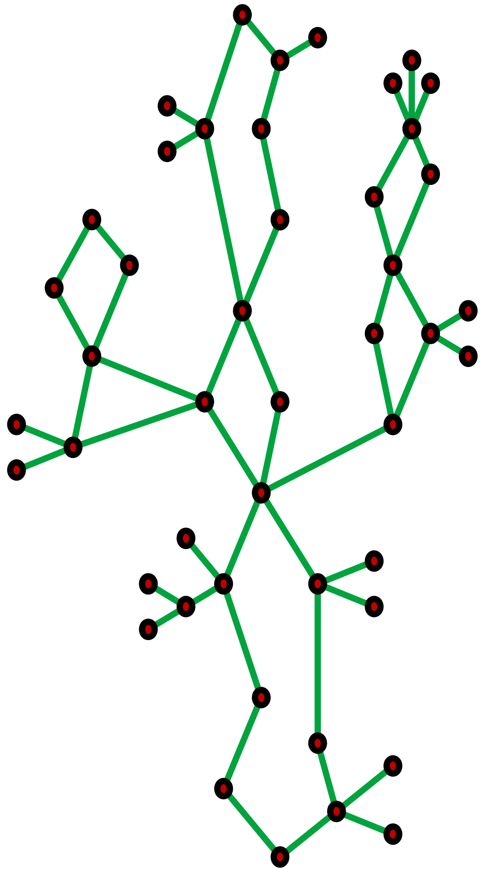
```
vector<vector<int>> G, H;  // Adjacency List
vector<int> st, low, s;
int tin = 0, k = N - 1; // 0-based

void dfs(int u, int par) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] == st[u]) {
        ++k;
        for (int x = -1; x != v; s.pop_back()) {
          x = s.back();
          H[k].emplace_back(x);
          H[x].emplace_back(k);
        }
        H[k].emplace_back(u);
        H[u].emplace_back(k);
      }
    }
    low[u] = min(low[u], st[v]);
  }
}
```

Practice Tasks

- https://oj.uz/problem/view/APIO18_duathlon
- https://dmoj.ca/problem/tle17c1p6
- https://www.luogu.com.cn/problem/P4606

