Competitive Programming
Winter Term 23/24

DFS-Tree and Applications

Kirill Simonov
Algorithm Engineering Group
HPI

Michael Zündorf
Scalable Algorithms Group
KIT
Overview
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*

![DFS-Tree Diagram]

- visited node
- unvisited node
- current node
- back edge
- path to current node (DFS stack)
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those back-edges
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those \textit{back-edges}
when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those back-edges
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*

![Graph with DFS-Tree](image)

- visited node
- unvisited node
- current node
- back edge
- path to current node (DFS stack)
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
- non-skipped edges form a tree → *tree edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those back-edges
- non-skipped edges form a tree → tree edges
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
- non-skipped edges form a tree → *tree edges*
DFS-Tree

- when DFS explores a node, edges to already visited nodes are skipped
- we call those *back-edges*
- non-skipped edges form a tree → *tree edges*

![Diagram of DFS-Tree]

- visited node
- unvisited node
- current node

Back-edges only connect nodes in an ancestor descendant relationship!
Bridges and Cut-Vertices

**Bridge:** edge whose removal disconnects the graph.
Bridges and Cut-Vertices

**Bridge**: edge whose removal disconnects the graph.

![Graph with labeled vertices and edges](image-url)
Bridges and Cut-Vertices

**Bridge**: edge whose removal disconnects the graph.

- bridges are 6–7 and 3–5
**Bridges and Cut-Vertices**

**Bridge:** edge whose removal disconnects the graph.

- bridges are 6–7 and 3–5
**Bridge**: edge whose removal disconnects the graph.

- bridges are 6–7 and 3–5
- a bridge connects a DFS-subtree that has no back-edges going up
Bridges and Cut-Vertices

**Bridge:** edge whose removal disconnects the graph.

- bridges are 6–7 and 3–5
- a bridge connects a **DFS-subtree** that has no back-edges going up

**Cut-Vertex:** vertex whose removal disconnects the graph.

- also called articulation point
Bridges and Cut-Vertices

**Bridge:** edge whose removal disconnects the graph.
- bridges are 6–7 and 3–5
- a bridge connects a **DFS-subtree** that has no back-edges going up

**Cut-Vertex:** vertex whose removal disconnects the graph.
- also called articulation point
- cut-vertices are 2, 3, 6, 7
Bridges and Cut-Vertices

**Bridge:** edge whose removal disconnects the graph.

- bridges are 6–7 and 3–5
- a bridge connects a DFS-subtree that has no back-edges going up

**Cut-Vertex:** vertex whose removal disconnects the graph.

- also callled articulation point
- cut-vertices are 2,3,6,7
- a cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- a bridge connects a DFS-subtree that has no back-edges going up
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- a bridge connects a **DFS-subtree** that has no **back-edges** going up

- **Algorithm Idea:** compute this number of each subtree in tree-DP
Computing Bridges

**Bridge**: edge whose removal disconnects the graph.

- A bridge connects a **DFS-subtree** that has no **back-edges** going up.

**Algorithm Idea**: compute this number of each subtree in tree-DP.

```c
// returns num of upwards back-edges
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1);
        else { // back edge
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==0) { /* BRIDGE!!! */ }
    return up;
}
```
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- A bridge connects a DFS-subtree that has no back-edges going up.

**Algorithm Idea:** compute this number of each subtree in tree-DP.

```cpp
// returns num of upwards back-edges
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1);
        else { // back edge
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==0) { /* BRIDGE!!! */ }
    return up;
}
```

Any Mistakes?
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- a bridge connects a DFS-subtree that has no back-edges going up
- **Algorithm Idea:** compute this number of each subtree in tree-DP

```c
// returns num of upwards back-edges
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1);
        else { // back edge
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==0) { /* BRIDGE!!! */ }
    return up;
}
```

edge to parent is counted as back-edge

Any Mistakes?
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- a bridge connects a **DFS-subtree** that has no **back-edges** going up

- **Algorithm Idea:** compute this number of each subtree in tree-DP

```c
// returns num of upwards back-edges
int dfs(int v, int p, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) { // tree edge
            up += dfs(v2, v, d+1);
        } else if(v2!=p) { // back edge
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==0) { /* BRIDGE!!! */ }
    return up;
}
```

edge to parent is counted as back-edge

Any Mistakes?
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- A bridge connects a **DFS-subtree** that has no **back-edges** going up

