Chapter 22 Solusion

https://github.com/frc123/CLRS

12/1/2021

22.1

22.1-1

out-degree: $\Theta(V+E)$ by simply counting the size of each adjacency list.

```
std::vector<int> OutDegree(const Graph& graph)
{
    size_t v_size, i;
    v_size = graph.adj.size();
    std::vector<int> degree(v_size);
    for (i = 0; i < v_size; ++i)
    {
        // assume graph.adj[i].size() takes O(n)
        // where n is size of  graph.adj[i]
        degree[i] = graph.adj[i].size();
    }
    return degree;
}
```

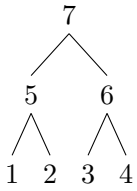
in-degree: $\Theta(V+E)$ by maintaining a counting table: each entry of the table is the counter for in-degree of the specific vertex.

```
std::vector<int> InDegree(const Graph& graph)
{
    size_t v_size, i;
    v_size = graph.adj.size();
    std::vector<int> degree(v_size);
    for (i = 0; i < v_size; ++i)
    {
        for (int v : graph.adj[i])
        {
```

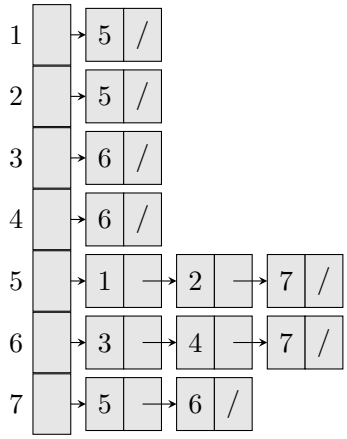
```
10                ++degree[v];
11            }
12        }
13        return degree;
14    }
```

22.1-2

Consider the following binary tree:



We have the following adjacency-list representation:



We have the following adjacency-matrix representation:

	1	2	3	4	5	6	7
1	0	0	0	0	1	0	0
2	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0
4	0	0	0	0	0	1	0
5	1	1	0	0	0	0	1
6	0	0	1	1	0	0	1
7	0	0	0	0	1	1	0

22.1-3

We can compute G^T from G for the adjacency-list representation in $\Theta(V + E)$ by the following algorithm:

```
1  AdjListGraph Transpose(const AdjListGraph& graph)
2  {
```

```
    size_t size, u;
    size = graph.adj.size();
    AdjListGraph target(size);
    for (u = 0; u < size; ++u)
    {
        for (int v : graph.adj[u])
        {
            target.adj[v].push_back(u);
        }
    }
    return target;
}
```

We can compute G^T from G for the adjacency-matrix representation in $\Theta(V^2)$ by the following algorithm:

```
AdjMatrixGraph Transpose(const AdjMatrixGraph& graph)
{
    size_t size, u, v;
    size = graph.adj.size();
    AdjMatrixGraph target(size);
    for (u = 0; u < size; ++u)
    {
        for (v = 0; v < size; ++v)
        {
            target.adj[v][u] = graph.adj[u][v];
        }
    }
    return target;
}
```

22.1-4

```
AdjListGraph Equivalent(const AdjListGraph& graph)
{
    size_t size, u;
    size = graph.adj.size();
    AdjListGraph target(size);
    std::vector<bool> edge_usage;
    for (u = 0; u < size; ++u)
    {
```

```
9           edge_usage = std::vector<bool>(size, false);
10          edge_usage[u] = true;
11          for (int v : graph.adj[u])
12          {
13              if (edge_usage[v] == false)
14              {
15                  target.adj[u].push_back(v);
16                  edge_usage[v] = true;
17              }
18          }
19      }
20      return target;
21  }
```

22.1-5

We can compute G^2 from G for the adjacency-list representation in $O(VE)$ by the following algorithm:

```
1   AdjListGraph Square(const AdjListGraph& graph)
2   {
3       size_t size, u;
4       size = graph.adj.size();
5       AdjListGraph result(size);
6       for (u = 0; u < size; ++u)
7       {
8           for (int v : graph.adj[u])
9           {
10              result.adj[u].push_back(v);
11              for (int w : graph.adj[v])
12              {
13                  result.adj[u].push_back(w);
14              }
15          }
16      }
17      return result;
18  }
```

We can compute G^2 from G for the adjacency-matrix representation in $\Theta(V^3)$ by the following algorithm (note that G^2 might be a not simple graph):

```
1   AdjMatrixGraph Square(const AdjMatrixGraph& graph)
```

```
{
    size_t size, u, v, w;
    size = graph.Rows();
    AdjMatrixGraph result(size, size);
    for (u = 0; u < size; ++u)
    {
        for (v = 0; v < size; ++v)
        {
            if (graph[u][v])
            {
                result[u][v] = true;
                for (w = 0; w < size; ++w)
                {
                    if (graph[v][w])
                    {
                        result[u][w] = true;
                    }
                }
            }
        }
    }
    return result;
}
```

We also can optimate computation of G^2 from G by using Strassen algorithm.

Lemma 1. Let $A = (a_{ij})$ be a $n \times n$ nonnegative matrix and $B = A^2 = (b_{ij})$. Then $b_{uw} > 0$ if and only if there exists some integer $v \in [1, n]$ such that $a_{uv} > 0$ and $a_{vw} > 0$.

Proof. Contrapositive: $b_{uw} = 0 \iff (\forall v \in \mathbb{Z}_{[1,n]}, a_{uv} = 0 \vee a_{vw} = 0)$

According to equation (4.8) on page 75, we have

$$b_{uw} = \sum_{v=1}^{n} a_{uv} \cdot a_{vw}.$$

Note A is nonnegative matrix. Clearly, $b_{uw} = 0$ if and only if $a_{uv} = 0$ or $a_{vw} = 0$ for all integer $v \in [1, n]$ □

Claim 2. Let $A = (a_{ij})$ be the adjacency-matrix repressentations of graph $G = (V, E)$. Let $B = A^2 = (b_{ij})$. Then $b_{uw} > 0$ if and only if there exists a path with exactly two edges between u and w.

Proof. To prove the claim, we just need to show that "there exists a path with exactly two edges between u and w" is equivalent to "there exists some integer $v \in [1, n]$ such that $a_{uv} > 0$ and $a_{vw} > 0$" so we can utilize lemma 1. Let $v \in V$. We have $(u, v) \in E$ if and only if $a_{uv} > 0$. Similarly,

$(v, w) \in E$ if and only if $a_{vw} > 0$. Also, $(u, v) \in E$ and $(v, w) \in E$ means there exists a path: $u \to v \to w$. □

Therefore, we have the following algorithm run in $\Theta(|V|^{\lg 7})$:

```
AdjMatrixGraph SquareByStrassen(const AdjMatrixGraph& graph)
{
    size_t size, u, v, w;
    size = graph.Rows();
    AdjMatrixGraph result = StrassenMultiplication(graph, graph);
    for (u = 0; u < size; ++u)
    {
        for (v = 0; v < size; ++v)
        {
            if (graph[u][v])
            {
                result[u][v] = 1;
            }
        }
    }
    return result;
}
```

22.1-6

Notice that we can check whether a vertex is a universal sink in $\Theta(|V|)$. However, it will take $O(|V|^2)$ to check all vertex precisely. So, we want to constraint to a unique possible vertex and check that unique possible vertex.

Claim 3. $v \in V$ is a universal sink if and only if $(\forall w \in V, a_{vw} = 0)$ and $(\forall u \in V \setminus \{v\}, a_{uv} = 1)$.

Then we have

$$\begin{cases} a_{uv} = 1 & \text{implies } u \text{ is not a universal sink,} \\ a_{uv} = 0 \wedge u \neq v & \text{implies } v \text{ is not a universal sink.} \end{cases}$$

Thus we can eliminate a candidate vertex either u or v in $\Theta(1)$ by access a_{uv} if $u \neq v$.

Therefore, we have the following algorithm run in $\Theta(|V|)$:

```
// graph must be a square matrix
// return vertex of universal sink
// return -1 if universal sink not exist
int UniversalSink(const Matrix& graph)
{
```

```
    size_t size, u, v;
    size = graph.size();
    // eliminate candidates
    u = 0;
    v = 1;
    while (v < size)
    {
        if (graph[u][v])
        {
            ++u;
            if (u == v)
            {
                ++v;
            }
        }
        else
        {
            ++v;
        }
    }
    // test the possible vertex u by claim 3
    for (v = 0; v < size; ++v)
    {
        if (graph[u][v])
            return -1;
    }
    for (v = 0; v < size; ++v)
    {
        if (graph[v][u] == false && u != v)
            return -1;
    }
    return u;
}
```

The following algorithm runs in $\Theta(|V|)$ also:

```
int UniversalSinkAnother(const Matrix& graph)
{
    size_t size, u, v;
    size = graph.size();
    u = 0;
```

```
6       v = 0;
7       while (u < size && v < size)
8       {
9           if (graph[u][v])
10          {
11              ++u;
12          }
13          else
14          {
15              ++v;
16          }
17      }
18      if (u >= size)
19          return -1;
20      for (v = 0; v < size; ++v)
21      {
22          if (graph[u][v])
23              return -1;
24      }
25      for (v = 0; v < size; ++v)
26      {
27          if (graph[v][u] == false && u != v)
28              return -1;
29      }
30      return u;
31  }
```

22.1-7

Let matrix $C = B^T = (c_{ij})$. This says C is a $|E| \times |V|$ matrix, and $c_{ij} = b_{ji}$. Let $D = BB^T = (d_{ij})$. Hence we have

$$d_{ij} = \sum_{k \in E} b_{ik} c_{kj} = \sum_{k \in E} b_{ik} b_{jk}$$

In conclusion, the meaning of d_{ij} depends on whether $i = j$.

Case 1 $i = j$

$b_{ik} b_{jk} = b_{ik} = 1 = 1 \cdot 1 = -1 \cdot -1$ implies edge k enters or leaves vertex i.

$b_{ik} b_{jk} = b_{ik} = 0$ implies edge k does not connect to vertex i.

$b_{ik} b_{jk} = b_{ik} = -1$ is impossible since $b_{ik} = b_{jk}$.

Hence d_{ij} means the total degree (in-degree + out-degree) of vertex i.

Case 2 $i \neq j$

$b_{ik}b_{jk} = 1 = 1 \cdot 1 = -1 \cdot -1$ is impossible since edge k cannot enter i and j simultaneously, and edge k cannot leave i and j simultaneously.

$b_{ik}b_{jk} = 0$ implies edge k does not connect to vertex i and j.

$b_{ik}b_{jk} = -1$ implies edge k leaves vertex i and enters j, or edge k leaves vertex j and enters i.

Hence $-d_{ij}$ means the number of edges connect to vertex i and j simultaneously.

22.1-8

Expected time to determine whether an edge is in the graph: $\Theta(1)$.

Disadvantage to use hash table: 1. we are not able to handle graphs that are not simple; 2. the worst case take $\Theta(|V|)$ time.

Suggest: utilize red-black trees containing keys v much that $(u, v) \in E$; add a counter (counter for unweighted graph; list for weighted graph) to the attributes of each node in the red-black tree to handle graphs that are not simple.

Disadvantage compared to the hash table: expect time of red-black tree is $\Theta(\lg n)$ where n is the size of elements in the red-black tree.

22.2

22.2-1

vertex 1: $d = \infty$, $\pi = NIL$.
vertex 2: $d = 3$, $\pi = 4$.
vertex 3: $d = 0$, $\pi = NIL$.
vertex 4: $d = 2$, $\pi = 5$.
vertex 5: $d = 1$, $\pi = 3$.
vertex 6: $d = 1$, $\pi = 3$.

22.2-2

vertex r: $d = 4$, $\pi = s$.
vertex s: $d = 3$, $\pi = w$.
vertex t: $d = 1$, $\pi = u$.
vertex u: $d = 0$, $\pi = NIL$.
vertex v: $d = 5$, $\pi = r$.
vertex w: $d = 2$, $\pi = x$.
vertex x: $d = 1$, $\pi = u$.
vertex y: $d = 1$, $\pi = u$.

