Graph (IV)

Gabriel Liu {gabrielliu2001}
2021-05-01

Reference

- Partly adapted from: Graph (IV) by Ian (2019)
  - https://assets.hkoi.org/training2019/g-iv.pdf

- Partly adapted from: [Tutorial] The DFS tree and its applications: how I found out I really didn't understand bridges
  - https://codeforces.com/blog/entry/68138

Our Powerful Weapons

- DFS
- DFS
- DFS
- ...

- Tree

Agenda

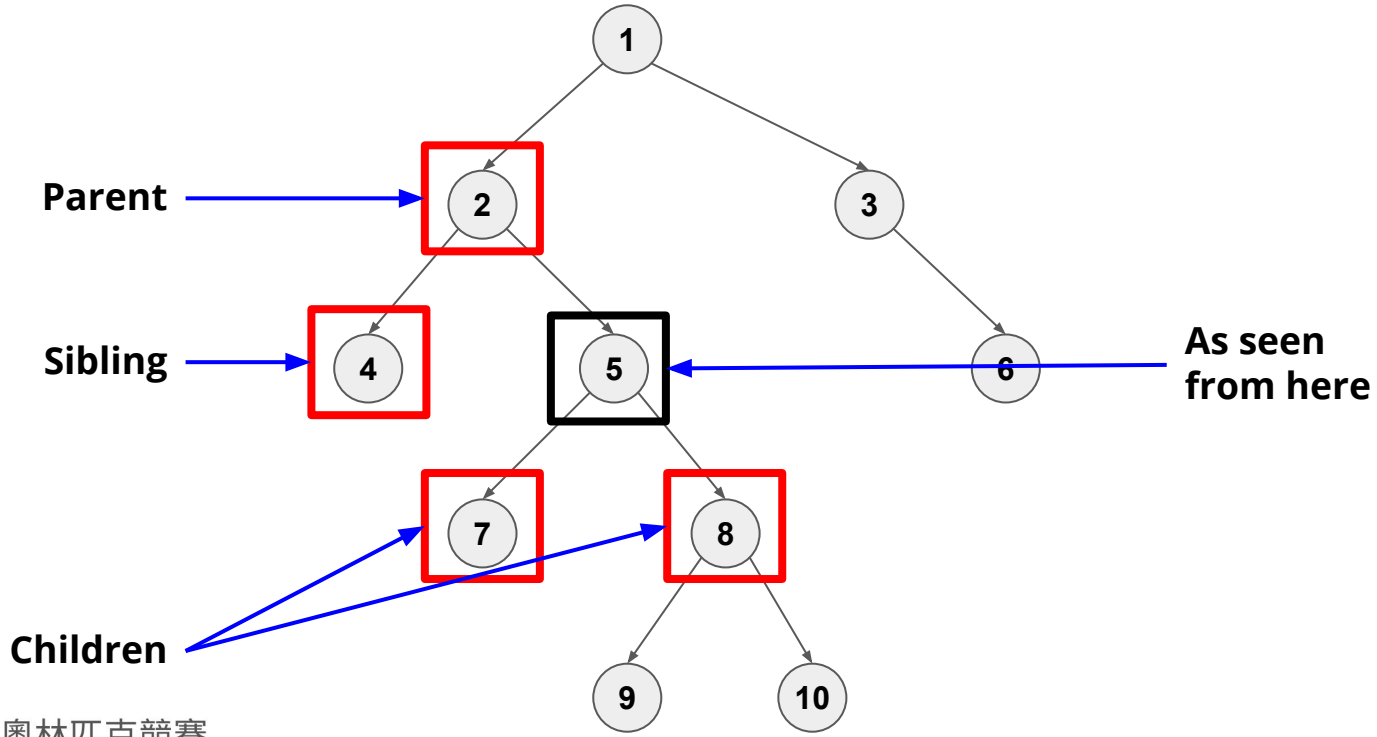
- Undirected Graph:
  - Bridge (Cut Edge)
  - Articulation Point
  - Biconnected Component
  - Bridge-connected Component
- Directed Graph:
  - Strongly Connected Component
  - Weakly Connected Component