- **Algorithm Idea:** compute this number of each subtree in tree-DP

```cpp
// returns num of upwards back-edges
int dfs(int v, int p, int d) {
  int up = 0;
  lvl[v] = d;
  for(int v2 : adj[v]) {
    if(lvl[v2]==-1) // tree edge
      up += dfs(v2, v, d+1);
    else if(v2!=p) {
      // back edge
      if(lvl[v2] < d) up++; // up
      if(lvl[v2] > d) up--; // down
    }
  }
  if(up==0) { /* BRIDGE!!! */ }
  return up;
}
```

Multigraphs?
Computing Bridges

**Bridge**: edge whose removal disconnects the graph.

- a bridge connects a **DFS-subtree** that has no **back-edges** going up
- **Algorithm Idea**: compute this number of each subtree in tree-DP

```
// returns num of upwards back-edges
int dfs(int v, int p, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, v, d+1);
        else if(v2!=p) {
            // back edge
            if(lvl[v2] < d) up++;
            if(lvl[v2] > d) up--;
        }
    }
    if(up==0) { /* BRIDGE!!! */ }
    return up;
}
```

Multigraphs?
multiedges to parent all ignored from below
Computing Bridges

**Bridge**: edge whose removal disconnects the graph.

- A bridge connects a DFS-subtree that has no back-edges going up

**Algorithm Idea**: compute this number of each subtree in tree-DP

```cpp
// returns num of upwards back-edges +1
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1) - 1;
        else { // back edge or parent
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==1) { /* BRIDGE!!! */ }
    return up;
}
```

Multigraphs?

multiedges to parent all ignored from below
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- A bridge connects a DFS-subtree that has no back-edges going up.
- **Algorithm Idea:** compute this number of each subtree in tree-DP.

```cpp
// returns num of upwards back-edges +1
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2] == -1) // tree edge
            up += dfs(v2, d+1) - 1;
        else { // back edge or parent
            if(lvl[v2] < d) up++; // up
            if(lvl[v2] > d) up--; // down
        }
    }
    if(up==1) { /* BRIDGE!!! */ }
    return up;
}
```

correct and works for multigraphs but ugly
Computing Bridges

**Bridge**: edge whose removal disconnects the graph.

- A bridge connects a DFS-subtree that has no back-edges going up.
- **Algorithm Idea**: compute this number of each subtree in tree-DP.
- Simpler: a bridge connects a DFS-subtree that has exactly one edge going up (the bridge itself).

```java
// returns num of upwards back-edges +1
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1) - 1;
        else if(lvl[v2] < d) up++; // up
        else if(lvl[v2] > d) up--; // down
    }
    if(up==1) { /* BRIDGE!!! */ }
    return up;
}
```

Correct and works for multigraphs but ugly.
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- A bridge connects a DFS-subtree that has no back-edges going up.

- **Algorithm Idea:** compute this number of each subtree in tree-DP.

- Simpler: a bridge connects a DFS-subtree that has exactly one edge going up (the bridge itself).

```cpp
// returns num of upwards edges
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1);
        if(lvl[v2] < d) up++; // up
        if(lvl[v2] > d) up--; // down
    }
    if(up==1) { /* BRIDGE!!! */ }
    return up;
}
```
Computing Bridges

**Bridge:** edge whose removal disconnects the graph.

- a bridge connects a DFS-subtree that has no back-edges going up
- **Algorithm Idea:** compute this number of each subtree in tree-DP
- simpler: a bridge connects a DFS-subtree that has exactly one edge going up (the bridge itself)
- most other tutorials find the highest back-edge of each subtree, which also works for multi-graphs

```cpp
// returns num of upwards edges
int dfs(int v, int d) {
    int up = 0;
    lvl[v] = d;
    for(int v2 : adj[v]) {
        if(lvl[v2]==-1) // tree edge
            up += dfs(v2, d+1);
        if(lvl[v2] < d) up++; // up
        if(lvl[v2] > d) up--; // down
    }
    if(up==1) { /* BRIDGE!!! */ }
    return up;
}
cp-algorithms.com/graph/bridge-searching.html
codeforces.com/blog/entry/68138
```
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)

---

Competitive Programming | Algorithm Engineering Group (HPI) & Scalable Algorithms Group (KIT)
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
2-Edge-Connected Components