22.2-3

We can replace all "GRAY"s with "BLACK"s. Notice line 13 is the only place to check color, and it only checks whether or not the vertex is "WHITE". Hence, as long as we can make sure all non-white vertex are in non-"WHITE" color, our algorithm works.

```
1  std::list<int> BFS(Graph& graph, int src_idx)
2  {
3      int u_idx, v_idx;
4      std::list<int> discover_result;
5      std::queue<int> q;
6      for (Vertex& u : graph.vertices)
7      {
8          u.color = Color::kWhite;
9          u.distance = INT_MAX;
10         u.prev = -1;
11     }
12     graph.vertices[src_idx].color = Color::kNonWhite;
13     graph.vertices[src_idx].distance = 0;
14     graph.vertices[src_idx].prev = -1;
15     q.push(src_idx);
16     while (q.empty() == false)
17     {
18         u_idx = q.front();
19         q.pop();
20         discover_result.push_back(u_idx);
21         Vertex& u = graph.vertices[u_idx];
22         for (int v_idx : graph.adj[u_idx])
23         {
24             Vertex& v = graph.vertices[v_idx];
25             if (v.color == Color::kWhite)
26             {
27                 v.color = Color::kNonWhite;
28                 v.distance = u.distance + 1;
29                 v.prev = u_idx;
30                 q.push(v_idx);
31             }
32         }
33     }
34     return discover_result;
35 }
```

22.2-4

$O(V^2)$ since we need to iterate all entries in the matrix.

```
std::list<int> BFS(Graph& graph, int src_idx)
{
    int u_idx, v_idx;
    std::list<int> discover_result;
    std::queue<int> q;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
        u.distance = INT_MAX;
        u.prev = -1;
    }
    graph.vertices[src_idx].color = Color::kGray;
    graph.vertices[src_idx].distance = 0;
    graph.vertices[src_idx].prev = -1;
    q.push(src_idx);
    while (q.empty() == false)
    {
        u_idx = q.front();
        q.pop();
        discover_result.push_back(u_idx);
        Vertex& u = graph.vertices[u_idx];
        for (v_idx = 0; v_idx < graph.size; ++v_idx)
        {
            if (graph.adj[u_idx][v_idx])
            {
                Vertex& v = graph.vertices[v_idx];
                if (v.color == Color::kWhite)
                {
                    v.color = Color::kGray;
                    v.distance = u.distance + 1;
                    v.prev = u_idx;
                    q.push(v_idx);
                }
            }
        }
        u.color = Color::kBlack;
    }
```

```
38      return discover_result;
39  }
```

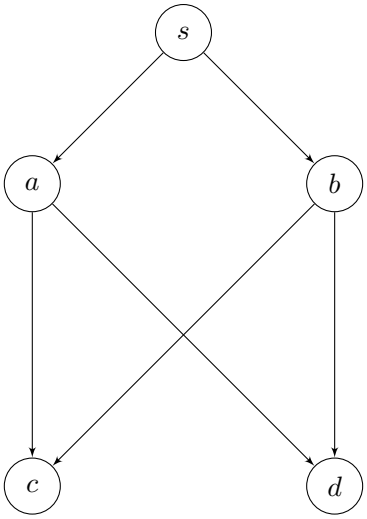
22.2-5

For all reachable vertices $v \in V$, we have $v.d = \delta(s, v)$ by Theorem 22.5. For unreachable vertices $v \in V$, we have $v.d = \infty = \delta(s, v)$. Since $\delta(s, v)$ is independent of the order in which the vertices appear in each adjacency list, so does $v.d$ for all $v \in V$.

If t is in front of x in the adjacency list of w, then t will be enqueued before x enqueue, and $u.\pi = t$, so $(t, u) \in E_\pi$. If t is in after x in the adjacency list of w, then t will be enqueued after x enqueue, and $u.\pi = x$, so $(x, u) \in E_\pi$.

22.2-6

Consider the following directed graph $G = (V, E)$:



Consider $E_\pi = \{(s, a), (s, b), (a, c), (b, d)\}$. This E_π cannot be produced by running BFS on G. The only two possible predecessor subgraph is

$$(V, \{(s, a), (s, b), (a, c), (a, d)\})$$

and

$$(V, \{(s, b), (s, a), (b, c), (b, d)\}).$$

22.2-7

Let $G = (V, E)$ be an undirected graph.

Let each vertex $v \in V$ represent a professional wrestler, and let each edge $e \in E$ represent a rivalry.

```cpp
bool DesignateWrestlerType(Graph& graph)
{
    int u_idx, v_idx;
    std::queue<int> q;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
    }
    graph.vertices[0].color = Color::kNonWhite;
    graph.vertices[0].type = WrestlerType::kBabyfaces;
    q.push(0);
    while (q.empty() == false)
    {
        u_idx = q.front();
        q.pop();
        Vertex& u = graph.vertices[u_idx];
        for (int v_idx : graph.adj[u_idx])
        {
            Vertex& v = graph.vertices[v_idx];
            if (v.color == Color::kWhite)
            {
                v.color = Color::kNonWhite;
                graph.vertices[v_idx].type = (u.type == WrestlerType::kBabyfaces) ?
                    WrestlerType::kHeels : WrestlerType::kBabyfaces;
                q.push(v_idx);
            }
            else
            {
                if (u.type == v.type)
                    return false;
            }
        }
    }
    return true;
}
```

22.2-8

According to Theorem B.2 (Properties of free trees), We have T is connected, acyclic, and $|E| = |V| - 1$. This says we can do BFS in $O(|V| + |E|) = O(|V| + |V| - 1) = O(|V|)$ time. If we

can do constant times BFS, we will be able to find diameter in $O(|V|)$ time.

Let arbitrary node $s \in V$ be the sourse node of BFS. Since T is connected, all nodes in V are reachable. After performing BFS on T from the souece node s, we have $v.d = \delta(s, v)$ for each $v \in V$. Let s be the root node of the tree T. Then $v.d$ is the depth of node v is the tree since any two vertices in a tree are connected by a unique simple path (Theorem B.2 (2)) and the length of the path from s to v is $\delta(s, v)$, which is depth of node v. We can recursive traverse the tree to get the height of each node based on depth, and we can get the longest path containes the root of the subtree by height of children of the root.

Since BFS takes $O(|V|)$ and traverses tree takes $O(|V|)$, our algorithm run in $O(|V|)$.

```
int DiameterAux(Graph& graph, int subtree_root_idx)
{
    int first_largest_h, second_largest_h, longest_path_length;
    first_largest_h = -1;
    second_largest_h = -1;
    longest_path_length = INT_MIN;
    Vertex& subtree_root = graph.vertices[subtree_root_idx];
    for (int child_idx : graph.adj[subtree_root_idx])
    {
        if (child_idx != subtree_root.prev)
        {
            Vertex& child = graph.vertices[child_idx];
            longest_path_length = std::max(longest_path_length,
                DiameterAux(graph, child_idx));
            if (child.height > first_largest_h)
            {
                second_largest_h = first_largest_h;
                first_largest_h = child.height;
            }
            else if (child.height > second_largest_h)
            {
                second_largest_h = child.height;
            }
        }
    }
    subtree_root.height = first_largest_h + 1;
    longest_path_length = std::max(longest_path_length,
                first_largest_h + second_largest_h + 2);
    return longest_path_length;
}
```

```
31
32  int Diameter(Graph& graph)
33  {
34      int u_idx, v_idx;
35      std::queue<int> q;
36      // let vertex 0 be the root node
37      graph.vertices[0].depth = 0;
38      graph.vertices[0].prev = -1;
39      q.push(0);
40      while (q.empty() == false)
41      {
42          u_idx = q.front();
43          q.pop();
44          Vertex& u = graph.vertices[u_idx];
45          for (int v_idx : graph.adj[u_idx])
46          {
47              if (v_idx != u.prev)
48              {
49                  Vertex& v = graph.vertices[v_idx];
50                  v.depth = u.depth + 1;
51                  v.prev = u_idx;
52                  q.push(v_idx);
53              }
54          }
55      }
56      return DiameterAux(graph, 0);
57  }
```

22.2-9

```
1   std::list< std::pair<int, int> > TraverseEdge(Graph& graph, int src_idx)
2   {
3       int u_idx, v_idx;
4       std::list< std::pair<int, int> > traverse_result;
5       std::list< std::pair<int, int> >::iterator result_it;
6       std::queue<int> q;
7       for (Vertex& u : graph.vertices)
8       {
9           u.color = Color::kWhite;
10      }
```

```
    graph.vertices[src_idx].color = Color::kGray;
    graph.vertices[src_idx].result_pos = traverse_result.end();
    q.push(src_idx);
    while (q.empty() == false)
    {
        u_idx = q.front();
        q.pop();
        Vertex& u = graph.vertices[u_idx];
        result_it = u.result_pos;
        for (int v_idx : graph.adj[u_idx])
        {
            Vertex& v = graph.vertices[v_idx];
            if (v.color == Color::kWhite)
            {
                v.color = Color::kGray;
                traverse_result.emplace(result_it, u_idx, v_idx);
                v.result_pos =
                    traverse_result.emplace(result_it, v_idx, u_idx);
                q.push(v_idx);
            }
            else if (v.color == Color::kGray)
            {
                traverse_result.emplace(result_it, u_idx, v_idx);
                traverse_result.emplace(result_it, v_idx, u_idx);
            }
        }
        u.color = Color::kBlack;
    }
    return traverse_result;
}
```

Let each spot of the maze be a vertex in V. If there is no wall between two spots, we add an edge to connect two vertices. Then we traverse each edge in E by using the above algorithm to find the way out of the maze.

22.3

22.3-1

Directed:

i \ j	WHITE	GRAY	BLACK
W	tree/back/forward/cross	back/cross	cross
G	tree/forward	tree/back/forward	tree/forward/cross
B	N/A	back	tree/back/forward/cross

Undirected:

i \ j	WHITE	GRAY	BLACK
W	tree/back	tree/back	N/A
G	tree/back	tree/back	tree/back
B	N/A	tree/back	tree/back

22.3-2

Discovery and finish time:

q 1/16

s 2/7

v 3/6

w 4/5

t 8/15

x 9/12

z 10/11

y 13/14

r 17/20

u 18/19

Classification of each edge:

(q,s) tree

(s,v) tree

(v,w) tree

(w,s) back

(q,t) tree

(t,x) tree

(x,z) tree

(z,x) back

(t,y) tree

(y,q) back

(q,w) forward

(r,u) tree

(u,y) cross

(r,y) cross

22.3-3

(u (v (y (x x) y) v) u) (w (z z) w)

22.3-4

The algorithm only check if the color is white or non white, so we can use only two colors.

```
1  void DFSVisit(Graph& graph, int u_idx, int& time)
2  {
3      Vertex& u = graph.vertices[u_idx];
4      ++time;
5      u.discovery_time = time;
6      u.color = Color::kNonWhite;
7      for (int v_idx : graph.adj[u_idx])
8      {
9          Vertex& v = graph.vertices[v_idx];
10         if (v.color == Color::kWhite)
11         {
12             v.prev = u_idx;
13             DFSVisit(graph, v_idx, time);
14         }
15     }
16     ++time;
17     u.finishing_time = time;
18 }
19
20 void DFS(Graph& graph)
21 {
```

```
22      int time, u_idx;
23      for (Vertex& u : graph.vertices)
24      {
25          u.color = Color::kWhite;
26      }
27      time = 0;
28      for (u_idx = 0; u_idx < graph.size; ++u_idx)
29      {
30          Vertex& u = graph.vertices[u_idx];
31          if (u.color == Color::kWhite)
32          {
33              u.prev = -1;
34              DFSVisit(graph, u_idx, time);
35          }
36      }
37  }
```

22.3-5

(a)

Proof. Edge (u, v) is a tree edge or forward edge if and only if v is a proper descendant of u. By Corollary 22.8 (Nesting of descendants' intervals), we have $u.d < v.d < v.f < u.f$ if and only if v is a proper descendant of u. □

(b)

Proof. Edge (u, v) is a back edge if and only if u is a descendant of v (include self-loop). By Theorem 22.7 (Parenthesis theorem), we have $v.d \leq< u.d < u.f \leq v.f$ if and only if u is a descendant of v immediately. □

(c)

Proof. Edge (u, v) is a cross edge if and only if u is neither descendant of v nor ancestor of v. By Theorem 22.7 (Parenthesis theorem), we have $v.d < v.f < u.d < u.f$ or $u.d < u.f < v.d < v.f$ if and only if u is neither descendant of v nor ancestor of v. Thus, we have $v.d < v.f < u.d < u.f \implies (u, v)$ is a cross edge.