Revision: Terms of Tree

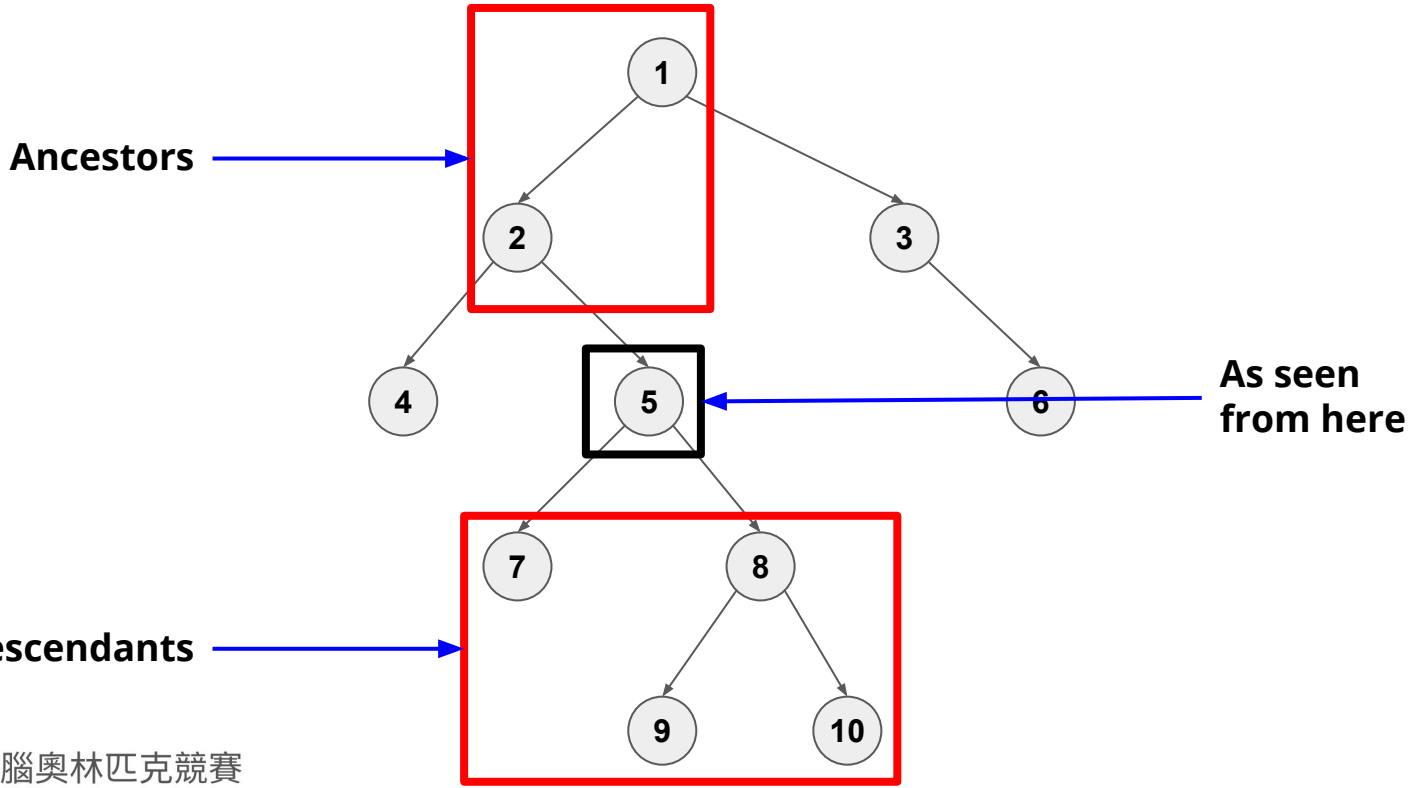
Adapted from Graph (III) by Jamie (2021)