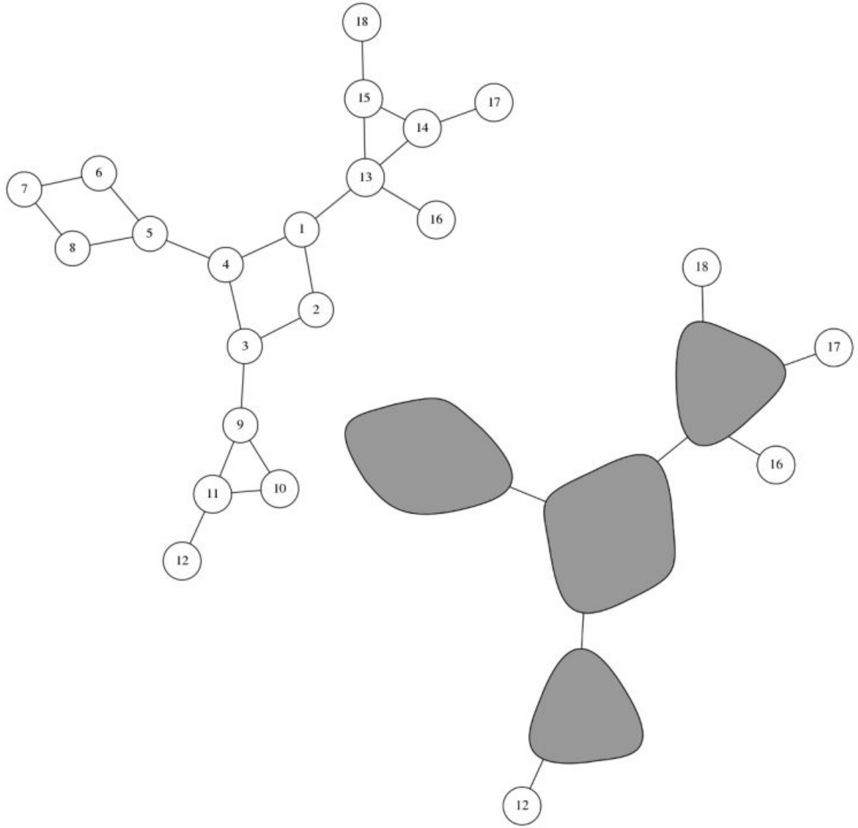
Cactus

(From Wikipedia) In graph theory, a cactus (sometimes called a cactus tree) is a connected graph in which any two simple cycles have at most one vertex in common. Equivalently, it is a connected graph in which every edge belongs to at most one simple cycle.



Cactus

- Can shrink to both 2-edge-cc & BCC
- Depends on tasks (edges / nodes)
- Tree solution + annoying cycles handling
- ~~Toxic lovers~~



Practice Tasks

- https://judge.hkoi.org/task/M1932
- https://codeforces.com/gym/104021/problem/J
- https://codeforces.com/problemset/problem/231/E
- https://codeforces.com/problemset/problem/856/D
- https://codeforces.com/problemset/problem/1578/C
- https://codeforces.com/problemset/problem/1510/C
- https://codeforces.com/problemset/problem/980/F
- https://codeforces.com/problemset/problem/1236/F
- https://www.zybuluo.com/zhangche0526/note/806323

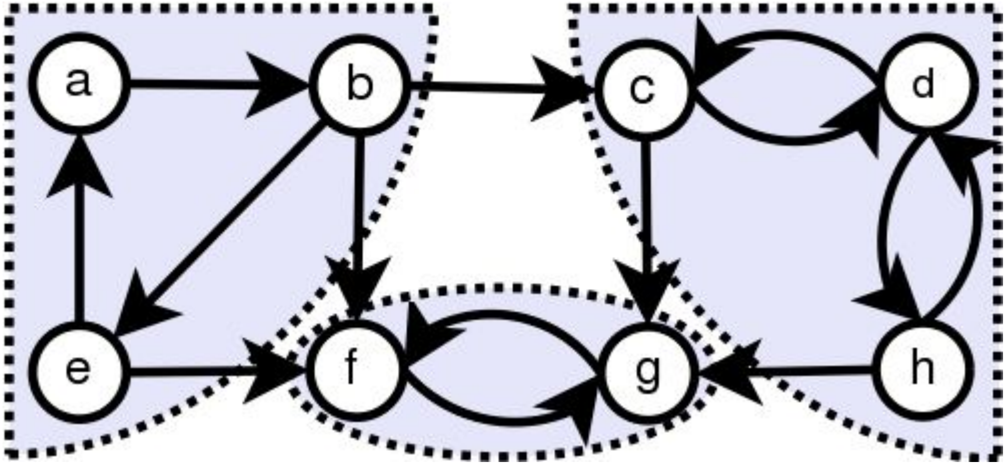
Strongly Connected Component

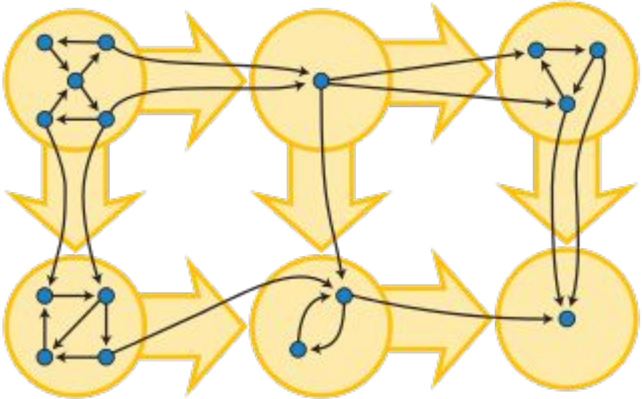
Definition: a **directed** graph is strongly connected if every node u could reach every other node v (including node u itself) in the graph.

A strongly connected component of a directed graph G is a subgraph that is strongly connected, and is maximal (i.e. no additional edges or nodes from G can be included in the subgraph without breaking its SCC property).

Reminder: From now on all the graphs are **directed graphs**.