We claim (u, v) is a cross edge $\implies v.d < v.f < u.d < u.f$. Assume $u.d < u.f < v.d < v.f$, for the purpose of contradiction. This says u is black and v is white at some point during the DFS. However, there is a edge from u to v, which is a contradiction. □

22.3-6

Proof. By definition of tree edge, edge (u, v) is classified as a tree edge if v is first discovered by exploring edge (u, v), which is equivalent to (u, v) is encountered first. By Theorem 22.10, every edge in an undirected graph is either a tree edge or a back edge. Thus, if (u, v) is not classified as a tree edge, (u, v) is classified as a back edge, which is equivalent to (v, u) is encountered first. □

22.3-7

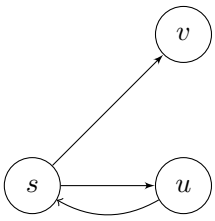
```
struct StackElement
{
    int vertex_idx;
    AdjList::iterator adj_it_curr;
    AdjList::iterator adj_it_end;
};

void DFS(Graph& graph)
{
    int time, u_idx, v_idx;
    std::stack< StackElement > path;
    AdjList vertices_list;
    for (u_idx = 0; u_idx < graph.size; ++u_idx)
    {
        Vertex& u = graph.vertices[u_idx];
        u.color = Color::kWhite;
        vertices_list.push_back(u_idx);
    }
    time = 0;
    path.push({ -1, vertices_list.begin(), vertices_list.end() });
    while (true)
    {
        u_idx = path.top().vertex_idx;
        AdjList::iterator& v_it = path.top().adj_it_curr;
        if (v_it == path.top().adj_it_end)
        {
            path.pop();
            if (path.empty())
                break;
            Vertex& u = graph.vertices[u_idx];
            u.color = Color::kBlack;
            ++time;
```

```
33              u.finishing_time = time;
34          }
35          else
36          {
37              v_idx = *v_it;
38              Vertex& v = graph.vertices[v_idx];
39              if (v.color == Color::kWhite)
40              {
41                  ++time;
42                  v.discovery_time = time;
43                  v.color = Color::kGray;
44                  v.prev = u_idx;
45                  path.push({ v_idx, graph.adj[v_idx].begin(), graph.adj[v_idx].end() });
46              }
47              ++v_it;
48          }
49      }
50  }
```

22.3-8

Consider the following directed graph:



Let s be the source vertex, and let u be in front of v in the adjacency list of s. Thus, we have the following result of DFS:

vetex	d	f
s	0	5
u	1	2
v	3	4

There exist a path from u to v: $u \to s \to v$.

22.3-9

Consider the same case as the solution of 22.3-8.

22.3-10

Directed graph:

```
struct Edge
{
    int vertex_from;
    int vertex_to;
    enum Type { kTree, kBack, kForward, kCross } type;

    Edge(int vertex_from, int vertex_to, Type type)
        : vertex_from(vertex_from), vertex_to(vertex_to), type(type) {}
};

void DFSVisit(Graph& graph, int u_idx, int& time, std::list<Edge>& result)
{
    Vertex& u = graph.vertices[u_idx];
    ++time;
    u.discovery_time = time;
    u.color = Color::kGray;
    for (int v_idx : graph.adj[u_idx])
    {
        Vertex& v = graph.vertices[v_idx];
        if (v.color == Color::kWhite)
        {
            result.emplace_back(u_idx, v_idx, Edge::kTree);
            v.prev = u_idx;
            DFSVisit(graph, v_idx, time, result);
        }
        else if (v.color == Color::kGray)
        {
            result.emplace_back(u_idx, v_idx, Edge::kBack);
        }
        else if (u.discovery_time < v.discovery_time)
        {
            result.emplace_back(u_idx, v_idx, Edge::kForward);
        }
        else
        {
            result.emplace_back(u_idx, v_idx, Edge::kCross);
        }
    }
    u.color = Color::kBlack;
    ++time;
```

```
    u.finishing_time = time;
}

std::list<Edge> DFS(Graph& graph)
{
    int time, u_idx;
    std::list<Edge> result;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
        u.prev = -1;
    }
    time = 0;
    for (u_idx = 0; u_idx < graph.size; ++u_idx)
    {
        if (graph.vertices[u_idx].color == Color::kWhite)
        {
            DFSVisit(graph, u_idx, time, result);
        }
    }
    return result;
}
```

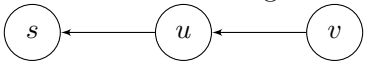
Undirected graph:

```
void DFSVisitUndirected(Graph& graph, int u_idx, int& time, std::list<Edge>& result)
{
    Vertex& u = graph.vertices[u_idx];
    ++time;
    u.discovery_time = time;
    u.color = Color::kGray;
    for (int v_idx : graph.adj[u_idx])
    {
        Vertex& v = graph.vertices[v_idx];
        if (u.prev != v_idx)
        {
            if (v.color == Color::kWhite)
            {
                result.emplace_back(u_idx, v_idx, Edge::kTree);
                v.prev = u_idx;
                DFSVisitUndirected(graph, v_idx, time, result);
```

```
17              }
18              else if (v.color == Color::kGray)
19              {
20                  result.emplace_back(u_idx, v_idx, Edge::kBack);
21              }
22          }
23      }
24      u.color = Color::kBlack;
25      ++time;
26      u.finishing_time = time;
27  }
28
29  std::list<Edge> DFSUndirected(Graph& graph)
30  {
31      int time, u_idx;
32      std::list<Edge> result;
33      for (Vertex& u : graph.vertices)
34      {
35          u.color = Color::kWhite;
36          u.prev = -1;
37      }
38      time = 0;
39      for (u_idx = 0; u_idx < graph.size; ++u_idx)
40      {
41          if (graph.vertices[u_idx].color == Color::kWhite)
42          {
43              DFSVisitUndirected(graph, u_idx, time, result);
44          }
45      }
46      return result;
47  }
```

22.3-11

Consider the following directed graph:



Let s be the first source vertex, let u be the second, and let v be the third. Thus, we have the following result of DFS:

vetex	d	f
s	1	2
u	3	4
v	5	6

There exist a depth-first tree containing only u.

22.3-12

```
1  void DFSVisitUndirected(Graph& graph, int u_idx, int& time, int cc)
2  {
3      Vertex& u = graph.vertices[u_idx];
4      ++time;
5      u.discovery_time = time;
6      u.color = Color::kGray;
7      u.connecting_component = cc;
8      for (int v_idx : graph.adj[u_idx])
9      {
10         Vertex& v = graph.vertices[v_idx];
11         if (u.prev != v_idx)
12         {
13             if (v.color == Color::kWhite)
14             {
15                 v.prev = u_idx;
16                 DFSVisitUndirected(graph, v_idx, time, cc);
17             }
18         }
19     }
20     u.color = Color::kBlack;
21     ++time;
22     u.finishing_time = time;
23 }
24
25 void DFSUndirected(Graph& graph)
26 {
27     int time, u_idx, cc;
28     for (Vertex& u : graph.vertices)
29     {
30         u.color = Color::kWhite;
31         u.prev = -1;
32     }
```

```
33      time = 0;
34      cc = 0;
35      for (u_idx = 0; u_idx < graph.size; ++u_idx)
36      {
37          if (graph.vertices[u_idx].color == Color::kWhite)
38          {
39              ++cc;
40              DFSVisitUndirected(graph, u_idx, time, cc);
41          }
42      }
43  }
```

22.3-13

We can run DFS multiple times with source on each vertex on the graph. A directed graph is not signly connected if there exist some forward edges or cross edges that connect vertices in the same depth-first tree (i.e. in the same component). Hence, we have the following algorithm runs in $O(|V|(|V|+|E|))$:

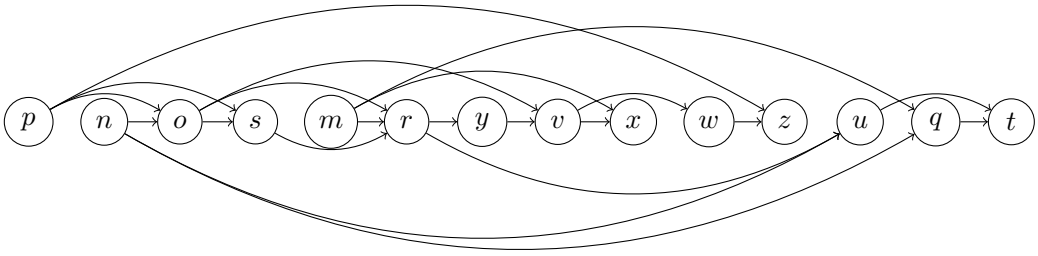
```
1   bool DFSVisit(Graph& graph, int u_idx, int& time)
2   {
3       Vertex& u = graph.vertices[u_idx];
4       ++time;
5       u.discovery_time = time;
6       u.color = Color::kGray;
7       for (int v_idx : graph.adj[u_idx])
8       {
9           Vertex& v = graph.vertices[v_idx];
10          if (v.color == Color::kWhite)
11          {
12              v.prev = u_idx;
13              DFSVisit(graph, v_idx, time);
14          }
15          else if (v.color == Color::kBlack)
16          {
17              return false;
18          }
19      }
20      u.color = Color::kBlack;
21      ++time;
22      u.finishing_time = time;
```

```
23      return true;
24  }
25
26  bool DFSWithSrc(Graph& graph, int src_vertex)
27  {
28      int time, u_idx;
29      for (Vertex& u : graph.vertices)
30      {
31          u.color = Color::kWhite;
32          u.prev = -1;
33      }
34      time = 0;
35      return DFSVisit(graph, src_vertex, time);
36  }
37
38  bool CheckSinglyConnected(Graph& graph)
39  {
40      int u_idx;
41      for (u_idx = 0; u_idx < graph.size; ++u_idx)
42      {
43          if (DFSWithSrc(graph, u_idx) == false)
44              return false;
45      }
46      return true;
47  }
```

22.4

22.4-1

vetex	d	f
m	1	20
n	21	26
o	22	25
p	27	28
q	2	5
r	6	19
s	23	24
t	3	4
u	7	8
v	10	17
w	11	14
x	15	16
y	9	18
z	12	13



22.4-2

We can perform topological sort on G and utilize dynamic programming to count simple path in linear time.

Denote $c[u]$ as the number of simple paths from u to t in G. We have the following recursive solution:

$$c[u] = \begin{cases} 1 & \text{if } u = t, \\ 0 & \text{if } u.f < t.f, \\ \sum\limits_{(u,v)\in E} c[v] & \text{if } u.f > t.f. \end{cases}$$

The following bottom-up algorithm runs in $O(V + E)$:

```
1  // graph must be DAG
```

```cpp
// s and t must be in the graph
int CountSimplePaths(Graph& graph, int s, int t)
{
    int u;
    std::vector<int> dp_count(graph.size, 0);
    std::list<int> topo_sort = TopologicalSort(graph);
    // find t
    std::list<int>::reverse_iterator rit = topo_sort.rbegin();
    while (rit != topo_sort.rend())
    {
        if (*rit == s)
        {
            return 0;
        }
        else if (*rit == t)
        {
            ++rit;
            break;
        }
        ++rit;
    }
    // perform dp
    dp_count[t] = 1;
    while (rit != topo_sort.rend())
    {
        u = *rit;
        for (int v : graph.adj[u])
        {
            dp_count[u] += dp_count[v];
        }
        if (u == s)
            break;
        ++rit;
    }
    return dp_count[s];
}
```

22.4-3

A forest contains multiple free trees (components). By Theorem B.2, $|E| = |V| - 1$ in a free tree. Hence, we can perform DFS on the graph. For each component, if the number of edges is more than the number of vertices, we can determine the graph contains a cycle. Hence, we can bound the running time in $O(|V| + |V|) = O(|V|)$.