https://assets.hkoi.org/training2021/g-iii.pdf

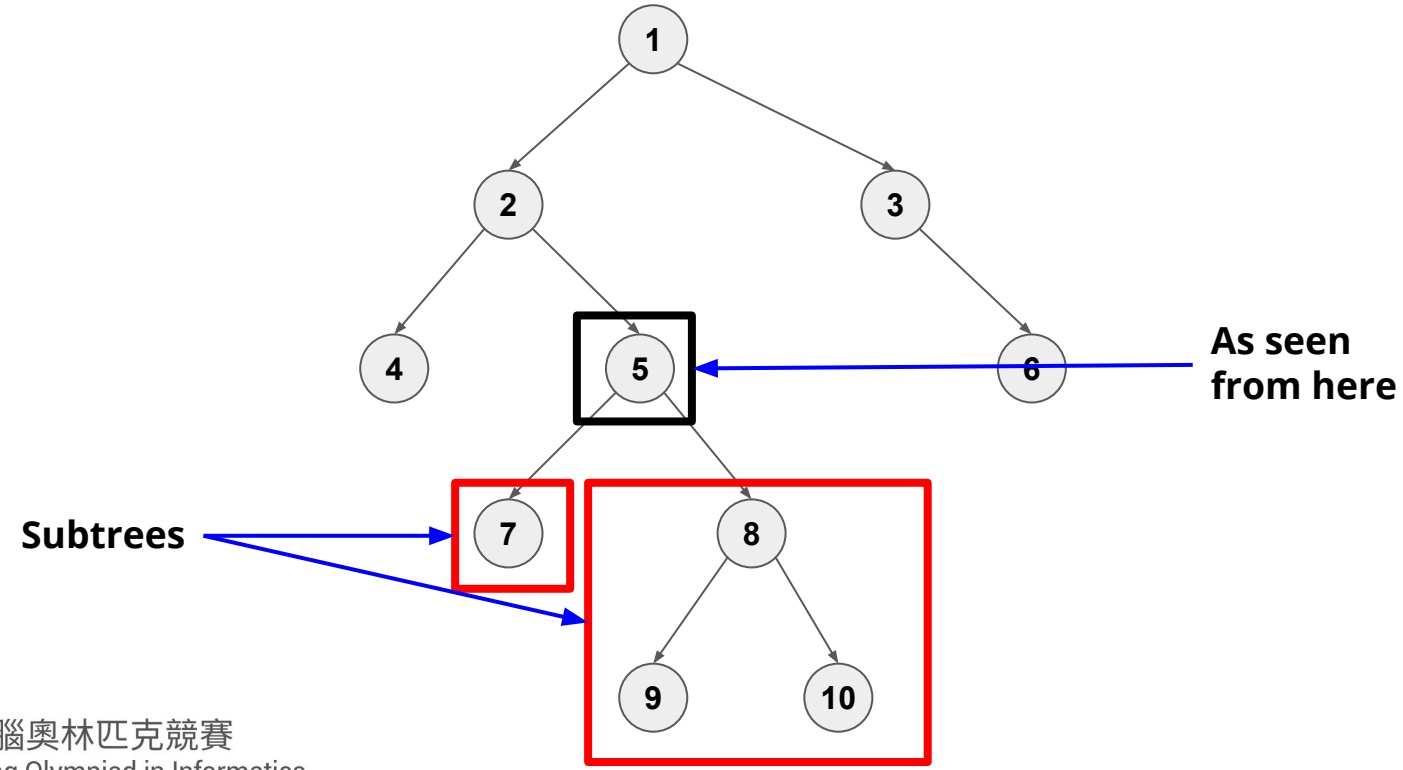
Revision: Terms of Tree



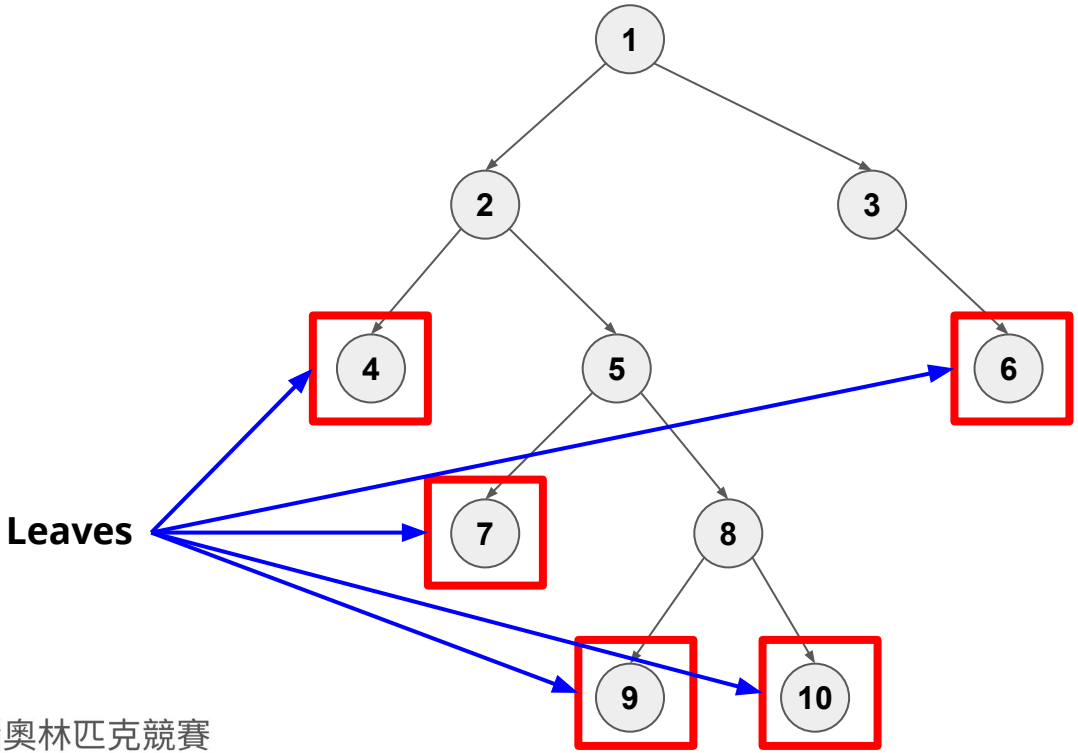
Revision: Terms of Tree



Revision: Terms of Tree



Revision: Terms of Tree



Revision: DFS

```cpp
