Applications of DFS

November 12, 2015

Outline

Applications of DFS

Basic Algorithm

```
vector<int> visited(n,0);
vector< vector<int> > adj; // adjacency list

void dfs(int v) {
  visited[v] = 1;
  for (int i = 0; i < adj[v].size(); i++) {
    if (!visited[adj[v][i]]) {
      dfs(adj[v][i]);
    }
  }
}
```

I. Topological Sorting

Problem

Given a directed acyclic graph (DAG), order the vertices as $v_1, \ldots, v_n$ such that for every edge $\overrightarrow{v_i v_j}$, we have $i \leq j$.

I. Topological Sorting

Equivalent formulation:

Problem

Given a partially ordered set $(V, \preceq)$, order the elements as $v_1, \ldots, v_n$ such that for every pair $v_i \preceq v_j$, we have $i \leq j$.

I. Topological Sorting

Equivalent formulation:

Problem

Given a partially ordered set $(V, \preceq)$, order the elements as $v_1, \ldots, v_n$ such that for every pair $v_i \preceq v_j$, we have $i \leq j$.

- The resulting order on V is called a linear extension of $\preceq$.

I. Topological Sorting

Equivalent formulation:

Problem

Given a partially ordered set $(V, \preceq)$, order the elements as $v_1, \ldots, v_n$ such that for every pair $v_i \preceq v_j$, we have $i \leq j$.

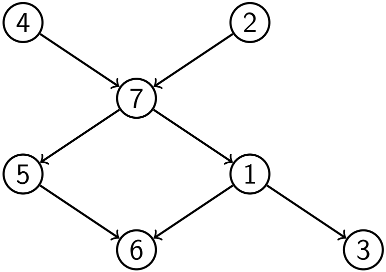
- The resulting order on V is called a linear extension of $\preceq$.
- Application: dependency resolution.

I. Topological Sorting

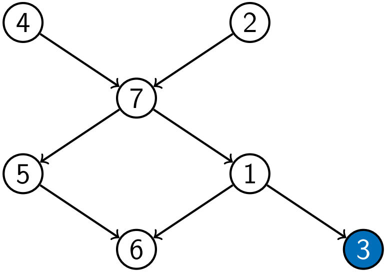
Algorithm (idea):

```
vector<int> result;

while (there is a  minimal element v) {
    result.push_back(v);
    remove v from the graph;
}
```