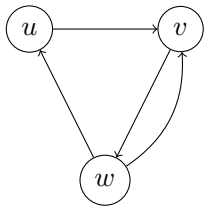
```
bool DFSVisit(Graph& graph, int u_idx, int& time)
{
    Vertex& u = graph.vertices[u_idx];
    ++time;
    u.discovery_time = time;
    u.color = Color::kGray;
    for (int v_idx : graph.adj[u_idx])
    {
        Vertex& v = graph.vertices[v_idx];
        if (u.prev != v_idx)
        {
            if (v.color == Color::kWhite)
            {
                v.prev = u_idx;
                DFSVisit(graph, v_idx, time);
            }
            else if (v.color == Color::kGray)
            {
                return false;
            }
        }
    }
    u.color = Color::kBlack;
    ++time;
    u.finishing_time = time;
    return true;
}

bool IsAcyclic(Graph& graph)
{
    int time, u_idx;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
```

```
35          u.prev = -1;
36      }
37      time = 0;
38      for (u_idx = 0; u_idx < graph.size; ++u_idx)
39      {
40          if (graph.vertices[u_idx].color == Color::kWhite)
41          {
42              if (DFSVisit(graph, u_idx, time) == false)
43              {
44                  return false;
45              }
46          }
47      }
48      return true;
49  }
```

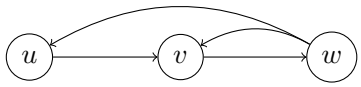
22.4-4

The claim is incorrect. Observed "bad" edges produced by topological sort are exactly back edges classified durning DFS. However, DFS might not minimized back edges. Consider the following graph:



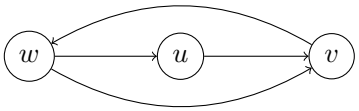
Let u be our source edge of DFS. Then we have the following DFS result and topological sort result which only has two "bad" edges: (w, u) and (w, v).

vetex	d	f
u	1	6
v	2	5
w	3	4



However, if we let w be out source edge of DFS and let u be in front of v in the adjacency list of w, we have the following DFS result and topological sort result which has only one "bad" edge: (v, w).

vetex	d	f
u	2	5
v	3	4
w	1	6



22.4-5

```
std::list<int> TopologicalSort(Graph& graph)
{
    int u;
    std::list<int> topo_sort;
    std::vector<int> in_degree_counter(graph.size, 0);
    std::queue<int> zero_in_degree;
    // compute in-degree
    for (AdjList& adj_list : graph.adj)
    {
        for (int v : adj_list)
        {
            ++in_degree_counter[v];
        }
    }
    // init queue contains vertices with zero in-degree
    for (u = 0; u < graph.size; ++u)
    {
        if (in_degree_counter[u] == 0)
        {
            zero_in_degree.push(u);
        }
    }
    // perform topological sort
    while (zero_in_degree.empty() == false)
    {
        u = zero_in_degree.front();
        zero_in_degree.pop();
        topo_sort.push_back(u);
        for (int v : graph.adj[u])
        {
```

```
31                --in_degree_counter[v];
32                if (in_degree_counter[v] == 0)
33                {
34                    zero_in_degree.push(v);
35                }
36            }
37        }
38        return topo_sort;
39    }
```

If the graph has cycles, the array (std::vector) "in_degree_counter" contains non-zero values at the end of the procedure, so there exist some vertices not contained in the result of topological sort.

22.5

22.5-1

The number of strongly connected components of a graph will stay same or decrease if a new edge is added.

22.5-2

By Exercise 22.3-2, the result of DFS(G) (the first DFS) is the following:

vetex	d	f
q	1	16
s	2	7
v	3	6
w	4	5
t	8	15
x	9	12
z	10	11
y	13	14
r	17	20
u	18	19

Hence DFS(G^T) (the second DFS), in the main loop of DFS, consider the vetices in the following order: r u q t y x z s v w.

Therefore, the strongly connected components of the graph is the following:

r

u

q y t

x z

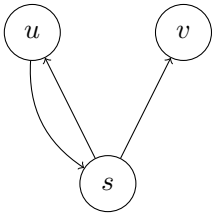
s w v

22.5-3

The claim is incorrect.

Construct the sequence $A = \langle a_1, a_2, \cdots, a_{|V|} \rangle$ where $a_i \in V$ for $i \in [1, |V|]$. WLOG, let $a_i.f$ to be strictly increase for $i \in [1, |V|]$. Let C be arbitrary strongly connected component. Let $B = \langle b_1, b_2, \cdots, b_{|C|} \rangle$ be the subsequence of A such that elements in B are vertex of C. Note that $f(C) = \max_{u \in C}\{u.f\}$. This says $f(C) = b_{|C|}.f$. In the main loop of the second DFS, the procedure consider the vetices in the order of A. This says, in the case of $|C| > 1$, the procedure will consider b_1 before consider $b_{|C|}$, unlike the process in Strongly-Connected-Components. However, we have $b_1.f = \min_{u \in C}\{u.f\}$. Let C' be a strongly connected component that is distinct from C. Hence, by lemma 22.14, there might be a edge (b_1, v) where $v \in C'$ and $b_1.f < v.f \leq f(C') < f(C)$ (especially $b_1.f < f(C')$), which means depth-first forest produced by the second DFS might not output strongly connected components correctly.

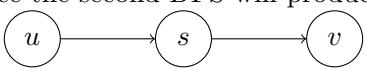
Consider the following counterexample:



Let s be the source vertex in the first DFS, and let u be in front of v in the adjacency list of s. The result of the first DFS is the following:

vetex	d	f
s	1	6
u	2	3
v	4	5

Then the second DFS, in the main loop, will consider the vetices in the following order: u v s. Hence the second DFS will produce the following depth-first forest:



The depth-first forest implies u-s-v are in a single strongly connected component, which is incorrect.

22.5-4

Proof. Let $G^{SCC} = (V^{SCC}, E^{SCC})$, Observed that G and G^T have exactly the same strongly connected components (page 616), we have $(G^T)^{SCC} = (V^{SCC}, (E^T)^{SCC})$. To show $G^{SCC} = ((G^T)^{SCC})^T$, we claim $E^{SCC} = ((E^T)^{SCC})^T$.

($\subseteq$) Suppose that $(u, v) \in E^{SCC}$. Then $u, v \in V^{SCC}$ and $(u, v) \in E$ where u and v are in distinct components. By definition of E^T, $(v, u) \in E^T$. Since u and v are in distinct components and $u, v \in V^{SCC}$, we have $(v, u) \in (E^T)^{SCC}$. Hence $(u, v) \in ((E^T)^{SCC})^T$.

($\supseteq$) Similar to the proof of ($\subseteq$). □

22.5-5

```
void DFSVisitComputeComponentGraph(Graph& graph, int u_idx,
    std::list<int>& adj_list)
{
    Vertex& u = graph.vertices[u_idx];
    for (int v_idx : graph.adj[u_idx])
    {
        Vertex& v = graph.vertices[v_idx];
        if (v.component < 0)
        {
            v.component = u.component;
            DFSVisitComputeComponentGraph(graph, v_idx, adj_list);
        }
        else if (v.component != u.component)
        {
            adj_list.push_back(v.component);
        }
    }
}

std::list< std::list<int> > DFSComputeComponentGraphAdjList(Graph& graph,
    const std::list<int>& order)
{
    int curr_component, v_idx;
    std::list< std::list<int> > adj_lists;
    for (Vertex& u : graph.vertices)
    {
        u.component = INT_MIN;
    }
    curr_component = 0;// index of component graph start from 0
    for (int u_idx : order)
    {
        Vertex& u = graph.vertices[u_idx];
        if (u.component < 0)
```

```
34          {
35              // DFS find edges (in adj_list) connected with curr_component
36              u.component = curr_component;
37              adj_lists.emplace_back(std::list<int>());
38              DFSVisitComputeComponentGraph(graph, u_idx, adj_lists.back());
39              ++curr_component;
40          }
41      }
42      return adj_lists;
43  }
44
45  Graph ComponentGraphAdjLists(Graph& graph)
46  {
47      int i;
48      std::list<int> topo_sort = TopologicalSort(graph);
49      Graph tranpose = Transpose(graph);
50      std::list< std::list<int> > tranpose_component_graph_adj_lists =
51          DFSComputeComponentGraphAdjList(tranpose, topo_sort);
52      // build ComponentGraph
53      Graph component_graph(tranpose_component_graph_adj_lists.size());
54      std::list< std::list<int> >::iterator adj_lists_it =
55          tranpose_component_graph_adj_lists.begin();
56      for (i = 0; i < component_graph.size; ++i)
57      {
58          // adjacency list might exist repeat elements
59          // we delete repeat elements by utilizing counter
60          // Note that the adjacency list is for (G^T)^{SCC}
61          // By Exercise 22.5-4, we showed that G^{SCC} = ((G^T)^{SCC})^T
62          std::vector<bool> counter(i + 1, false);
63          for (std::list<int>::iterator it = adj_lists_it->begin();
64              it != adj_lists_it->end(); ++it)
65          {
66              if (counter[*it] == false)
67              {
68                  counter[*it] = true;
69                  component_graph.adj[*it].push_back(i);
70              }
71          }
72          ++adj_lists_it;
73      }
```

```
    for (i = 0; i < graph.size; ++i)
    {
        component_graph.vertices_lists_from_original_graph
            [tranpose.vertices[i].component].push_back(i);
    }
    return component_graph;
}
```

22.5-6

Notice that a simple directed cycle constructed by vertices of the component minimized edges in each component. We can utilize the procedure in 25.5-5 to get component graph in $O(V + E)$. Therefore, we have the following algorithm which takes $O(V + E)$ time:

```
Graph CreateGraphExercise22_5_6(Graph& graph)
{
    int component_idx, u;
    Graph component_graph = ComponentGraphAdjLists(graph);
    Graph result_graph(graph.size);
    for (component_idx = 0; component_idx < component_graph.size; ++component_idx)
    {
        // build cycle
        std::list<int>& vertices =
            component_graph.vertices_lists_from_original_graph[component_idx];
        if (vertices.front() != vertices.back())
        {
            u = vertices.back();
            for (int v : vertices)
            {
                result_graph.adj[u].push_back(v);
                u = v;
            }
        }
        // build edges among components
        u = vertices.front();
        std::list<int>& adj = component_graph.adj[component_idx];
        for (int c : adj)
        {
            int v = component_graph.vertices_lists_from_original_graph[c].front();
            result_graph.adj[u].push_back(v);
        }
```

```
28        }
29        return result_graph;
30    }
```

22.5-7

Let $G = (V, E)$ and $G^{SCC} = (V^{SCC}, E^{SCC})$. Suppose that G has strongly connected components $C_1, C_2, \cdots, C_k$. The vertex set V^{SCC} is $\{s_1, s_2, \cdots, s_k\}$, and it contains a vertex s_i for each strongly connected component C_i of G.

Lemma 4. Let $x \in C_i$ and $y \in C_j$. $x \rightsquigarrow y \iff s_i \rightsquigarrow s_j$

Proof. ($\Longrightarrow$) Suppose that $x \rightsquigarrow y$. Then $\exists w_1, w_2, \cdots, w_p \in V$ such that $(x, w_1), (w_1, w_2), \cdots, (w_p, y) \in E$. Suppose that $c_1, c_2, \cdots c_p$ are the indexs of components contain $w_1, w_2, \cdots, w_p$; i.e. $w_l \in C_{c_l}$ for all $l \in [1, p]$. By definition of E^{SCC}, we have $(s_i, s_{c_1}), (s_{c_1}, s_{c_2}), \cdots, (s_{c_j}, s_j) \in E^{SCC}$. Thus, $s_i \rightsquigarrow s_j$.

($\Longleftarrow$) Suppose that $s_i \rightsquigarrow s_j$. Then $\exists w_1, w_2, \cdots, w_p \in V^{SCC}$ such that $(s_i, w_1), (w_1, w_2), \cdots, (w_p, s_j) \in E^{SCC}$. Suppose that $c_1, c_2, \cdots c_p$ are the indexs of components $w_1, w_2, \cdots, w_p$; i.e. $w_l = s_{c_l}$ for all $l \in [1, p]$. Pick arbitrary $u_l \in C_{c_l}$ for all $l \in [1, p]$. By definition of strongly connected components and E^{SCC}, we have $x \rightsquigarrow u_1 \rightsquigarrow u_2 \cdots u_p \rightsquigarrow y$. Thus, $x \rightsquigarrow y$. □

Lemma 5. G is semiconnected if and only if G^{SCC} is semiconnected.