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);
}
```

Revision: DFS

```cpp
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;
}
```

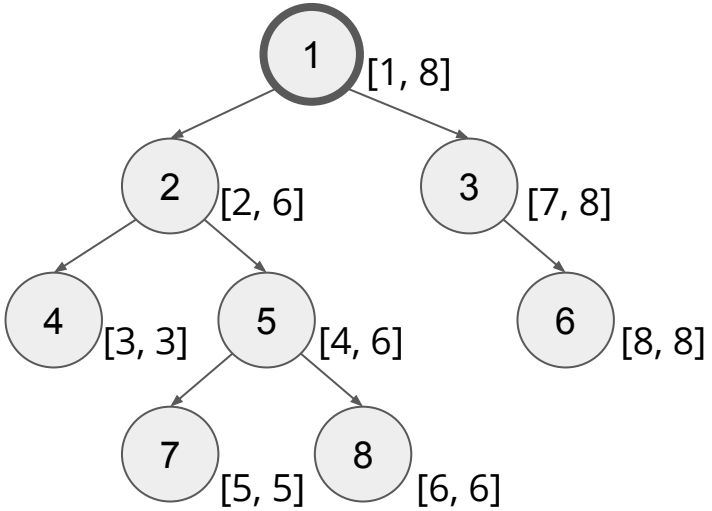
Some tricks

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

st[u] < st[v]

ed[u] >= ed[v]