- Bridges partition the nodes of a graph into 2-edge-connected components.
- Those components form a tree :)
- Can be computed by removing all bridges and then computing connected components.
- Instead, look at a leaf in component tree.
- At some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge.
- Use a stack to track visited nodes.
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)  
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
- use a stack to track visited nodes
- allows to identify the nodes visited between entering and exiting the current component
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
- use a stack to track visited nodes
- allows to identify the nodes visited between entering and exiting the current component
- eat away components from the outside
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
- use a stack to track visited nodes
- allows to identify the nodes visited between entering and exiting the current component
- eat away components from the outside
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
- use a stack to track visited nodes
- allows to identify the nodes visited between entering and exiting the current component
- eat away components from the outside
2-Edge-Connected Components

- bridges partition the nodes of a graph into 2-edge-connected components
- those components form a tree :)
- can be computed by removing all bridges and then computing connected components
- instead, look at a leaf in component tree
- at some point DFS enters the component via the bridge, then visits all its nodes, and then backtracks back over the bridge
- use a stack to track visited nodes
- allows to identify the nodes visited between entering and exiting the current component
- **eat away** components from the outside

```cpp
vector<int> st;
vector<vector<int>> comps;
int dfs(int v, int d) {
    int st_pos = size(st);
st.push_back(v);
    //TODO: ... || up==1) { // exiting comp
        comps.emplace_back(
            begin(st)+st_pos, end(st));
        st.resize(st_pos);
    }
    return up;
}
```
Cut-Vertices

**Cut-Vertex**: vertex whose removal disconnects the graph.

- A cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors.
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- a cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors
- simpler: a cut-vertex has at least one DFS-subtree that has no edges going to his ancestors
Cut-Vertices

**Cut-Vertex**: vertex whose removal disconnects the graph.

- A cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors.
- Simpler: a cut-vertex has at least one DFS-subtree that has no edges going to his ancestors.

**Algorithm Idea**: compute highest edge out of each subtree.
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- a cut-vertex has at least one **DFS-subtree** that has no **back-edges** going to his ancestors
- simpler: a cut-vertex has at least one **DFS-subtree** that has no edges going to his ancestors
- **Algorithm Idea:** compute *highest* edge out of each subtree
- for *height* we track discover times
Cut-Vertices

**Cut-Vertex**: vertex whose removal disconnects the graph.

- a cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors.
- simpler: a cut-vertex has at least one DFS-subtree that has no edges going to his ancestors.
- **Algorithm Idea**: compute highest edge out of each subtree
  - for height we track discover times
  - smaller discover time $\rightarrow$ higher in DFS-tree
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- A cut-vertex has at least one **DFS-subtree** that has no back-edges going to his ancestors.
- Simpler: A cut-vertex has at least one **DFS-subtree** that has no edges going to his ancestors.
- **Algorithm Idea:** compute *highest* edge out of each subtree.
- For *height* we track discover times.
- Smaller discover time → higher in DFS-tree.

```c
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) { // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) {
                // v is cut-vertex
            }
        }
        low = min(low, disc[v2]);
    }
    return low;
}
```

// returns earliest discover time
// reachable via 1 edge from my subtree
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- a cut-vertex has at least one **DFS-subtree** that has no back-edges going to his ancestors
- simpler: a cut-vertex has at least one **DFS-subtree** that has no edges going to his ancestors
- **Algorithm Idea:** compute *highest* edge out of each subtree
- for *height* we track discover times
- smaller discover time $\rightarrow$ higher in DFS-tree

```cpp
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) { // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) {
                // v is cut-vertex
            }
        }
        low = min(low, disc[v2]);
    }
    return low;
}
```

There is an edge case missing.
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- A cut-vertex has at least one **DFS-subtree** that has no back-edges going to his ancestors.
- Simpler: a cut-vertex has at least one **DFS-subtree** that has no edges going to his ancestors.

**Algorithm Idea:** compute highest edge out of each subtree.

- For **height** we track discover times.
- Smaller discover time $\rightarrow$ higher in DFS-tree.

```c
// returns earliest discover time reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) {
            // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) {
                // exiting a bicon. comp.
                
            }
        }
        low = min(low, disc[v2]);
    }
    return low;
}
```