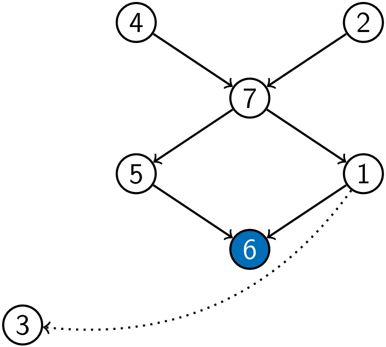
I. Topological Sorting



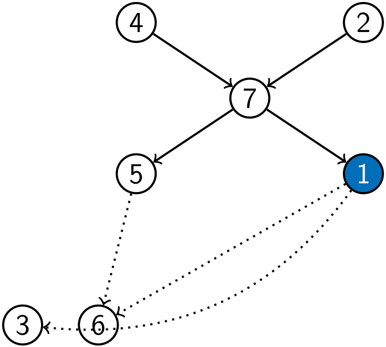
I. Topological Sorting



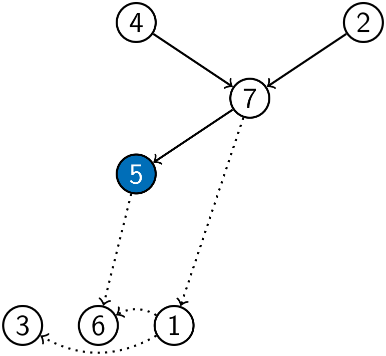
I. Topological Sorting



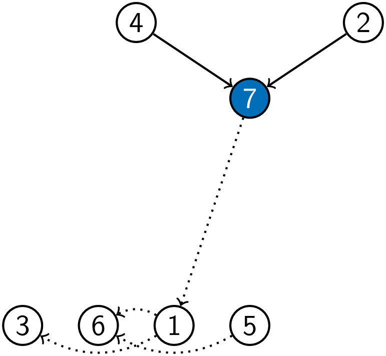
I. Topological Sorting



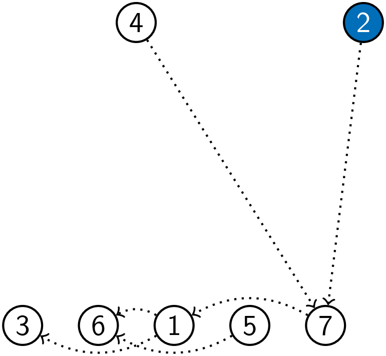
I. Topological Sorting



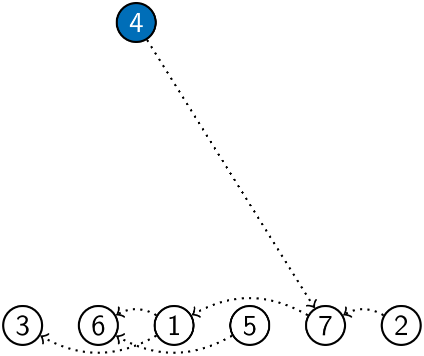
I. Topological Sorting



I. Topological Sorting

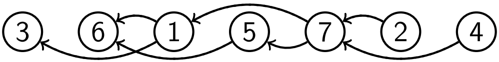


I. Topological Sorting



I. Topological Sorting

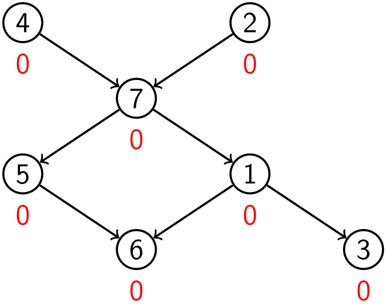
I. Topological Sorting



This idea can be implemented with multiple DFS's:

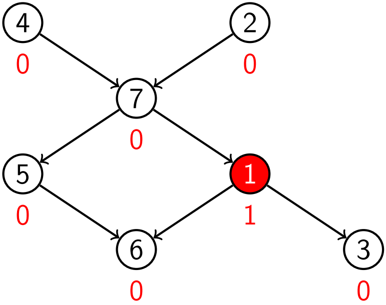
```
vector<int> result;
vector<int> visited(n,0);

for (int i = 0; i < n; i++) {
  if (!visited[i]) dfs(i);
}

void dfs(int v) {
    if (!visited[v]) {
        visited[v] = 1;
        for (int i = 0; i < adj[v].size(); i++) {
            if (!visited[adj[v][i]]) dfs(m);
        }
        result.push_back(v);
    }
}
```

I. Topological Sorting



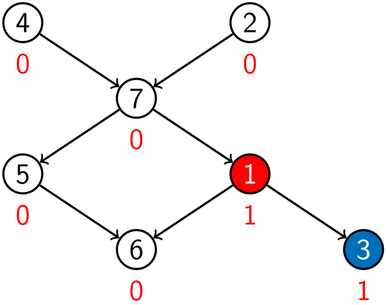
Result:

I. Topological Sorting



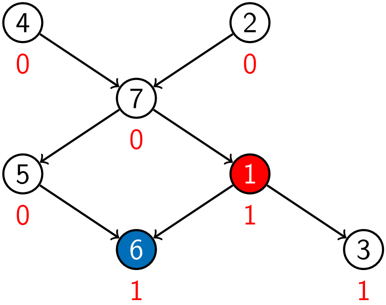
Result:

I. Topological Sorting



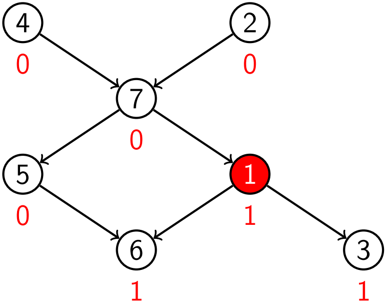
Result:

I. Topological Sorting



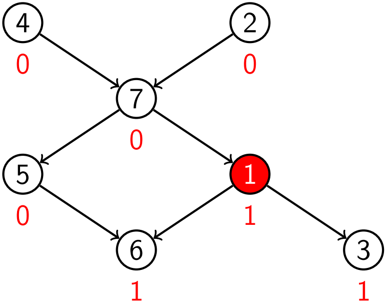
Result: 3

I. Topological Sorting

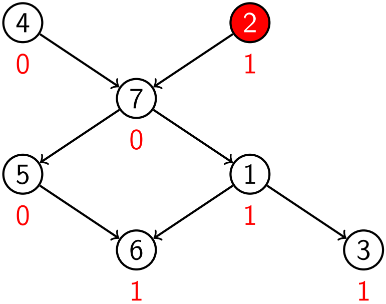


Result: 3 6

I. Topological Sorting

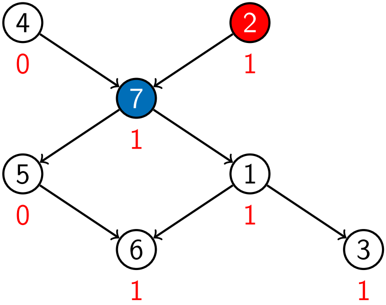


Result: 3 6 1



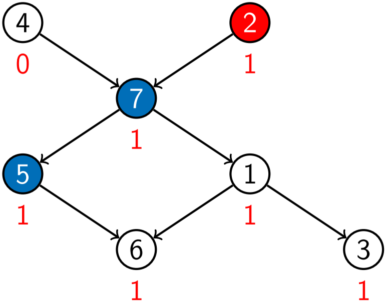
Result: 3 6 1

I. Topological Sorting



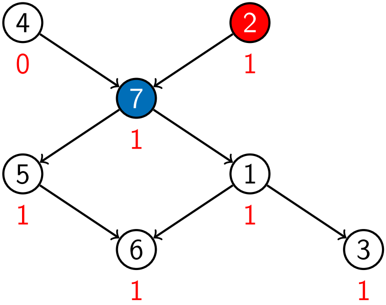
Result: 3 6 1

I. Topological Sorting



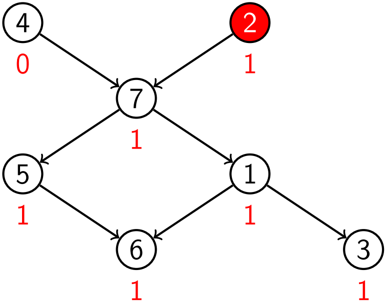
Result: 3 6 1

I. Topological Sorting



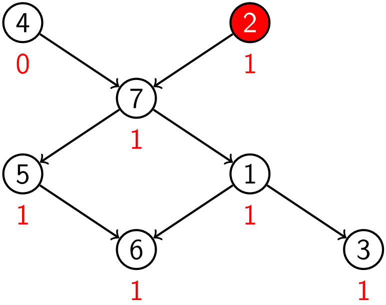
Result: 3 6 1 5

I. Topological Sorting



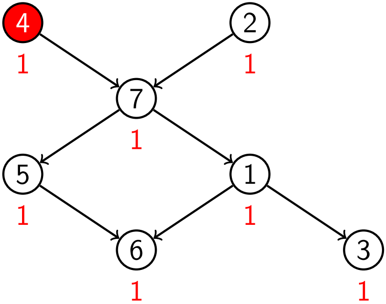
Result: 3 6 1 5 7

I. Topological Sorting



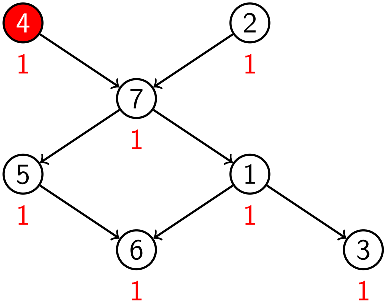
Result: 3 6 1 5 7 2

I. Topological Sorting



Result: 3 6 1 5 7 2

I. Topological Sorting



Result: 3 6 1 5 7 2 4

I. Topological Sorting

- Each vertex and edge is considered only once.
- So the running time of this algorithm is linear (more precisely: $O(|V| + |E|)$).

II. BCC/Articulation Points/Bridges

Definition

Let G be a simple graph. An articulation point is a vertex of G whose removal separates G into at least two components.

II. BCC/Articulation Points/Bridges

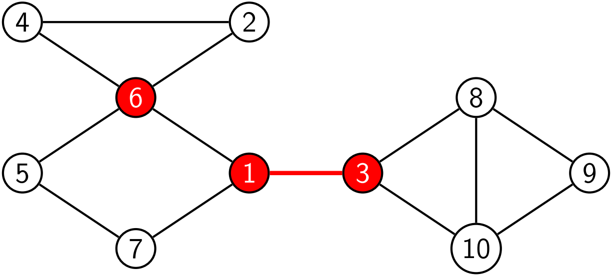
Definition

Let G be a simple graph. An articulation point is a vertex of G whose removal separates G into at least two components.