More: refer to 2019 Graph (IV) ppt



DFS Tree

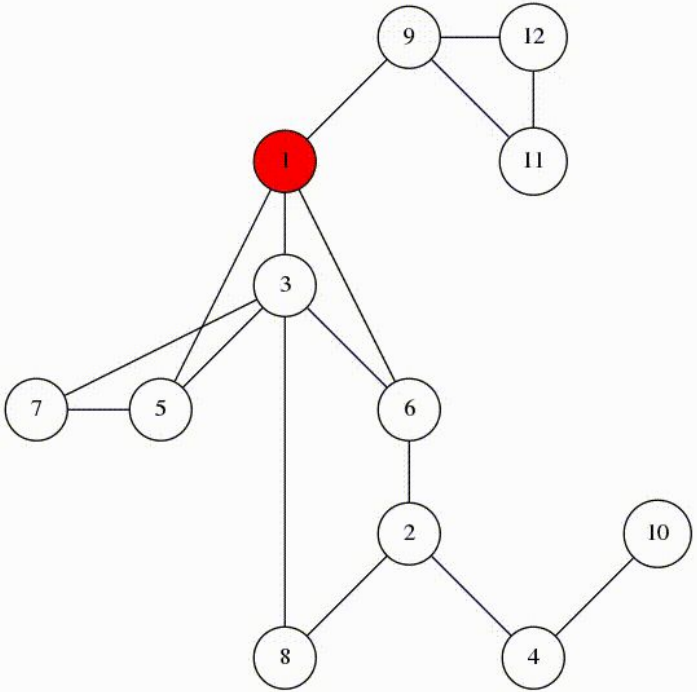
DFS Tree

- 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

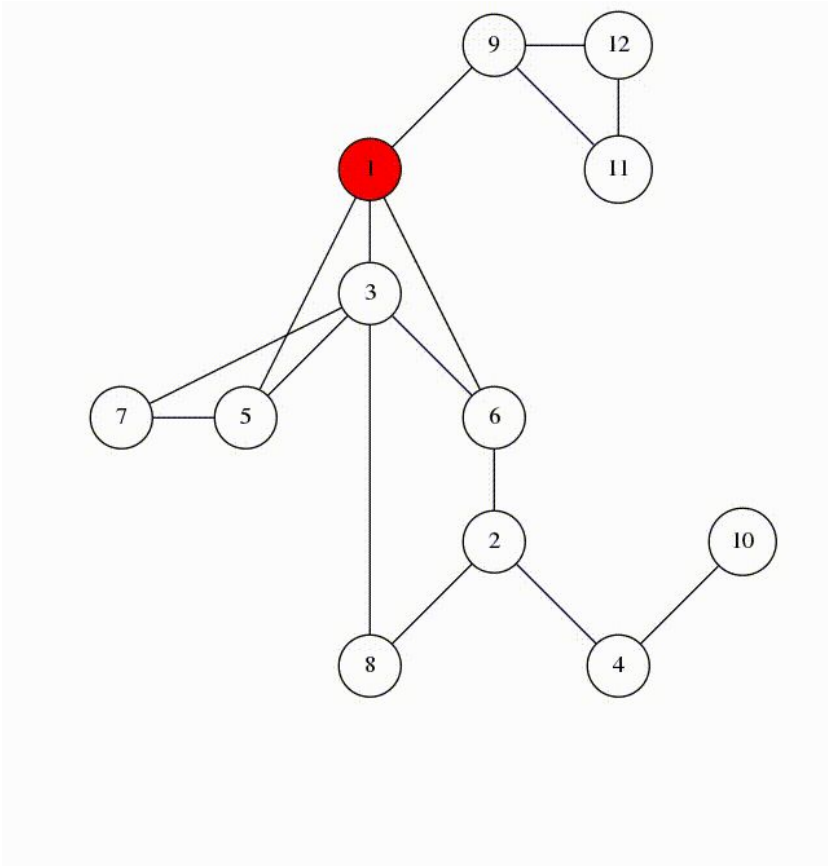
DFS Tree

Consider the graph and function next page:

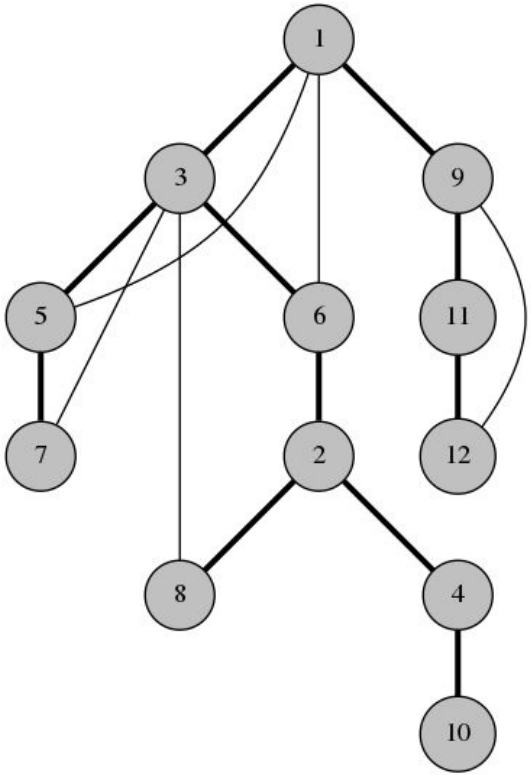
what edges will be marked in line 5 when calling `visit(1)`?




```
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)
```




```
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)
```




```
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)
```

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 specific notation, assume u is parent of v in remaining slides

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 to have some implementation details. The algorithm to be introduced is called “Tarjan’s Algorithm”.

Tarjan's Algorithm

Two important arrays:

1. st[u]: starting time(preorder) of node u
2. low[u]: **smallest st[x]** which node u can reach node x using **at most 1 back edge** from **node u** itself and **descendants of u**

Tarjan's Algorithm (Bridge)

1. Find the DFS tree of the graph
   a. It is already done when we DFS (just we didn't mark the edge 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 basic DFS!

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

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

By definition, low[u] = st[u] by default (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 cnt = 0;