There is an edge case missing.
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- a cut-vertex has at least one DFS-subtree that has no back-edges going to his ancestors
- simpler: a cut-vertex has at least one DFS-subtree that has no edges going to his ancestors
- **Algorithm Idea:** compute *highest* edge out of each subtree
  - for *height* we track discover times
  - smaller discover time $\rightarrow$ higher in DFS-tree
- **Edge Case:**
  - root is cut-vertex $\iff$ it has 2+ subtrees

```cpp
// returns earliest discover time reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    for (int v2 : adj[v]) {
        if (disc[v2] == -1) {
            // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if (low2 >= disc[v]) {
                // exiting a bicon. comp.
            }
        }
        low = min(low, disc[v2]);
    }
    return low;
}
```

There is an edge case missing.
Cut-Vertices

**Cut-Vertex:** vertex whose removal disconnects the graph.

- A cut-vertex has at least one **DFS-subtree** that has no back-edges going to its ancestors.
- Simpler: A cut-vertex has at least one **DFS-subtree** that has no edges going to its ancestors.
- **Algorithm Idea:** Compute highest edge out of each subtree.
- For **height** we track discover times.
- Smaller discover time $\rightarrow$ higher in DFS-tree.
- **Edge Case:** Root is cut-vertex $\iff$ it has 2+ subtrees.

```cpp
// returns earliest discover time reachable via 1 edge from my subtree
int dfs(int v, int p) {
    int low = disc[v] = time++;
    int kids = 0;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) {
            // tree edge
            kids++;
            int low2 = dfs(v2,v);
            low = min(low, low2);
            if(low2>=disc[v] &&
               (p!=-1 || kids>1)) {
                // v is cut-vertex
                }
            }
        }
    return low;
}
```

Simpler: A cut-vertex has at least one DFS-subtree that has no edges going to its ancestors.
Cut-Vertices

Cut-Vertex: vertex whose removal disconnects the graph.

- a cut-vertex has at least one **DFS-subtree** that has no back-edges going to his ancestors
- simpler: a cut-vertex has at least one **DFS-subtree** that has no edges going to his ancestors
- **Algorithm Idea:** compute *highest* edge out of each subtree
- for *height* we track discover times
- smaller discover time $\rightarrow$ higher in DFS-tree
- **Edge Case:** root is cut-vertex $\iff$ it has 2+ subtrees

```cpp
// returns earliest discover time reachable via 1 edge from my subtree
int dfs(int v, int p) {
    int low = disc[v] = time++;
    int kids = 0;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) {
            // tree edge
            kids++;
            int low2 = dfs(v2,v);
            low = min(low, low2);
            if(low2>=disc[v] &&
               (p!=-1 || kids>1)) {
                // v is cut-vertex
                }
            }
        low = min(low, disc[v2]);
    }
    return low;
}
```

// returns earliest discover time reachable via 1 edge from my subtree
int dfs(int v, int p) {
    int low = disc[v] = time++;
    int kids = 0;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) {
            // tree edge
            kids++;
            int low2 = dfs(v2,v);
            low = min(low, low2);
            if(low2>=disc[v] &&
               (p!=-1 || kids>1)) {
                // v is cut-vertex
                }
            }
        low = min(low, disc[v2]);
    }
    return low;
}`
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)
- cannot assign each node a unique component
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

- cannot assign each node a unique component
- instead assign edges to components
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

![Graph partition into biconnected subgraphs](image)

- cannot assign each node a unique component
- instead assign edges to components
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

\[
\begin{array}{c}
\text{cannot assign each node a unique component} \\
\text{instead assign edges to components} \\
\text{components are connected at cut-vertices and form a tree :)}
\end{array}
\]
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

\[ \text{cannot assign each node a unique component} \]

\[ \text{instead assign edges to components} \]

\[ \text{components are connected at cut-vertices and form a tree :) } \]
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

A graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex).

- components are connected at cut-vertices and form a tree :)
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

![Diagram of 2-Vertex-Connected Components]

- cannot assign each node a unique component
- instead assign edges to components
- components are connected at cut-vertices and form a tree :)

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

A graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex). Components are connected at cut-vertices and form a tree :)

- cannot assign each node a unique component
- instead assign edges to components
- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

\[ 	ext{Diagram of a graph with cut-vertices and components} \]

- cannot assign each node a unique component
- instead assign edges to components
- components are connected at cut-vertices and form a tree :)

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide

\[ \text{Why?} \]
2-Vertex-Connected Components

- also called biconnected components, or 2-connected components
- partition graph into biconnected subgraphs
- a graph is biconnected if it cannot be disconnected by removing one node (i.e. has no cut-vertex)

![Graph Example]

- cannot assign each node a unique component
- instead assign edges to components
- components are connected at cut-vertices and form a tree :)