Proof. ($\Longrightarrow$) Suppose that G is semiconnected. That is $\forall x, y \in V, x \rightsquigarrow y \vee y \rightsquigarrow x$. We prove by induction on the number of strongly connected components of G. Our inductive hypothesis is $\forall u, v \in \{s_1, s_2, \cdots, s_k\}, u \rightsquigarrow v \vee v \rightsquigarrow u$. The base case is when G containes no vertex, which is trivial. Suppose that $\forall u, v \in \{s_1, s_2, \cdots, s_k\}, u \rightsquigarrow v \vee v \rightsquigarrow u$. Let $x \in C_t$ where $t \in [1, k]$ and $y \in C_{k+1}$ be arbitrary choices. Since $x, y \in V$, we have $x \rightsquigarrow y \vee y \rightsquigarrow x$. If $x \rightsquigarrow y$, then, by lemma 4, we have $s_t \rightsquigarrow s_{k+1}$. Similarily, if $y \rightsquigarrow x$, we have $s_{k+1} \rightsquigarrow s_t$. Since $t \in [1, k]$ is a arbitrary choice, by induction, we conclude G^{SCC} is semiconnected.

($\Longleftarrow$) Suppose that G^{SCC} is semiconnected. That is $\forall x, y \in V^{SCC}, x \rightsquigarrow y \vee y \rightsquigarrow x$. We prove by induction on the number of strongly connectly components of G. Our inductive hypothesis is $\forall u, v \in C_1 \cup C_2 \cdots C_k, u \rightsquigarrow v \vee v \rightsquigarrow u$. The base case is when G containes no vertex, which is trivial. Suppose that $\forall u, v \in C_1 \cup C_2 \cdots C_k, u \rightsquigarrow v \vee v \rightsquigarrow u$. Let $x \in C_t$ where $t \in [1, k]$ and $y \in C_{k+1}$ be arbitrary choices. Since $s_t, s_{k+1} \in V^{SCC}$, we have $s_t \rightsquigarrow s_{k+1} \vee s_{k+1} \rightsquigarrow s_t$. If $s_t \rightsquigarrow s_{k+1}$, then, by lemma 4, we have $x \rightsquigarrow y$. Similarily, if $s_{k+1} \rightsquigarrow s_t$, we have $y \rightsquigarrow x$. Since $t \in [1, k]$ is a arbitrary choice, by induction, we conclude G is semiconnected. □

Lemma 6. G is semiconnected if and only if we are able to put all elements of V in a sequence $\langle v_1, v_2, \cdots, v_{|V|} \rangle$ such that $v_1 \rightsquigarrow v_2 \cdots v_{|V|-1} \rightsquigarrow v_{|V|}$.

Proof. ($\Longrightarrow$) Suppose that G is semiconnected. That is $\forall u, v \in V, u \rightsquigarrow v \vee v \rightsquigarrow u$. Let $A = \langle a_1, a_2, \cdots, a_k \rangle$ be a sequence. We prove by induction on k. The base case is $k = 0$, which is trivial. Suppose that $a_1 \rightsquigarrow a_2 \cdots a_{k-1} \rightsquigarrow a_k$. Pick arbitrary element $u \in V$ and $u \notin A$. If $\exists i \in [1, k)$ such that $a_i \rightsquigarrow u \rightsquigarrow a_{i+1}$, then we insert u between a_i and a_{i+1}; otherwise, insert u

before a_1 or after a_k appropriately. Re-assigning index of A, we get $A = \langle a_1, a_2, \cdots, a_k, a_{k+1} \rangle$ where $a_1 \rightsquigarrow a_2 \cdots a_k \rightsquigarrow a_{k+1}$.

($\Longleftarrow$) Immediately. □

Lemma 7. Let G be arbitrary DAG. If we are able to put all elements of V in a sequence $\langle v_1, v_2, \cdots, v_{|V|} \rangle$ such that $v_1 \rightsquigarrow v_2 \cdots v_{|V|-1} \rightsquigarrow v_{|V|}$, then $\{(v_1, v_2), (v_2, v_3), \cdots, (v_{|V|-1}, v_{|V|})\} \subseteq E$ and the result of topological sort order of G is $v_1, v_2, \cdots, v_{|V|}$.

Proof. Suppose that $v_1 \rightsquigarrow v_2 \cdots v_{|V|-1} \rightsquigarrow v_{|V|}$. If $|V| = 1$, the lemma is trivial correct. Let integer $i \in [1, |V|)$. Assume $(v_i, v_{i+1}) \notin E$, for the purpose of contradiction. Then $\exists j \neq i, i+1$ such that $v_i \rightsquigarrow v_j \rightsquigarrow v_{i+1}$. If $|V| = 2$, we obviously have a contradiction. If $j < i$, we have $v_i \rightsquigarrow v_j$ and $v_j \rightsquigarrow v_i$. If $j > i + 1$, we have $v_j \rightsquigarrow v_{i+1}$ and $v_{i+1} \rightsquigarrow v_j$. Contradict to G is a DAG.

In order to get other order of $\langle v_1, v_2, \cdots, v_{|V|} \rangle$. we must swap some elements in the sequence, and no linear ordering is possible, which contradicts to the definition of topological sort. □

Theorem 8. (Correctness) Suppose that the result of topological sort order of G^{SCC} is $v_1, v_2, \cdots, v_{|V^{SCC}|}$. G is semiconnected if and only if $\{(v_1, v_2), (v_2, v_3), \cdots, (v_{|V^{SCC}|-1}, v_{|V^{SCC}|})\} \subseteq E^{SCC}$

Proof. G is semiconnected $\overset{\text{lemma 5}}{\Longleftrightarrow}$ G^{SCC} is semiconnected $\overset{\text{lemma 6}}{\Longleftrightarrow}$
$\exists \langle v_1, v_2, \cdots, v_{|V^{SCC}|} \rangle$ where $v \in V^{SCC}$ such that $v_1 \rightsquigarrow v_2 \cdots v_{|V^{SCC}|-1} \rightsquigarrow v_{|V^{SCC}|}$

($\Longrightarrow$) Suppose that $v_1 \rightsquigarrow v_2 \cdots v_{|V^{SCC}|-1} \rightsquigarrow v_{|V^{SCC}|}$. Since G^{SCC} is a DAG, by lemma 7, we have $\{(v_1, v_2), (v_2, v_3), \cdots, (v_{|V|-1}, v_{|V|})\} \subseteq E$.

($\Longleftarrow$) Suppose that $\{(v_1, v_2), (v_2, v_3), \cdots, (v_{|V^{SCC}|-1}, v_{|V^{SCC}|})\} \subseteq E^{SCC}$. Then, obviously, we have $v_1 \rightsquigarrow v_2 \cdots v_{|V^{SCC}|-1} \rightsquigarrow v_{|V^{SCC}|}$. □

Therefore, we have the following $O(V + E)$ algorithm:

```
1  bool Semiconnected(Graph& graph)
2  {
3      int u;
4      Graph component_graph = ComponentGraphAdjLists(graph);
5      std::list<int> topo_sort = TopologicalSort(component_graph);
6      std::list<int>::iterator it = topo_sort.begin();
7      std::list<int>::iterator end = topo_sort.end();
8      u = *it;
9      ++it;
10     while (it != end)
11     {
12         if (std::find(component_graph.adj[u].begin(), component_graph.adj[u].end(),
13             *it) == component_graph.adj[u].end())
14             return false;
15         u = *it;
16     }
```

```
17      return true;
18  }
```

Chapter 22 Problems

22-1

(a)

Claim 9. There are no back edges in a BFS of an undirected graph.

Proof. Let $G = (V, E)$ be an undirected graph. Suppose that $(u, v) \in E$ is a back edge, for the purpose of contradiction. Then v is an ancestor of u in a breadth-first tree. If $u.\pi = v$, then (v, u) should be classified as a tree edge before (u, v) is classifed as a back edge. Now, consider the case of $u.\pi \neq v$. That is $(v, w_1), (w_1, w_2), \cdots, (w_{k-1}, w_k), (w_k, u) \in E_\pi$ for some $k \in \mathbb{N}$. We have $u.\pi = w_k$. However, since $(u, v) \in E$, we have $(v, u) \in E$, and (v, u) should be searched before (w_k, u) is searched, so $u.\pi$ should be v, and (v, u) should be classified as a tree edge. Contradiction. □

Claim 10. There are no forward edges in a BFS of an undirected graph.

Proof. Let $G = (V, E)$ be an undirected graph. Suppose that $(u, v) \in E$ is a forward edge, for the purpose of contradiction. Then v is an descendant of u in a breadth-first tree. If $v.\pi = u$, then (u, v) should be classified as a tree edge. Now, consider the case of $v.\pi \neq u$. That is $(u, w_1), (w_1, w_2), \cdots, (w_{k-1}, w_k), (w_k, v) \in E_\pi$ for some $k \in \mathbb{N}$. We have $v.\pi = w_k$. However, since $(u, v) \in E$, (u, v) should be searched before (w_k, v) is searched, so $v.\pi$ should be u, and (u, v) should be classified as a tree edge. Contradiction. □

Claim 11. In a BFS of an undirected graph, for each tree edge (u, v), we have $v.d = u.d + 1$.

Proof. Let $G = (V, E)$ be an undirected graph. Suppose that $(u, v) \in E$ is a tree edge. By definition, v was first discovered by exploring edge (u, v). Hence line 15 of BFS must be performed, which is setting $v.d = u.d + 1$. □

Claim 12. In a BFS of an undirected graph, for each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.

Proof. Let $G = (V, E)$ be an undirected graph. Suppose that $(u, v) \in E$ is a cross edge. Suppose that $v.d \neq u.d$ and $v.d \neq u.d+1$, for the purpose of contradiction. That is $u.d > v.d$ or $v.d > u.d+1$. $u.d > v.d$ is impossible since u is discovered before v. Note that u and v must be in a same breath-first tree since $(u, v) \in E$ and G is a undirected graph. Since $v.d > u.d + 1$, we have $(w, v) \in E$ for some $w \in V \setminus \{u, v\}$ where $w.d = v.d - 1$ and $v.\pi = w$. Then $w.d > u.d$. By lemma 22.3, u is dequeued before w. Then, when u is dequeued, $w.color$ is WHITE or GRAY, so $v.color$ is WHITE. However, since $(u, v) \in E$ and $v.color$ is WHITE, $v.\pi$ will become u, which means (u, v) is a tree edge. Contradiction. □

(b)

Claim 13. There are no forward edges in a BFS of an directed graph.

Proof. Same as the proof of claim 10. □

Claim 14. In a BFS of an directed graph, for each tree edge (u, v), we have $v.d = u.d + 1$.

Proof. Same as the proof of claim 11. □

Claim 15. In a BFS of an directed graph, for each cross edge (u, v), we have $v.d \leq u.d + 1$.

Proof. Let $G = (V, E)$ be an directed graph. Suppose that $(u, v) \in E$ is a cross edge. Then $v.color$ must be GRAY or BLACK when (u, v) is explored. Suppose that $v.d > u.d + 1$, for the purpose of contradiction. Then, by lemma 22.3, when u is dequeued, v has not been enqueued yet. This says $v.color$ is WHITE at the time that u is dequeued and at the time that (u, v) is explored. Contradiction.

Actually, $v.d \leq u.d + 1$ must be true for all kind of edge (u, v) since "when u is dequeued, v has not been enqueued" is contradicting to $(u, v) \in E$ directly. □

Claim 16. In a BFS of an directed graph, for each back edge (u, v), we have $0 \leq v.d \leq u.d$.

Proof. Let $G = (V, E)$ be an directed graph. Suppose that $(u, v) \in E$ is a back edge. Then v is an ancestor of u. We have $0 \leq v.d \leq u.d$ immediately. □

22-2

We assume $G = (V, E)$ is a simple, connected, undirected graph in this problem.

(a)

Claim 17. Let $G = (V, E)$ be a connected, undirected graph. Let s be the root of G_π. s is an articulation point of G if and only if s has at least two children in G_π.

Proof. ($\Longrightarrow$) Suppose that $s \in V$ is the root of G_π and is an articulation point of G. Let $G' = (V', E')$ be the graph if we remove s (i.e. $V' = V \setminus \{s\}$ and E' does not contain a edge connected s). Since s is an articulation point of G, there exists two non-empty, disjoint set $W, U \subseteq V'$ (i.e. $W, U \neq \emptyset$ and $W \cap U = \emptyset$) such that for every $w \in W$ and $u \in U$, $w \not\rightsquigarrow u$ and $u \not\rightsquigarrow w$ in G'.