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









}
```

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

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

Notice that this implies (u - v) is a tree edge.

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

void dfs(int u, int par) {
  st[u] = low[u] = ++cnt;
  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 low[u] value with low[v].

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

void dfs(int u, int par) {
  st[u] = low[u] = ++cnt;
  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 v cannot reach node u or ancestor of u.

This implies (u - v) is a bridge!

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

void dfs(int u, int par) {
  st[u] = low[u] = ++cnt;
  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 parent of u, then (u - v) must be a back edge.

low[u] can be updated with 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 1 back edge violates the definition.

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

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

You can also try the algorithm with the graph in previous slide.

Articulation Point

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

With the understanding of 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 increases the graph's number of 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 cnt = 0;

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

  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] &&          )
        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 is root. But then u isn't always an articulation point.

How? Simply check if the root has more than 1 child or not.

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

void dfs(int u, int par) {
  st[u] = low[u] = ++cnt;
  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, you can try the algorithm with the graph in previous slide.

Practice Problems

Bridge:

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

Articulation Point:

- http://poj.org/problem?id=1523
- https://www.spoj.com/problems/SUBMERGE/

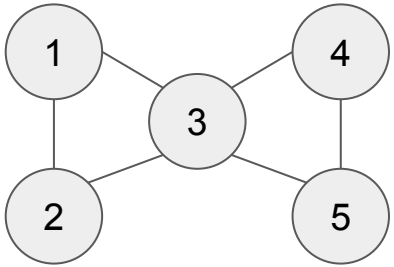
Biconnected Component

- Biconnected component is a maximal biconnected subgraph such that there is no articulation point.