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide
- because when backtracking out of a subtree with no higher back-edges, then you always exit a component
2-Vertex-Connected Components

// returns earliest discover time
// reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) { // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) { // exiting comp
                comps.emplace_back(begin(st)+st_pos, end(st));
                st.resize(st_pos);
            }
        }
        if(disc[v2]<disc[v]) // from below
            st.push_back({v,v2});
        low = min(low, disc[v2]);
    }
    return low;
}

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide
- because when backtracking out of a subtree with no higher back-edges, then you always exit a component
2-Vertex-Connected Components

// returns earliest discover time
// reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) { // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) { // exiting comp
                st.push_back({v,v2});
                low = min(low, disc[v2]);
            }
        }
        if(disc[v2]<disc[v]) // from below
            st.push_back({v,v2});
            low = min(low, disc[v2]);
    }
    return low;
}

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide
- because when backtracking out of a subtree with no higher back-edges, then you always exit a component
2-Vertex-Connected Components

// returns earliest discover time
// reachable via 1 edge from my subtree
int dfs(int v) {
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        int st_pos = size(st);
        if(disc[v2]==-1) { // tree edge
            int low2 = dfs(v2);
            low = min(low, low2);
            if(low2>=disc[v]) {
                // exiting comp
                comps.emplace_back(begin(st)+st_pos, end(st));
                st.resize(st_pos);
            }
        } else if(disc[v2]<disc[v]) { // from below
            st.push_back({v,v2});
            low = min(low, disc[v2]);
        }
    }
    return low;
}

- as for 2-edge-connected components, use a stack but push edges instead of vertices
- be careful not to push edges from both sides
- small trick: you can skip the root-edge-case from last slide
- because when backtracking out of a subtree with no higher back-edges, then you always exit a component
Directed DFS-Tree

- directed DFS different than undirected
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
- sometimes more difficult because there are more kinds of edges
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
- sometimes more difficult because there are more kinds of edges

![Directed DFS-Tree Diagram]

- back-edge
- forward-edge
- cross-edge
- tree-edge
Directed DFS-Tree

- Directed DFS different than undirected
- Sometimes simpler because you don’t see the same edge from both sides
- Sometimes more difficult because there are more kinds of edges

Diagram:
- Back-edges point to an ancestor
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
- sometimes more difficult because there are more kinds of edges

![Directed DFS-Tree Diagram]

- back-edges point to an ancestor
- forward-edges point to a descendant
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
- sometimes more difficult because there are more kinds of edges

- back-edges point to an ancestor
- forward-edges point to a descendant
- cross-edges point to a node that the DFS already finished processing
Directed DFS-Tree

- directed DFS different than undirected
- sometimes simpler because you don’t see the same edge from both sides
- sometimes more difficult because there are more kinds of edges

back-edges point to an ancestor

forward-edges point to a descendant

cross-edges point to a node that the DFS already finished processing

multiple DFS needed to explore the full graph unless it’s strongly connected
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)

Diagram:

[Diagram showing strongly connected components]
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)

```
  1  2  3  4  5
  ▼  ▼  ▼  ▼  ▼
  ▲  ▲  ▲  ▲  ▲
```

Strongly Connected Components (SCCs)

- a graph is strongly connected, if we can reach each node from every other node
- we can partition a graph into strongly connected components
- the SCCs form a DAG :)
- let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes)
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let's consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :) 
- Let's consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- a graph is strongly connected, if we can reach each node from every other node
- we can partition a graph into strongly connected components
- the SCCs form a DAG :)
- let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes)
- we say that we exit a SCC when we exit its last vertex
- on a DAG, exit times of DFS are in reverse topological order (sink first)
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let's consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.

- We can partition a graph into strongly connected components.

- The SCCs form a DAG :)

- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).

- We say that we exit a SCC when we exit its last vertex.

- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- a graph is strongly connected, if we can reach each node from every other node
- we can partition a graph into strongly connected components
- the SCCs form a DAG :) 
- let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes)
- we say that we exit a SCC when we exit its last vertex
- on a DAG, exit times of DFS are in reverse topological order (sink first)
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :) (successor-first ordering).
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- a graph is strongly connected, if we can reach each node from every other node
- we can partition a graph into strongly connected components
- the SCCs form a DAG :)
- let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes)
- we say that we exit a SCC when we exit its last vertex
- on a DAG, exit times of DFS are in reverse topological order (sink first)
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG.:)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG ;)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let's consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

Algo 1: (Kosaraju, Sharir)
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

Algo 1: (Kosaraju, Sharir)
- Collect exit times.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

Algo 1: (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.
Strongly Connected Components (SCCs)

- A graph is strongly connected if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG ;)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

### Algo 1: (Kosaraju, Sharir)
- Collect exit times
- Transpose graph (reverse all edges)
- Repeatedly do DFS from unvisited vertex with highest exit time
- Everything reachable is one SCC
**Strongly Connected Components (SCCs)**

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1:** (Kosaraju, Sharir)

- Collect exit times.
-Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.

Why not lowest exit time and skip transpose?
Strongly Connected Components (SCCs)

- A graph is strongly connected, if we can reach each node from every other node.
- We can partition a graph into strongly connected components.
- The SCCs form a DAG :)
- Let’s consider a DFS traversal of the graph (repeated DFS from unreached nodes).
- We say that we exit a SCC when we exit its last vertex.
- On a DAG, exit times of DFS are in reverse topological order (sink first).

**Algo 1**: (Kosaraju, Sharir)
- Collect exit times.
- Transpose graph (reverse all edges).
- Repeatedly do DFS from unvisited vertex with highest exit time.
- Everything reachable is one SCC.

**Algo 2**: Use stack trick again to eat away components in reverse topological order during first traversal → Tarjan’s Algorithm (next slide).
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- We want to identify the point during DFS when we exit a SCC
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC

![Diagram of SCC algorithm](image)
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC

- back-edges point to an ancestor
- will be to node in same SCC only
**Tarjan’s SCC Algorithm**

- **Algo**: use stack trick to eat away components in reverse topological order

- we want to identify the point during DFS when we exit a SCC

- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)

- all edges out of subtree point to own or completed SCC

- back-edges point to an ancestor

- will be to node in same SCC only

- forward-edges point to a descendant

- either to node in same SCC or completed SCC
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC

- back-edges point to an ancestor
- will be to node in same SCC only
- forward-edges point to a descendant
- either to node in same SCC or completed SCC
- cross-edges point to a node that the DFS already finished
- either to node in same SCC or completed SCC
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC
- all edges pointing out of subtree point to lower discover time

- back-edges point to an ancestor
- will be to node in same SCC only
- forward-edges point to a descendant
- either to node in same SCC or completed SCC
- cross-edges point to a node that the DFS already finished
- either to node in same SCC or completed SCC
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC
- all edges pointing out of subtree point to lower discover time
- find lowest disc. time reachable via edge from subtree ignoring completed SCCs

- back-edges point to an ancestor
- will be to node in same SCC only
- forward-edges point to a descendant
- either to node in same SCC or completed SCC
- cross-edges point to a node that the DFS already finished
- either to node in same SCC or completed SCC
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order
- we want to identify the point during DFS when we exit a SCC
- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)
- all edges out of subtree point to own or completed SCC
- all edges pointing out of subtree point to lower discover time
- find lowest disc. time reachable via edge from subtree ignoring completed SCCs

- back-edges point to an ancestor
- will be to node in same SCC only
- forward-edges point to a descendant
- either to node in same SCC or completed SCC
- cross-edges point to a node that the DFS already finished
- either to node in same SCC or completed SCC

**how to ignore completed SCC?**
Tarjan’s SCC Algorithm

- **Algo**: use stack trick to eat away components in reverse topological order

- we want to identify the point during DFS when we exit a SCC

- it’s when we can’t reach any node outside our subtree (ignoring completed SCCs)

- all edges out of subtree point to own or completed SCC

- all edges pointing out of subtree point to lower discover time

- find lowest disc. time reachable via edge from subtree ignoring completed SCCs

```c
// returns lowest disc. time of nodes in SCC reachable via 1 edge from my subtree
int dfs(v) {
    int st_pos = size(st);
    st.push_back(v);
    in_stack[v] = true;
    int low = disc[v] = time++;
    for(int v2 : adj[v]) {
        if(disc[v2]==-1) // tree edge
            low = min(low, dfs(v2));
        if(in_stack[v2]) // not yet assigned
            low = min(low, disc[v2]);
    }
    if(low==disc[v]) { // exiting SCC
        SCC is in st[st_pos..] // TODO: remove SCC from stack
    } else {
        Hint: in_stack[v2] → v2 reaches v
    }
    return low;
}
```