Examples





From https://en.wikipedia.org/wiki/Strongly_connected_component

Kosaraju's Algorithm (SCC)

- Firstly, DFS all unvisited nodes from 1 to N.
  - dfs(u):
    mark u as visited
    for all unvisited node v connected to u (u → v), dfs(v)
    append u in vector V
- Secondly, DFS(rdfs(u, k)) on the reversed graph from the back of the vector. Each rdfs produces a SCC.
  - rdfs(u, k):
    mark u as visited, mark u belonging to scc group k
    for all unvisited node u connected to v (v → u), rdfs(v, k)

Kosaraju's Algorithm (SCC): Step-by-step

Step 1: DFS all unvisited nodes from 1 to N.

```
vector<bool> vis;
vector<vector<int>> G, R;  // Adjacency List
vector<int> scc, post_order;
int k = 0;

void dfs(int u) {
  vis[u] = true;
  for (int v : G[u])
    if (!vis[v]) dfs(v);
  post_order.emplace_back(u);
}






int main() {
  for (int i = 0; i < N; ++i)
    if (!vis[i]) dfs(i);



}
```

Kosaraju's Algorithm (SCC): Step-by-step

Step 2: DFS(rdfs(u, k)) on the reversed graph from the back of the vector.

Each rdfs produce a SCC group.

```
vector<bool> vis;
vector<vector<int>> G, R;  // Adjacency List
vector<int> scc, post_order;
int k = 0;

void dfs(int u) {
  vis[u] = true;
  for (int v : G[u])
    if (!vis[v]) dfs(v);
  post_order.emplace_back(u);
}

void rdfs(int u) {
  scc[u] = k;
  for (int v : R[u])
    if (!scc[v]) rdfs(v);
}

int main() {
  for (int i = 0; i < N; ++i)
    if (!vis[i]) dfs(i);
  reverse(begin(post_order), end(post_order));
  for (int u : post_order)
    if (!scc[u]) ++k, rdfs(u);
}
```

Strongly Connected Component

- Shrinking SCC gives you a DAG
- The graph structure is simpler~~, but the task difficulty may not :(~~
- Practice Sample:
  - https://codeforces.com/problemset/problem/427/C

Practice Tasks

- https://judge.hkoi.org/task/M1831
- https://www.spoj.com/problems/GOODA/
- https://www.spoj.com/problems/TFRIENDS/
- http://poj.org/problem?id=2186
- https://acm.timus.ru/problem.aspx?space=1&num=1198
- https://codeforces.com/problemset/problem/505/D
- https://codeforces.com/problemset/problem/894/E

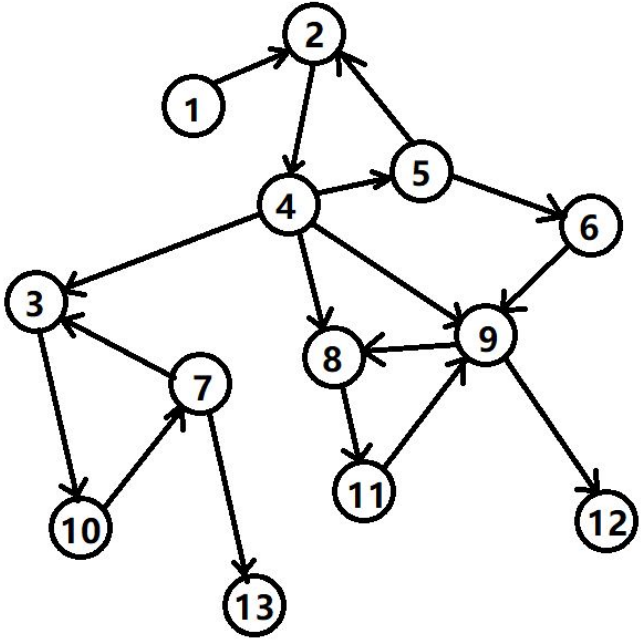
Other Connected Components in Directed Graph