- But for various OI problems, we often use the other definition, the bridge-connected component. When we delete all bridges, the vertices that are connected is in the same bridge-connected component.
- Be sure what you are looking for, biconnected component or bridge-connected component, they sound the same but actually have great difference.

Example

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

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



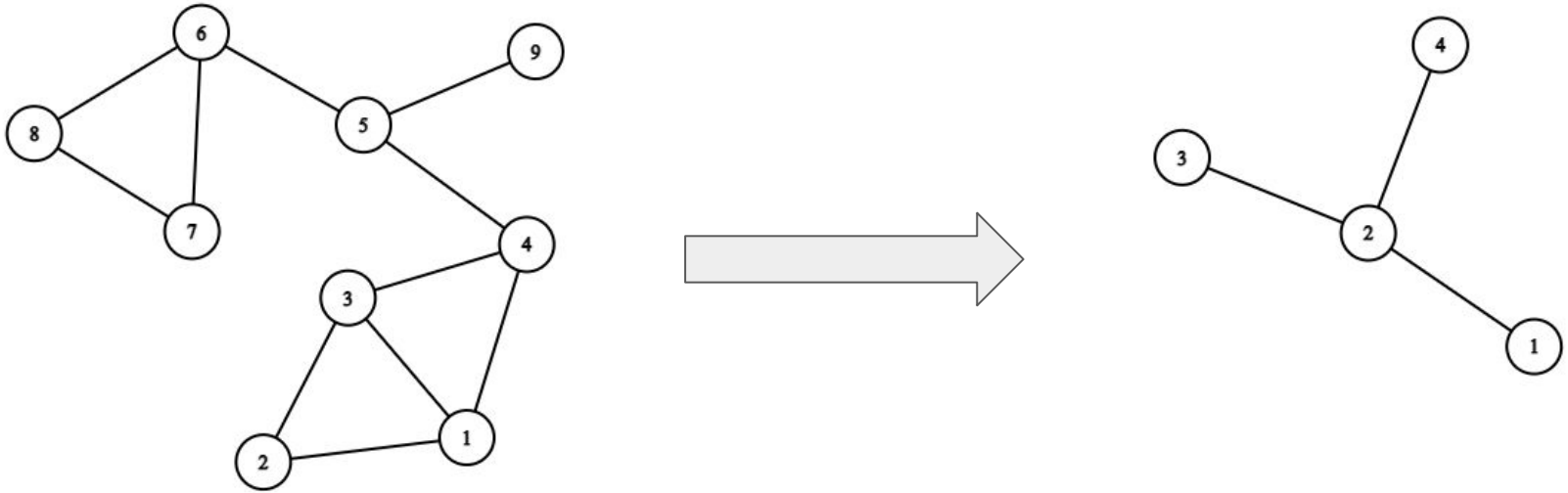
Bridge-connected Component

How to find BCC? Very simple!

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

Shrink Trick

- When we shrink all the bridge-connected components into a node, we would get a tree.
- This help us reduce some hard graph problems into easy tree problems.
- Because in every bridge-connected component, we can find a property: for every node v, it could reach node u in two ways without duplicate edges.
- Problems:
  - https://www.spoj.com/problems/GRAFFDEF/
  - https://codeforces.com/contest/118/problem/E


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

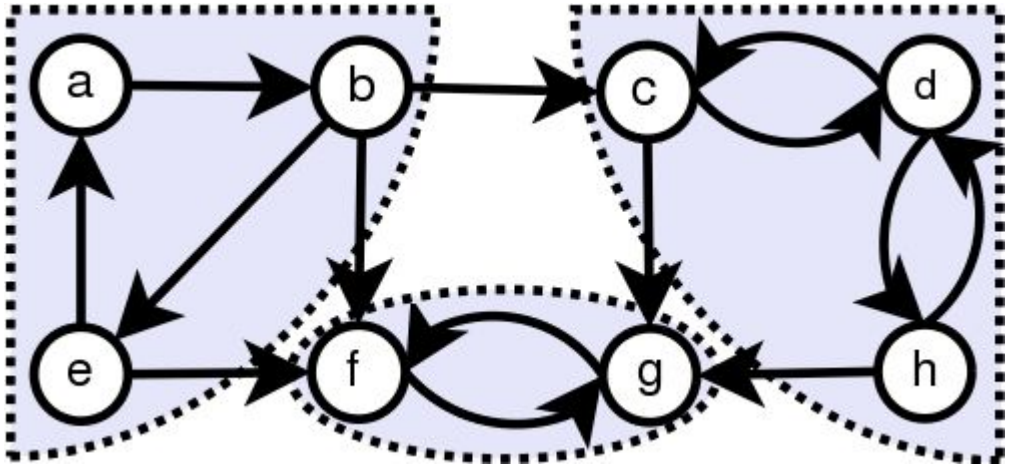
Strongly Connected Component

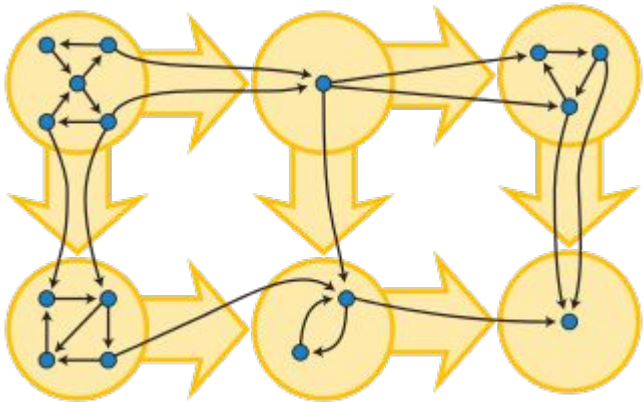
Definition: A graph is strongly connected if every node u could reach every node v (including node u itself) in the graph. Also, the graph need to be maximal (you could not add edges or nodes in this subgraph from the original without breaking this property).

Reminder: From now on all the graphs are **directed graphs**. SCC/WCC are based on directed graphs (bridge and articulation points are based on undirected graphs).

Examples

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





SCC: Kosaraju's Algorithm

- If you just want to memorize a linear algorithm for finding SCC, Kosaraju's Algorithm would be your choice.
- First, do DFS for unvisited node 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
- Second, do a “reverse” DFS(rdfs(u, k)) from the back of the vector, each time increasing parameter k by 1.
  - 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)

Strongly Connected Component

- Again, you could apply shrink trick to solve problems easily.

- Practice Problems:
  - https://judge.hkoi.org/task/M1831
  - https://codeforces.com/problemset/problem/427/C
  - https://www.spoj.com/problems/TFRIENDS/
  - https://acm.timus.ru/problem.aspx?space=1&num=1198

Weakly Connected Component