Suppose that $v \in V$ is the only one child s has in $G.\pi$, for the purpose of contradiction. Let $E.\pi' = E.\pi \setminus \{(s, v)\}$ and $G.\pi' = (V', E.\pi')$. Then $E.\pi' \subseteq E'$ since (s, v) is the only edge in $E.\pi$ that connects s. Since G is a connected, undirected graph, by Theorem 22.10, $G.\pi$ must be connected also. WLOG, assume $v \in W$ and $v \notin U$. Let u be any vertex in U. Then u is a descendant of v in $G.\pi$ since $G.\pi$ is a tree where s is the root and v is the only child of s. Then there exist a path $v \rightsquigarrow u$ in $G.\pi$ that does not visit s. Thus, such path is in $G.\pi'$ also. Since $E.\pi' \subseteq E'$, this path is in G'. Contradiction.

($\Longleftarrow$) Suppose that s be the root of G_π. Let $C \subseteq V$ contains children of s. Let $u, w \in C$ where $u \neq w$. Let U contains vertices in the subtree rooted at u. Let W contains vertices in the subtree rooted at w. We claim for all $x \in U$ and $y \in W$, $x \rightsquigarrow y$ in G must visit s. Then removing s will lead to disconnection of U and W.

Suppose that there exist a path $x \rightsquigarrow y$ in G that does not visit s, for the purpose of contradiction. Then $(a, b) \in E$ for some $a \in U$ and $b \in W$. WLOG, assume u is explored before w in DFS. By Theorem 22.9 (White-path theorem), w is a descendant of u in $G.\pi$, which contradicts to u, w is children of s. □

(b)

Proof. Let v be a nonroot vertex of G_π.

($\Longrightarrow$) Suppose that v is an articulation point of G. v must have a child; otherwise, removing v will even not lead G_π disconnected, and G will naturally be disconnected. For the purpose of contradiction, suppose that for every subtree rooted at a child of v, there exist a vertex u in this subtree such that there is a back edge from u to a proper ancestor of v. Then removing v will not disconnect G. Contradiction.

($\Longleftarrow$) Suppose that v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v. Let S be the subtree rooted at s (S is a subgraph of G_π). Then there is no back edge from vertices in S to a proper ancestor of v. By Theorem 22.10, every edge from vertices in S to a proper ancestor is a tree edge. Then removing v will cause subtree S being disconnected from other part of G. □

(c)

```
1  void DFSVisit(Graph& graph, int u_idx, int& time)
2  {
3      Vertex& u = graph.vertices[u_idx];
4      ++time;
5      u.discovery_time = time;
6      u.color = Color::kGray;
7      u.low = u.discovery_time;
8      for (int v_idx : graph.adj[u_idx])
9      {
10         Vertex& v = graph.vertices[v_idx];
11         if (v.color == Color::kWhite)
12         {
13             // tree edge
14             v.prev = u_idx;
15             DFSVisit(graph, v_idx, time);
16             u.low = std::min(u.low, v.low);
```

```
        }
        else if (v.color == Color::kGray && u.prev != v_idx)
        {
            // back edge
            u.low = std::min(u.low, v.low);
        }
    }
    u.color = Color::kBlack;
    ++time;
    u.finishing_time = time;
}

void DFS(Graph& graph)
{
    int time, u_idx;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
        u.prev = -1;
    }
    time = 0;
    for (u_idx = 0; u_idx < graph.size; ++u_idx)
    {
        if (graph.vertices[u_idx].color == Color::kWhite)
        {
            DFSVisit(graph, u_idx, time);
        }
    }
}
```

(d)

Let p be a nonroot vertex of G_π. $\exists (p, v) \in E_\pi$ such that $v.low \geq p.d \iff \nexists (u, w) \in E$ such that (u, w) is a back edge $\land u$ is a descendant of $v \land w$ is a proper ancestor of $p \iff p$ is an articulation point

```
void DFSVisit(Graph& graph, int u_idx, int& time, std::list<int>& articulations)
{
    bool is_articulation;
    int tree_edges_counter;
    Vertex& u = graph.vertices[u_idx];
```

```
6       ++time;
7       u.discovery_time = time;
8       u.color = Color::kGray;
9       u.low = u.discovery_time;
10      is_articulation = false;
11      tree_edges_counter = 0;
12      for (int v_idx : graph.adj[u_idx])
13      {
14          Vertex& v = graph.vertices[v_idx];
15          if (v.color == Color::kWhite)
16          {
17              // tree edge
18              ++tree_edges_counter;
19              v.prev = u_idx;
20              DFSVisit(graph, v_idx, time, articulations);
21              u.low = std::min(u.low, v.low);
22              if (v.low >= u.discovery_time)
23                  is_articulation = true;
24          }
25          else if (v.color == Color::kGray && u.prev != v_idx)
26          {
27              // back edge
28              u.low = std::min(u.low, v.low);
29          }
30      }
31      u.color = Color::kBlack;
32      ++time;
33      u.finishing_time = time;
34      if ((u.discovery_time != 1 && is_articulation) ||
35          (u.discovery_time == 1 && tree_edges_counter >= 2))
36          articulations.push_back(u_idx);
37  }
38
39  std::list<int> DFS(Graph& graph)
40  {
41      int time, u_idx;
42      std::list<int> articulations;
43      for (Vertex& u : graph.vertices)
44      {
45          u.color = Color::kWhite;
```

```
46          u.prev = -1;
47      }
48      time = 0;
49      for (u_idx = 0; u_idx < graph.size; ++u_idx)
50      {
51          if (graph.vertices[u_idx].color == Color::kWhite)
52          {
53              DFSVisit(graph, u_idx, time, articulations);
54          }
55      }
56      return articulations;
57  }
```

(e)

Proof. ($\Longrightarrow$) Suppose that (u, v) is an bridge, and suppose that (u, v) lies on a simple cycle, for the purpose of contradiction. Then there exists at least two simple path from u to v. Disconnecting (u, v), we still have a path from u to v, which means G is still connected. Contradiction.

($\Longleftarrow$) We prove by contrapositive. Suppose that (u, v) is not an bridge. Then removing (u, v) will not disconnect G. Hence there must exist another path from u to v that does not visit (u, v), so such path and (u, v) form a simple cycle. □

(f)

$v.low = v.d \Longleftrightarrow \nexists (u, w) \in E$ such that (u, w) is a back edge $\wedge$ u is a descendant of $v \wedge w.d < v.d$ $\Longleftrightarrow$ $(v.\pi, v)$ lies on a simple cycle $\Longleftrightarrow$ $(v.\pi, v)$ is a bridge

```
1   void DFSVisit(Graph& graph, int u_idx, int& time,
2       std::list< std::pair<int, int> >& bridges)
3   {
4       Vertex& u = graph.vertices[u_idx];
5       ++time;
6       u.discovery_time = time;
7       u.color = Color::kGray;
8       u.low = u.discovery_time;
9       for (int v_idx : graph.adj[u_idx])
10      {
11          Vertex& v = graph.vertices[v_idx];
12          if (v.color == Color::kWhite)
13          {
14              // tree edge
15              v.prev = u_idx;
```

```
16              DFSVisit(graph, v_idx, time, bridges);
17              u.low = std::min(u.low, v.low);
18              if (v.low == v.discovery_time)
19                  bridges.push_back({ u_idx, v_idx });
20          }
21          else if (v.color == Color::kGray && u.prev != v_idx)
22          {
23              // back edge
24              u.low = std::min(u.low, v.low);
25          }
26      }
27      u.color = Color::kBlack;
28      ++time;
29      u.finishing_time = time;
30  }
31
32  std::list< std::pair<int, int> > DFS(Graph& graph)
33  {
34      int time, u_idx;
35      std::list< std::pair<int, int> > bridges;
36      for (Vertex& u : graph.vertices)
37      {
38          u.color = Color::kWhite;
39          u.prev = -1;
40      }
41      time = 0;
42      for (u_idx = 0; u_idx < graph.size; ++u_idx)
43      {
44          if (graph.vertices[u_idx].color == Color::kWhite)
45          {
46              DFSVisit(graph, u_idx, time, bridges);
47          }
48      }
49      return bridges;
50  }
```

(g)

Proof. Let E' be the set of the nonbridge edges of G (so $E' \subseteq E$). We prove that the biconnected components of G partition E' by defining a relation $\sim$ on E' and showing it is a equivlence class. By

the definition of biconnected component, the collection of biconnected components of G is exactly $E'/_{\sim}$.

For all $(u,v),(x,y) \in E'$, we define relation $\sim$ by $(u,v) \sim (x,y) \iff (u,v)$ and (x,y) lies on a common simple cycle of G.

(*Reflexivity*) Let $(u,v) \in E'$. Then (u,v) is a nonbridge edge. By part (e), (u,v) lies on a simple cycle of G. Clearly, (u,v) and (u,v) on a common simple cycle. Hence $(u,v) \sim (u,v)$.

(*Symmetry*) Let $(u,v),(x,y) \in E'$. Suppose that $(u,v) \sim (x,y)$. Then (u,v) and (x,y) lies on a common simple cycle of G. It is trivial to show that $(x,y) \sim (u,v)$.

(*Transitivity*) Let $(u,v),(x,y),(a,b) \in E'$. Suppose that $(u,v) \sim (x,y)$ and $(x,y) \sim (a,b)$. Then (u,v) and (x,y) lies on a common simple cycle C_1, and (x,y) and (a,b) lies on a common simple cycle C_2.

Define symbol $u \overset{P}{\rightsquigarrow} v$ as the path from u to v ($u,v \notin P$) where P is the set contains vertices on the path from u to v.

WLOG (we can swap "u" and "v" or swap "x" and "y"), let $u \overset{P_1}{\rightsquigarrow} x$ and $v \overset{P_2}{\rightsquigarrow} y$ be the simple paths that formed cycle C_1, so we have $P_1 \cap P_2 = \emptyset$. WLOG, let $x \overset{P_3}{\rightsquigarrow} a$ and $y \overset{P_4}{\rightsquigarrow} b$ be the simple paths that formed cycle C_2, so we have $P_3 \cap P_4 = \emptyset$. Our goal is to form a simple cycle contains (u,v) and (a,b) by forming path $u \overset{A}{\rightsquigarrow} a$ and $v \overset{B}{\rightsquigarrow} b$ or forming path $u \overset{A}{\rightsquigarrow} b$ and $v \overset{B}{\rightsquigarrow} a$ where $A \cap B = \emptyset$ Note that we can find non-simple paths for A and B where $A \cap B = \emptyset$ first and transform them into simple paths without changing endpoints and violating $A \cap B = \emptyset$.

Case 1. Construct $u \overset{P_5}{\rightsquigarrow} a = u \overset{P_1}{\rightsquigarrow} x \overset{P_3}{\rightsquigarrow} a$, and construct $v \overset{P_6}{\rightsquigarrow} b = v \overset{P_2}{\rightsquigarrow} y \overset{P_4}{\rightsquigarrow} b$. We have $P_5 \cap P_6 = \emptyset$. Transforming P_5 and P_6 into simple path, we can construct the simple cycle by using path P_5 and P_6.

Case 2. $P_5 \cap P_6 \neq \emptyset$. Then $u \overset{P_5}{\rightsquigarrow} a = u \overset{P_{5.1}}{\rightsquigarrow} I_1 \overset{P_{5.2}}{\rightsquigarrow} a$ and $v \overset{P_6}{\rightsquigarrow} b = v \overset{P_{6.1}}{\rightsquigarrow} I_2 \overset{P_{6.2}}{\rightsquigarrow} b$ where I_1, I_2 are sets of vertices and $P_5 \cap P_6 \subseteq I_1$, $P_5 \cap P_6 \subseteq I_2$, so $(P_{5.1} \cup P_{5.2}) \cap (P_{6.1} \cup P_{6.2}) = \emptyset$. Let $I_1.f$ be the first vertex in I_1 on the path $u \overset{P_5}{\rightsquigarrow} a$ and $I_1.l$ be the last vertex in I_1 on the path. Let $I_2.f$ be the first vertex in I_2 on the path $v \overset{P_6}{\rightsquigarrow} b$ and $I_2.l$ be the last vertex in I_2 on the path. WLOG, assume $I_1.f, I_1.l \in I_2$ and $I_2.f, I_2.l \in I_1$. We claim ① $x \in P_{5.1}$ and $y \in P_{6.2}$, ② $x \in P_{5.2}$ and $y \in P_{6.1}$, or ③ $x \in I_1$ and $y \in I_2$, and show it by contradiction.