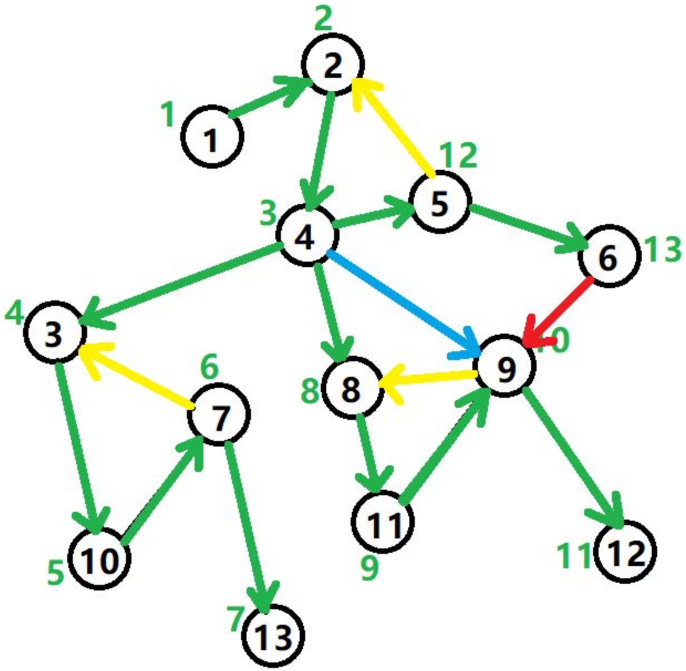
- Weakly connected: connected if treat it as undirected graph
- Unilaterally connected (Semi-connected): for every pair of node u and node v, at least node u could reach node v or node v could reach node u
  - https://judge.hkoi.org/task/M1321

Tarjan’s Algorithm (SCC)

Tarjan’s Algorithm is very powerful that we can use it to find SCC!
(well it’s for finding SCC originally)

But this time the DFS tree becomes a bit more complicated.


1
2
3
4
5
6
7
8
9
10
11
12
13


1
2
3
4
5
6
7
8
9
10
11
12
13

Tarjan’s Algorithm (SCC)

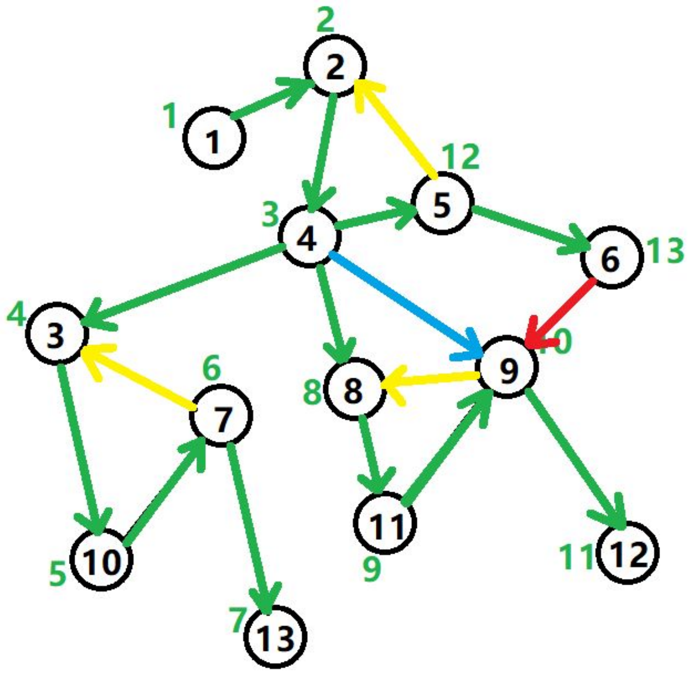
Green: tree edge

Yellow: back edge

Blue: forward edge

Red: cross edge

- Forward edge (u → v):
  - u is NOT the parent of v in DFS tree
  - st[u] < st[v]
  - ed[u] >= ed[v]
- Cross edge (u → v):
  - st[u] > st[v]
  - ed[u] > ed[v]



Observation 6

If node u is the first node reached of a particular SCC in the DFS tree, all other nodes of this particular SCC must be the descendants of node u.

This implies that st[u] == low[u].