Definition: A graph is weakly connected if for every pair of node u and node v, at least node u could reach node v or node v could reach node u. The graph need to be maximal too.

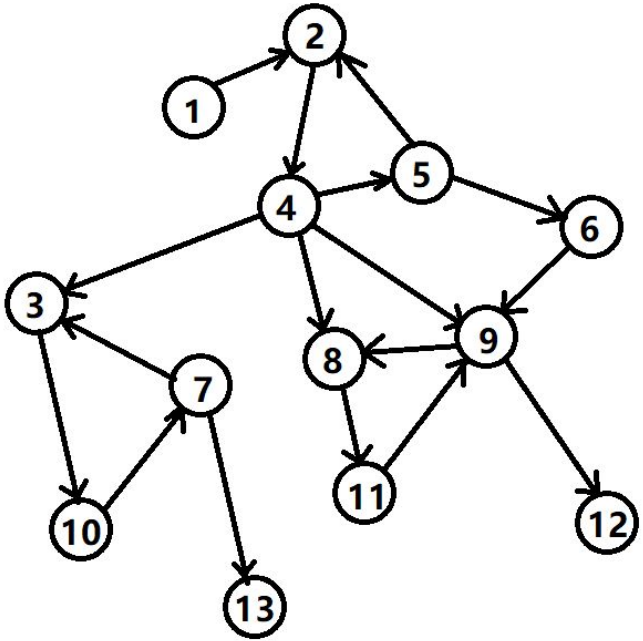
How to find WCC? Well it has been left as practice for a long time, so I would also like to do so XD.

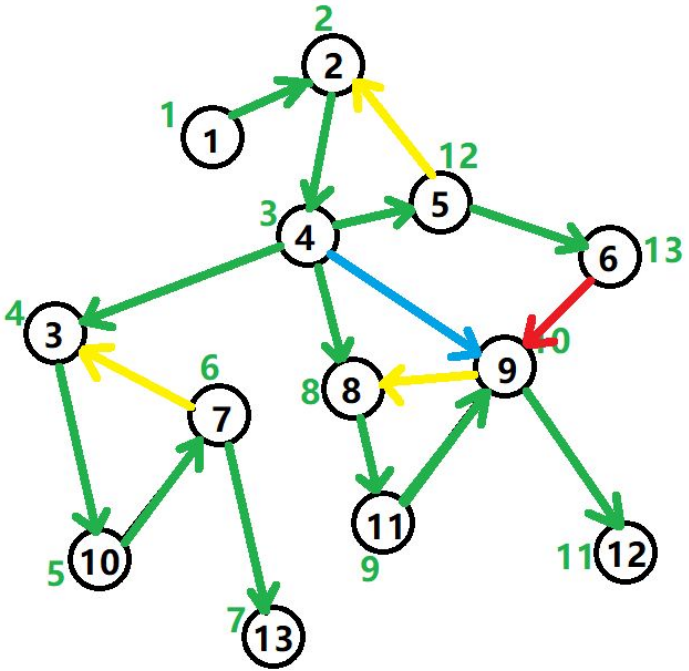
- https://judge.hkoi.org/task/M1321

Tarjan's Algorithm (SCC)

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

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

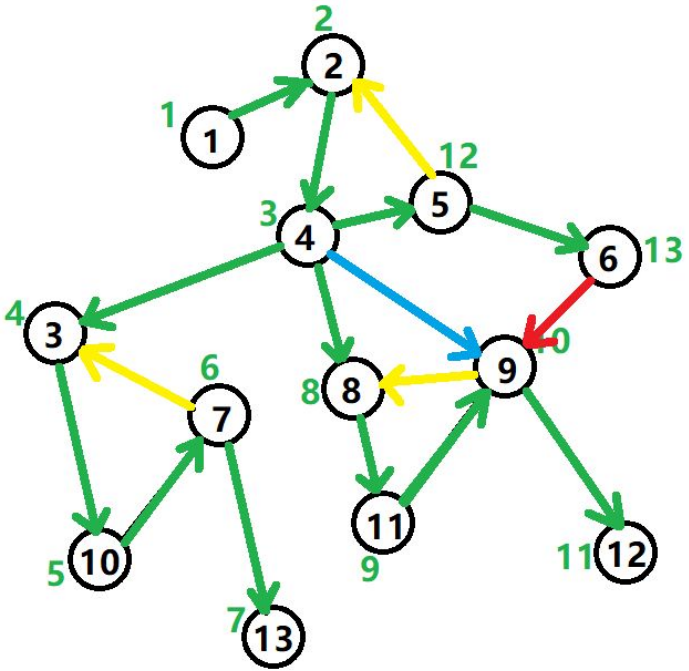

1
2
3
4
5
6
7
8
9
10
11
12
13


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 parent of v
  - st[u] < st[v]
  - ed[u] >= ed[v]
- Cross edge (u → v):
  - st[u] > st[v]
  - ed[u] > ed[v]



Observation 5

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.

It also implies that st[u] == low[u].

Detailed explanation: https://oi-wiki.org/graph/scc/

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 cnt = 0, k = 0;

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


  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 a stack keeps descendants of starting node u in each DFS.

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

void dfs(int u) {
  st[u] = low[u] = ++cnt;
  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) where v is in the stack? Because it ensures that it is a back edge (similar idea to finding bridge/articulation point) or a forward edge (doesn't affect result)

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

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

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

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

void dfs(int u) {
  st[u] = low[u] = ++cnt;
  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;
    while(s.back() != u) {
      scc[s.back()] = k;
      in_stack[s.back()] = false;
      s.pop_back();
    }
    scc[s.back()] = k;
    in_stack[s.back()] = false;
    s.pop_back();
  }
}
```

Tarjan's Algorithm (SCC)

That's it!

Again & again, try the algorithm with the graph in previous slide.

Further Readings

- Tarjan's Algorithm:
  - 2-SAT: https://oi-wiki.org/graph/2-sat/
  - Dominator Tree: https://www.luogu.com.cn/blog/214gtx/zhi-pei-shu-yang-xie

- Shrink Trick:
  - Round Square Tree: https://oi-wiki.org/graph/block-forest/