Suppose that $x \in I_1$ and $y \notin I_2$, for the purpose of contradiction. Then $u \overset{P_5}{\rightsquigarrow} a = u \overset{P_{5.1}}{\rightsquigarrow} I_{1.1} \overset{\emptyset}{\rightsquigarrow} x \overset{\emptyset}{\rightsquigarrow} I_{1.2} \overset{P_{5.2}}{\rightsquigarrow} a$ where $I_1 = I_{1.1} \cup \{x\} \cup I_{1.2}$. We have $P_1 = P_{5.1} \cup I_{1.1}$, $P_3 = I_{1.2} \cup P_{5.2}$, and $I_1.f \in I_{1.1}$, $I_1.l \in I_{1.2}$. Since $y \notin I_2$, there are two cases: $I_2 \subseteq P_2$ or $I_2 \subseteq P_4$. If $I_2 \subseteq P_2$, then $I_1.f \in I_{1.1} \wedge I_1.f \in I_2 \implies I_1.f \in I_{1.1} \cap I_2 \implies I_1.f \in P_1 \cap P_2 \implies P_1 \cap P_2 \neq \emptyset$, which is a contradiction. If $I_2 \subseteq P_4$, then $I_1.l \in I_{1.2} \wedge I_1.l \in I_2 \implies I_1.l \in I_{1.2} \cap I_2 \implies I_1.l \in P_3 \cap P_4 \implies P_3 \cap P_4 \neq \emptyset$, which is a contradiction. We have a contradiction for $x \notin I_1$ and $y \in I_2$ by the similar way, so we conclude $x \in I_1 \iff y \in I_2$.

Suppose that $x \in P_{5.1}$ and $y \in P_{6.1}$, for the purpose of contradiction. Then $I_1 \subseteq P_3$ and $I_2 \subseteq P_4$. Since $I_1 \cap I_2 \neq \emptyset$, we have $P_3 \cap P_4 \neq \emptyset$, which is a contradiction. We have a contradiction for $x \in P_{5.2}$ and $y \in P_{6.2}$ by the similar way, so we conclude $x \notin I_1 \implies (x \in P_{5.1} \wedge y \in P_{6.2}) \vee (x \in P_{5.2} \wedge y \in P_{6.1})$.

We have showed there are only three subcases: ① $x \in P_{5.1}$ and $y \in P_{6.2}$, ② $x \in P_{5.2}$ and $y \in P_{6.1}$, or ③ $x \in I_1$ and $y \in I_2$.

① Suppose that $x \in P_{5.1}$ and $y \in P_{6.2}$. WLOG, assume $I = I_1 = I_2$. Then there are two subsubcases according to whether or not $P_1 \cap P_{5.2} = \emptyset \wedge P_4 \cap P_{6.1} = \emptyset$. If $P_1 \cap P_{5.2} = \emptyset \wedge P_4 \cap P_{6.1} = \emptyset$, we consider paths $u \overset{A}{\rightsquigarrow} b = u \overset{P_1}{\rightsquigarrow} x \overset{\emptyset}{\rightsquigarrow} y \overset{P_4}{\rightsquigarrow} b$ and $v \overset{B}{\rightsquigarrow} a = v \overset{P_{6.1}}{\rightsquigarrow} I \overset{P_{5.2}}{\rightsquigarrow} a$. We claim $A \cap B = \emptyset$ so we have a simple cycle contains (u, v) and (a, b). We need to show that $(P_1 \cup \{x, y\} \cup P_4) \cap (P_{6.1} \cup I \cup P_{5.2}) = \emptyset$. Since $P_1 \cap P_2 = \emptyset$, $P_3 \cap P_4 = \emptyset$, and $u \overset{P_1}{\rightsquigarrow} x$, $v \overset{P_2}{\rightsquigarrow} y$, $x \overset{P_3}{\rightsquigarrow} a$, $y \overset{P_4}{\rightsquigarrow} b$ are simple paths, We just need to show that $(P_1 \cup P_4) \cap (P_{6.1} \cup P_{5.2}) = \emptyset$. Since $(P_{5.1} \cup P_{5.2}) \cap (P_{6.1} \cup P_{6.2}) = \emptyset$, we have $P_1 \cup P_{6.1} = \emptyset$ and $P_4 \cup P_{5.2} = \emptyset$. By the hypothesis of the subsubcase, we have $P_1 \cup P_{5.2} = \emptyset$ and $P_4 \cup P_{6.1} = \emptyset$. Thus, we have showed $A \cap B = \emptyset$ in this subsubcase. If $P_1 \cap P_{5.2} \neq \emptyset \vee P_4 \cap P_{6.1} \neq \emptyset$, and, WLOG, we assume $P_4 \cap P_{6.1} \neq \emptyset$, then $\exists s \in P_4 \cap P_{6.1}$. Consider paths $u \overset{A}{\rightsquigarrow} a = u \overset{P_5}{\rightsquigarrow} a$ and $v \overset{B}{\rightsquigarrow} b = v \overset{P_7}{\rightsquigarrow} s \overset{P_8}{\rightsquigarrow} a$ where $P_7 \subseteq P_{6.1}$ and $P_8 \subseteq P_4$. We claim $A \cap B = \emptyset$ so we have a simple cycle contains (u, v) and (a, b). We need to show that $P_5 \cap (P_7 \cup \{s\} \cup P_8) = \emptyset$. Since $(P_{5.1} \cup P_{5.2}) \cap (P_{6.1} \cup P_{6.2}) = \emptyset$ and $P_7 \cup \{s\} \cup P_8 \subseteq P_{6.1} \cup P_{6.2}$, we have $(P_{5.1} \cup P_{5.2}) \cap (P_7 \cup \{s\} \cup P_8) = \emptyset$. Since P_2 is a simple path, we have $I \cap (P_7 \cup \{s\}) = \emptyset$. Since $I \subseteq P_3 \wedge P_8 \subseteq P_4 \wedge P_3 \cap P_4 = \emptyset$, we have $I \cap P_8 = \emptyset$. Hence we have $I \cap (P_7 \cup \{s\} \cup P_8) = \emptyset$ also, and we have showed $A \cap B = \emptyset$ in this subsubcase. In subcase ②, we can form a simple cycle contains (u, v) and (a, b) and prove it in the similar way of ①.

③ Suppose that $x \in I_1$ and $y \in I_2$. Then we have $u \overset{P_5}{\rightsquigarrow} a = u \overset{P_{5.1}}{\rightsquigarrow} I_{1.1} \overset{\emptyset}{\rightsquigarrow} x \overset{\emptyset}{\rightsquigarrow} I_{1.2} \overset{P_{5.2}}{\rightsquigarrow} a$ where $I_1 = I_{1.1} \cup \{x\} \cup I_{1.2}$ and $v \overset{P_6}{\rightsquigarrow} b = v \overset{P_{6.1}}{\rightsquigarrow} I_{2.1} \overset{\emptyset}{\rightsquigarrow} y \overset{\emptyset}{\rightsquigarrow} I_{2.2} \overset{P_{6.2}}{\rightsquigarrow} b$ where $I_2 = I_{2.1} \cup \{y\} \cup I_{2.2}$. We have $I_{1.1} \subseteq P_1$, $I_{2.1} \subseteq P_2$, $I_{1.2} \subseteq P_3$, and $I_{2.2} \subseteq P_4$. Since $P_1 \cap P_2 = \emptyset$ and $P_3 \cap P_4 = \emptyset$, we have $I_{1.1} \cap I_{2.1} = \emptyset$ and $I_{1.2} \cap I_{2.2} = \emptyset$. Since $I_1 \cap I_2 = \emptyset$ and $x \notin I_2$, $y \notin I_1$, we have $I_{1.1} \cap I_{2.2} \neq \emptyset$ or $I_{1.2} \cap I_{2.1} \neq \emptyset$. WLOG, assume $I_{1.2} \cap I_{2.1} \neq \emptyset$.

Similar to ①, and, WLOG, assume $I = I_{1.2} = I_{2.1}$ and $I' = I_{1.1} = I_{2.2}$, there are two subsubcases: if $P_1 \cap P_{5.2} = \emptyset \wedge P_4 \cap P_{6.1} = \emptyset$, we have two paths $u \overset{A}{\rightsquigarrow} b = u \overset{P_1}{\rightsquigarrow} x \overset{\emptyset}{\rightsquigarrow} y \overset{P_4}{\rightsquigarrow} b$ and $v \overset{B}{\rightsquigarrow} a = v \overset{P_{6.1}}{\rightsquigarrow} I \overset{P_{5.2}}{\rightsquigarrow} a$ and $A \cap B = \emptyset$; if $P_1 \cap P_{5.2} \neq \emptyset \vee P_4 \cap P_{6.1} \neq \emptyset$, and, WLOG, we assume $P_4 \cap P_{6.1} \neq \emptyset$, then $\exists s \in P_4 \cap P_{6.1}$, so we have two paths $u \overset{A}{\rightsquigarrow} a = u \overset{P_5}{\rightsquigarrow} a$ and $v \overset{B}{\rightsquigarrow} b = v \overset{P_7}{\rightsquigarrow} s \overset{P_8}{\rightsquigarrow} a$ where $P_7 \subseteq P_{6.1}$ and $P_8 \subseteq P_{6.2} \subseteq P_4$ ($P_8 \cap I' = \emptyset$ since $s \notin I'$; otherwise $s \in I' \wedge I' \subseteq P_1 \wedge s \in P_{6.1} \wedge P_{6.1} \subseteq P_2 \Longrightarrow s \in P_1 \cap P_2$, contradiction) and $A \cap B = \emptyset$. Thus, we have a simple cycle contains (u, v) and (a, b).

Therefore, $(u, v) \sim (a, b)$. We have showed relation $\sim$ is a equivlence class on E'. □

(h)

Lemma 18. Let $G = (V, E)$ be a simple, undirected, connected graph and $x, y \in V$. After DFS, we have $(x.d \leq y.d) \Longrightarrow (\exists$ simple path $x \overset{A}{\rightsquigarrow} y$ such that $(\forall a \in A, a.d < y.d))$.

Proof. If x is an ancestor of y, we can simply connect tree edges between x and y, and we have $\forall a \in A, x.d < a.d < y.d$. If x is not an ancestor of y, then let s be the vertex where s is the ancestor of x and y with the grestest possible $s.d$. Note that such s must exist since G is a connected graph. Construct $x \overset{A}{\rightsquigarrow} y = x \overset{B}{\rightsquigarrow} s \overset{C}{\rightsquigarrow} y$ where $x \overset{B}{\rightsquigarrow} s$ is formed by connecting all tree edges from x to s and $s \overset{C}{\rightsquigarrow} y$ is formed by connecting all tree edges from s to y, so $A = B \cup \{s\} \cup C$, and B and C are simple paths. Clearly, $\forall a \in A, a.d < \max(x.d, y.d) = y.d$. We also have $B \cap C = \emptyset$ since otherwise $p.d > s.d$ where $p \in B \cap C$, which contradicts s is the candidate with the grestest possible $s.d$. Thus,

$x \overset{A}{\rightsquigarrow} y$ is a simple path. □

Lemma 19. Let $G = (V, E)$ be a simple, undirected, connected graph and $u, v \in V$ where u is nonroot vertex of G_π. Let E' be the set of the nonbridge edges of G (so $E' \subseteq E$). (u, v) is a back edge $\implies (u, v) \in E' \wedge (u, v).bcc = (u.\pi, u).bcc$.

Proof. Consider the cycle formed by the path $v \overset{A}{\rightsquigarrow} u.\pi \overset{\emptyset}{\rightsquigarrow} u \overset{\emptyset}{\rightsquigarrow} v$. We construct $u \overset{\emptyset}{\rightsquigarrow} v$ by using the back edge (u, v) and construct $u.\pi \overset{\emptyset}{\rightsquigarrow}$ by using the tree edge $(u.\pi, u)$ simply. Since (u, v) is a back edge, we have $v.d < u.d$, so $v.d \leq u.\pi.d$. By lemma 18, $\exists$ simple path $v \overset{A}{\rightsquigarrow} u.\pi$ such that $(\forall a \in A, a.d < u.\pi.d)$. Hence (u, v) and $(u.\pi, u)$ lies on the common simple cycle, which means $(u, v).bcc = (u.\pi, u).bcc$. □

Claim 20. Let $G = (V, E)$ be a simple, undirected, connected graph and $u, v \in V$ where u is nonroot vertex of G_π. Let E' be the set of the nonbridge edges of G (so $E' \subseteq E$). After DFS, we have $v.low < u.d \iff (u, v) \in E' \wedge (u, v).bcc = (u.\pi, u).bcc$.