Tarjan’s Algorithm (SCC): Step-by-step

Step 1: copy & paste.

```
vector<bool> in_stack;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low, scc, s;
int tin = 0, k = 0;

void dfs(int u) {
  st[u] = low[u] = ++tin;


  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v);
      low[u] = min(low[u], low[v]);
    }
    else if (            ) {
      low[u] = min(low[u], st[v]);
    }
  }








}
```

Tarjan’s Algorithm (SCC): Step-by-step

Step 2: why do we need a stack? Because it keeps descendants of the starting node u in each DFS.

```
vector<bool> in_stack;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low, scc, s;
int tin = 0, k = 0;

void dfs(int u) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  in_stack[u] = true;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v);
      low[u] = min(low[u], low[v]);
    }
    else if (            ) {
      low[u] = min(low[u], st[v]);
    }
  }









}
```

Tarjan’s Algorithm (SCC): Step-by-step

Step 3: why only using edges (u - v) when v is in the stack? Because it ensures that it is a back edge or a forward edge.

(forward edge doesn’t affect result)

If (u - v) is a cross edge, it means that v and its descendants have already been processed.

```
vector<bool> in_stack;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low, scc, s;
int tin = 0, k = 0;

void dfs(int u) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  in_stack[u] = true;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v);
      low[u] = min(low[u], low[v]);
    }
    else if (in_stack[v]) {
      low[u] = min(low[u], st[v]);
    }
  }









}
```

Tarjan's Algorithm (SCC): Step-by-step

Step 4: process a SCC group

```
vector<bool> in_stack;
vector<vector<int>> G;  // Adjacency List
vector<int> st, low, scc, s;
int tin = 0, k = 0;

void dfs(int u) {
  st[u] = low[u] = ++tin;
  s.emplace_back(u);
  in_stack[u] = true;
  for (int v : G[u]) {
    if (!st[v]) {
      dfs(v);
      low[u] = min(low[u], low[v]);
    }
    else if (in_stack[v]) {
      low[u] = min(low[u], st[v]);
    }
  }
  if (st[u] == low[u]) {
    ++k;
    for (int x = -1; x != u; s.pop_back()) {
      x = s.back();
      scc[x] = k;
      in_stack[x] = false;
    }
  }
}
```

Tarjan's Algorithm (SCC)

That's it!

Again & again, try the algorithm with different graphs to reinforce your understanding.

Reference

- Graph (IV) (2022)
  - https://assets.hkoi.org/training2022/g-iv.pdf
- [Tutorial] The DFS tree and its applications: how I found out I really didn't understand bridges
  - https://codeforces.com/blog/entry/68138

Extra Readings

- 2-SAT
  - https://codeforces.com/blog/entry/92977
- Euler Tour Technique
  - https://en.wikipedia.org/wiki/Euler_tour_technique
  - https://usaco.guide/gold/tree-euler?lang=cpp
  - https://codeforces.com/blog/entry/18369
  - https://codeforces.com/blog/entry/102087
- S-T Bridge & Articulation Point
  - https://www.sciencedirect.com/science/article/pii/S0166218X21003334
  - https://judge.hkoi.org/task/S191
  - https://codeforces.com/problemset/problem/1214/D

Extra Readings

For those who want to solve toxic problems in ICPC.

- Dominator Tree
    - https://en.wikipedia.org/wiki/Dominator_(graph_theory)
    - https://codeforces.com/blog/entry/22811
    - https://www.luogu.com.cn/blog/214gtx/zhi-pei-shu-yang-xie
    - https://www.mina.moe/archives/9619
    - https://www.cnblogs.com/fenghaoran/p/dominator_tree.html
    - https://www.luogu.com.cn/problem/P5180
    - https://www.luogu.com.cn/problem/P2597

Extra Readings

For those who want to solve toxic problems in ICPC.

- Bridge & Articulation Point in Directed Graph
  - https://team.inria.fr/erable/files/2014/11/2-connectivity.pdf
- Dynamic Connectivity
  - https://team.inria.fr/erable/files/2014/11/connectivity.pdf
  - https://team.inria.fr/erable/files/2014/11/shortestpaths.pdf