Definition

Let G be a simple graph. A bridge is an edge of G whose removal separates G into two components.

II. BCC/Articulation Points/Bridges

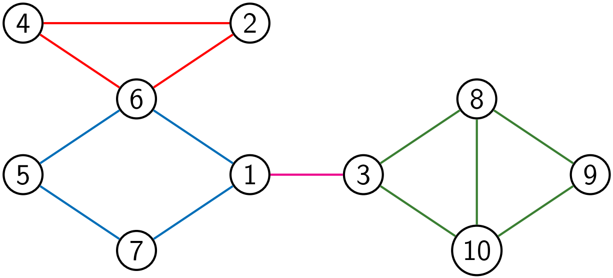


II. BCC/Articulation Points/Bridges

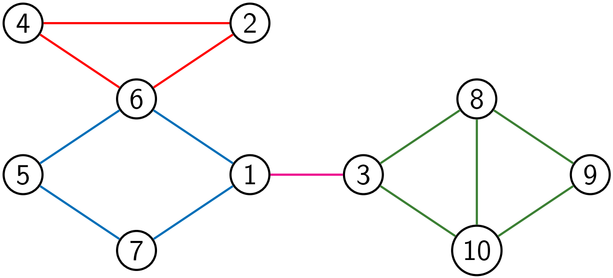
Definition

A graph G is biconnected if it has no articulation points.
A biconnected component of a graph is a maximally biconnected subgraph.

II. BCC/Articulation Points/Bridges

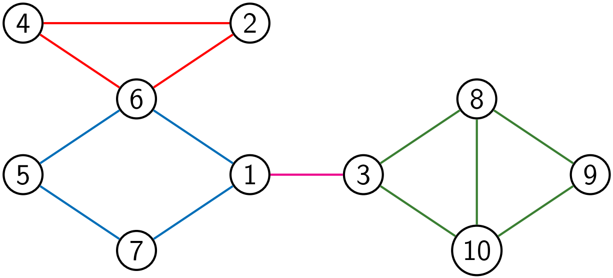


II. BCC/Articulation Points/Bridges



- Biconnected components intersect in articulation points.

II. BCC/Articulation Points/Bridges



- Biconnected components intersect in articulation points.
- Bridges are biconnected components of size two.

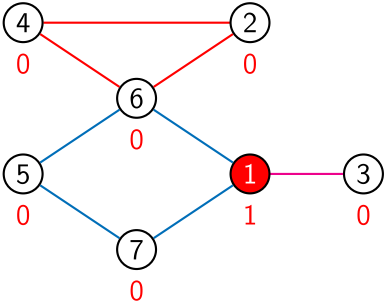
II. BCC/Articulation Points/Bridges

Using DFS, we can find biconnected components, articulation points, and bridges in linear time.

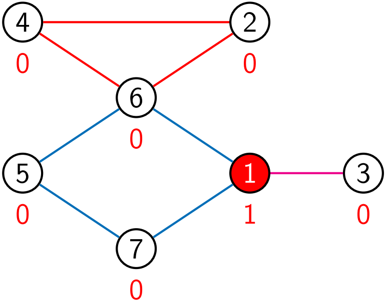
II. BCC/Articulation Points/Bridges

First, observe that running a DFS from a node creates a spanning tree (the *DFS tree*), as well as some back-edges.

Example:

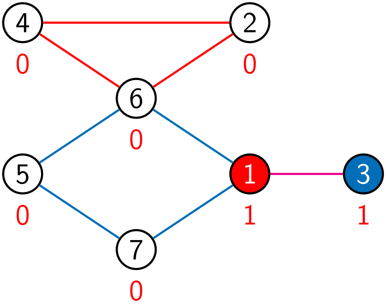


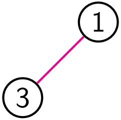
Example:



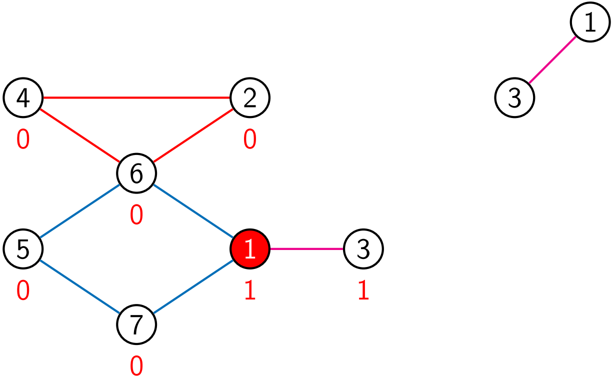
II. BCC/Articulation Points/Bridges

Example:



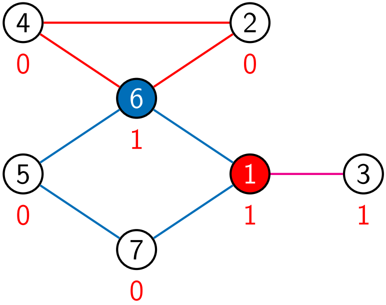


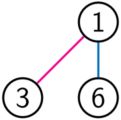
Example:



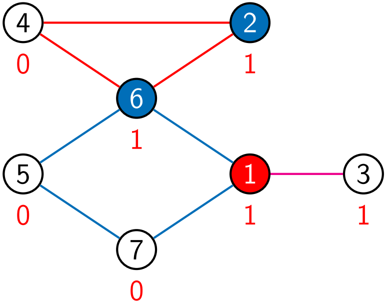
II. BCC/Articulation Points/Bridges

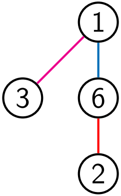
Example:





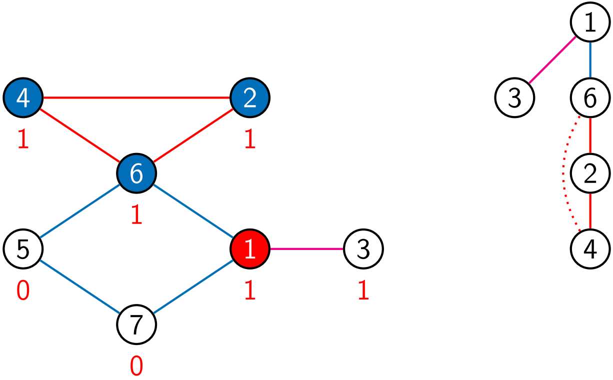
Example:





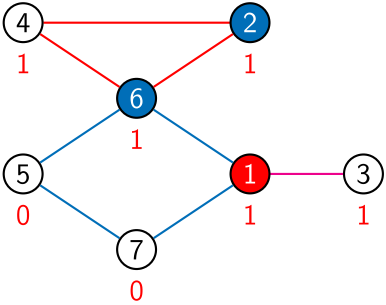
II. BCC/Articulation Points/Bridges

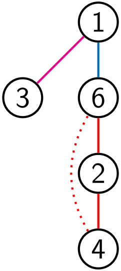
Example:



II. BCC/Articulation Points/Bridges

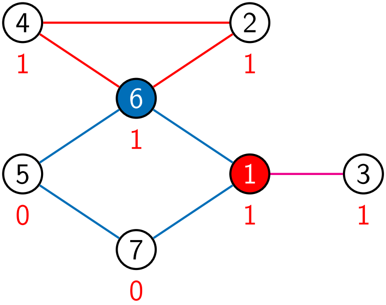
Example:

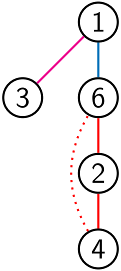




II. BCC/Articulation Points/Bridges

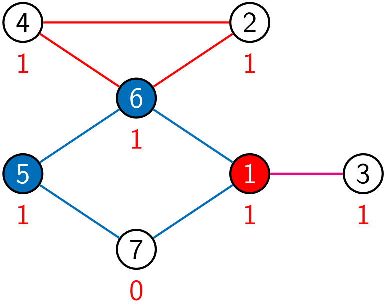
Example:

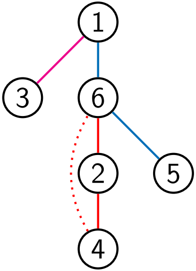




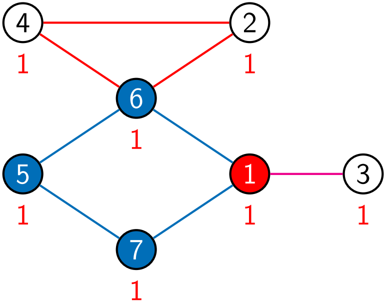
II. BCC/Articulation Points/Bridges

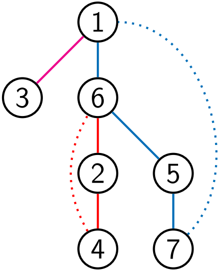
Example:





Example:



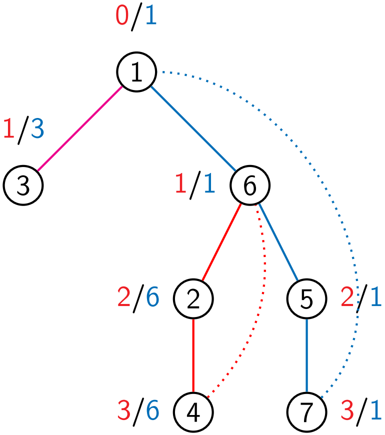


II. BCC/Articulation Points/Bridges

We will compute

- `depth[v]`: the depth of vertex v in the DFS tree
- `low[v]`: the lowest-depth neighbor of v or a descendant of v via back-edges in the DFS tree (or v itself if there is no lower-depth neighbor)

In our example, `depth` and `low` are as follows:



We can find articulation points:

- The root vertex is an articulation point iff it has more than one child.
- A non-root vertex `u` is an articulation point iff it has a child `v` with `depth[low[v]] >= depth[u]`.

II. BCC/Articulation Points/Bridges

We can find bridges:

- All bridges must be edges of the DFS tree.
- A vertex `u` forms a bridge with a child `v` iff `depth[low[v]] > depth[u]`.

II. BCC/Articulation Points/Bridges

We can find biconnected components:

- follow the DFS tree from the leaves upwards
- put vertices into the component of the leaf along the way
- stop at articulation points.

II. BCC/Articulation Points/Bridges

It is easy to compute depth and low during the DFS.

```
void dfs(int v, int d) {
  visited[v] = 1;
  depth[v] = d; // set depth
  low[v] = v;
  for (each child u of v) {
    if (!visited[u]) {
      dfs(u, d+1);
      if (depth[low[u]] <= depth[l]) low[v] = low[u];
    } else if (depth[u] < depth[v]-1) low[v] = u;
  }
}
dfs(0,0); // call like this
```

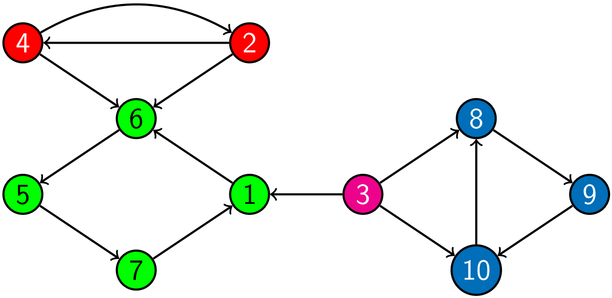
III. Strongly Connected Components

Definition

Let D be a directed graph. Write $u \sim v$ if vertex u can be reached from v and vice-versa. Then

- $\sim$ is an equivalence relation on $V(D)$, and
- the equivalence classes are called strongly connected components of D.

III. Strongly Connected Components (SCC)



- After contracting each SCC, we are left with a DAG!

III. Strongly Connected Components

How to compute strongly connected components?

- Do a DFS and compute finishig times.
- Compute transposed graph, by reverting all edges.
- Do DFS on transposed graph, prioritizing large finishing times.

III. Strongly Connected Components

How to compute strongly connected components?

- Do a DFS and compute finishig times.
- Compute transposed graph, by reverting all edges.
- Do DFS on transposed graph, prioritizing large finishing times.

The complexity is $\mathcal{O}(|V| \log |V| + |E|)$.

III. Strongly Connected Components

How to compute strongly connected components?

- Do a DFS and compute finishig times.
- Compute transposed graph, by reverting all edges.
- Do DFS on transposed graph, prioritizing large finishing times.

The complexity is $\mathcal{O}(|V| \log |V| + |E|)$.
To get $\mathcal{O}(|V| + |E|)$ you can use Tarjans algorithm which uses `low[v]`.

That's all.