Proof. ($\implies$) Suppose that $v.low < u.d$.

Case 1. (u, v) is tree edge.

Then $u.d = v.d - 1$ and $\exists (z, w) \in E$ such that (z, w) is a back edge $\wedge$ z is a descendant of $v \wedge w.d < u.d$. Hence $w.d \leq u.\pi.d$. By lemma 18, $\exists$ simple path $w.d \overset{A}{\rightsquigarrow} u.\pi$ such that $(\forall a \in A, a.d < u.\pi.d)$. Consider the cycle formed by the path $w.d \overset{A}{\rightsquigarrow} u.\pi \overset{\emptyset}{\rightsquigarrow} u \overset{\emptyset}{\rightsquigarrow} v \overset{B}{\rightsquigarrow} z \overset{\emptyset}{\rightsquigarrow} w$ where $v \overset{B}{\rightsquigarrow} z$ is formed by connecting all tree edges from v to z Since $(\forall a \in A, a.d < u.\pi.d)$, $(\forall b \in B, b.d > v.d)$, and $u.\pi.d + 1 = u.d = v.d - 1$, we conclude $A \cap B = \emptyset$. Therefore, it is a simple cycle, and (u, v) and $(u.\pi, u)$ lies on the common simple cycle, which means $(u, v).bcc = (u.\pi, u).bcc$.

Case 2. (u, v) is back edge.

Immediately from lemma 19.

($\impliedby$) We prove by contrapositive. Suppose that $v.low \geq u.d$. (u, v) cannot be a back edge; otherwise, we have $u.d > v.d \geq v.low$, which contradicts to our hypothesis. Hence (u, v) is a tree edge. Then $u.d = v.d - 1$ and $v.low \geq u.d = v.d - 1$. Since $v.low \leq v.d$, there are two case: $v.low = v.d$ or $v.low = v.d - 1$.

Case 1. $v.low = v.d$.

By part (f), we have $(v.\pi, v) = (u, v)$ is a bridge. Hence $(u, v) \notin E'$.

Case 2. $v.low = v.d - 1$.

Then $(v.\pi, v) = (u, v)$ is not a bridge. Hence $(u, v) \notin E'$. Suppose that $(u, v).bcc = (u.\pi, u).bcc$, for the purpose of contradiction. Then (u, v) and $(u.\pi, u)$ lies on a common simple cycle. We already have $u.\pi \overset{\emptyset}{\rightsquigarrow} u \overset{\emptyset}{\rightsquigarrow} v$ formed by tree edges. We also need a back edge (z, w) such that z is a descendant of v and $w.d \leq u.\pi.d$. Then we have $v.low \leq u.\pi.d$, which contradicts to $v.low = v.d - 1 = u.d$. □

```
1  void DFSVisitLabelVertice(Graph& graph, int u_idx, int& time)
2  {
3      Vertex& u = graph.vertices[u_idx];
4      ++time;
```

```
5       u.discovery_time = time;
6       u.color = Color::kGray;
7       u.low = u.discovery_time;
8       for (Edge& e : graph.adj[u_idx])
9       {
10          Vertex& v = graph.vertices[e.vertex];
11          if (v.color == Color::kWhite)
12          {
13              // tree edge
14              v.prev = u_idx;
15              DFSVisitLabelVertice(graph, e.vertex, time);
16              u.low = std::min(u.low, v.low);
17          }
18          else if (v.color == Color::kGray && u.prev != e.vertex)
19          {
20              // back edge
21              u.low = std::min(u.low, v.low);
22          }
23      }
24      u.color = Color::kBlack;
25      ++time;
26      u.finishing_time = time;
27  }
28
29  void DFSVisitLabelEdges(Graph& graph, int u_idx, int last_bcc, int& bcc_counter)
30  {
31      Vertex& u = graph.vertices[u_idx];
32      u.color = Color::kGray;
33      for (Edge& e : graph.adj[u_idx])
34      {
35          Vertex& v = graph.vertices[e.vertex];
36          if (v.color == Color::kWhite)
37          {
38              // tree edge
39              if (v.low < u.discovery_time)
40                  e.bcc = last_bcc;
41              else if (v.discovery_time == v.low)
42                  e.bcc = -1;// bridge edge
43              else
44                  e.bcc = ++bcc_counter;
```

```
45
46              DFSVisitLabelEdges(graph, e.vertex, e.bcc, bcc_counter);
47          }
48          else if (v.color == Color::kGray && u.prev != e.vertex)
49          {
50              // back edge
51              e.bcc = last_bcc;
52          }
53      }
54      u.color = Color::kBlack;
55  }
56
57  // graph: simple, connected, undirected
58  void LabelBcc(Graph& graph)
59  {
60      int time, bcc_counter;
61      // first DFS: compute prev, discovery_time, finishing_time,
62      //      low attributes for each vertex
63      graph.vertices[0].prev = -1;
64      for (Vertex& u : graph.vertices)
65      {
66          u.color = Color::kWhite;
67      }
68      time = 0;
69      DFSVisitLabelVertice(graph, 0, time);
70      // second DFS: label bcc attributes for each edge
71      for (Vertex& u : graph.vertices)
72      {
73          u.color = Color::kWhite;
74      }
75      bcc_counter = 0;
76      DFSVisitLabelEdges(graph, 0, 0, bcc_counter);
77  }
```

22-3

(a)

Lemma 21. If in-degree(v) = out-degree(v) for each vertex $v \in V$, then (for all $u \in V$, in-degree(u) = out-degree(u) $\geq 1 \Longrightarrow$($\exists$ cycle contains u and in-degree(v) = out-degree(v) for each vertex $v \in V$ after removing each edges in the cycle)).

Note that the path of a cycle contains at least one edge.

Proof. Suppose that in-degree(v) = out-degree(v) for each vertex $v \in V$. Let $u \in V$ be arbitrary choice. Suppose that in-degree(u) = out-degree(u) ≥ 1. We inductively construct an cycle $\langle a_1, a_2, \cdots, a_n \rangle$ contains u, where k is our induction varible. Let $a_1 = u$. Assume we remove the edge after we visited it. The base case is $k = 2$: since out-degree(u) ≥ 1, $\exists (u, w) \in E$, so we remove (u, w) and let $a_2 = w$, and we have in-degree(a_1) = out-degree(a_1) + 1 and in-degree(a_2) = out-degree(a_2) − 1. Assume in-degree(a_k) = out-degree(a_k) − 1 for some $k \geq 2$ and in-degree(a_j) = out-degree(a_j) for all $j = 2, \cdots, k-1$, which is our inductive hypothesis. Then out-degree(a_k) = in-degree(a_k) + 1 ≥ 1. We have $\exists (a_k, x) \in E$. We remove (a_k, x) and let $a_{k+1} = x$, which means in-degree(a_{k+1}) and out-degree(a_k) decreased by 1. If $a_k = u$, then we stop our construction and have a cycle contains u, and we conclude in-degree(v) = out-degree(v) for all $v \in V$ at this time (after construction of the cycle). If $a_k \neq u$, then we have in-degree(a_k) = out-degree(a_k) and in-degree(a_{k+1}) = out-degree(a_{k+1}) − 1, so our inductive hypothesis hold for next induction varible. □

Claim 22. G has an Euler tour $\iff$ in-degree(v) = out-degree(v) for each vertex $v \in V$.

Proof. ($\Longrightarrow$) We prove by contrapositive. Suppose that there exists $v \in V$ such that in-degree(v) $\neq$ out-degree(v). If in-degree(v) > out-degree(v), then there will be in-degree(v) − out-degree(v) edges to v cannot be visit since the tour need to go into v in-degree(v) times and only go out v out-degree(v) times then the tour will stuck in v. Similarily, if in-degree(v) < out-degree(v), then there will be out-degree(v) − int-degree(v) edges from v cannot be visit. Hence G does not have an Euler tour.

($\Longleftarrow$) Notice the our tour is actually a cycle unless there are only one node in the graph, which is trivial. Assume there are at least two nodes in the graph. Then we the graph contains at least one cycle. Suppose that in-degree(v) = out-degree(v) for each vertex $v \in V$ before we start the construction. We inductively construct the set of cycles $S = \{C : C \text{ is a cycle}\}$ where the induction varible is $|S|$. Assume we remove the edges that in the cycles from the graph. The inductive hypothesis is in-degree(v) = out-degree(v) for all $v \in V$. The base case is $|S| = 0$, which is before the start of the construction By our hypothesis of the problem, the inductive hypothesis hold for next step. Suppose that the inductive hypothesis is true after cycles in S are constructed where $|S| = k$. If $\exists u \in V$ such that in-degree(u) = out-degree(u) ≥ 1, then by lemma 21, we have a cycle C contains u. We insert this C into the set S and $|S| = k + 1$. Since in-degree(v) = out-degree(v) for each vertex $v \in V$ after removing each edges in the cycle C, our inductive hypothesis hold for next step. If $\forall u \in V$, in-degree(u) = out-degree(u) = 0, then we stop the indcution and all edges are removed (visited).

Now we reconsider the removed edges, and we have cycles in S contains all edges in the graph. In order to form an Euler tour, we just need to merge these cycles to a single cycle, which is exactly our Euler tour. If $|S| = 1$, then the only cycle in S is our Euler tour. If $|S| > 1$, then we claim $\exists C, D \in S$ where $C = \langle c_1, c_2, \cdots, c_m \rangle$ and $D = \langle d_1, d_2, \cdots, d_n \rangle$ such that $c_i = d_j$ for some integers $i \in [1, m]$ and $j \in [1, n]$; otherwise, the graph is not strongly connected. We merge C and D at the

vertex $c_i = d_j$ by forming cycle $\langle c_1, c_2, \cdots, c_i, d_{j+1}, d_{j+2}, \cdots, d_n, d_1, \cdots, d_{j-1}, c_{i+1}, \cdots, c_m \rangle$. Clearly, we can inductively merge all cycles in S and find an Euler tour. □

(b)

```
std::list< std::pair<int, int> > EulerTour(Graph& graph)
{
    int u, v, s;

    if (graph.size <= 1)
        return std::list< std::pair<int, int> >();

    std::vector< AdjList > adjs_copy = graph.adj;// make a copy of adjacency list
    std::list< std::pair<int, int> > tour;// result
    std::vector< std::list< std::pair<int, int> >::iterator >
        tour_its(graph.size, tour.end());// iterators of tours for each insert-availble
                                         // (in the tour list) vertices
    std::queue<int> todo;// queue of vertices in the tour list already

    todo.push(0);

    while (todo.empty() == false)
    {
        u = todo.front();
        s = u;
        todo.pop();
        std::list< std::pair<int, int> >::iterator tour_curr_it = tour_its[u];
            // current iterator of insert postion in the tour list

        while (adjs_copy[u].empty() == false)
        {
            v = adjs_copy[u].front();
            tour_its[u] = tour.insert(tour_curr_it, { u, v });
            adjs_copy[u].pop_front();
            todo.push(u);
            u = v;
        }

        if (s != u)
            return std::list< std::pair<int, int> >();
```

```
    }

    return tour;
}
```

22-4

Approach 1 Find the strongly connected components (SCC) of the graph first. Each pair in a SCC are reachable each other, so we can treat each SCC as a whole, and take the minimum value in the SCC as the value of the SCC. Then the SCC graph is a DAG, so we can utilize a simple dynamic programming. Define $L(u)$ and $R(u)$ on $G^{SCC} = (V^{SCC}, E^{SCC})$ with the same definition of G. Denote $c[u]$ as $\min\{L(w) : w \in R(u)\}$ for all $u \in V^{SCC}$.

$$c[u] = \min \begin{cases} L(u) \\ \min\limits_{(u,v)\in E^{SCC}}(c[v]) \end{cases}$$

```
void TopoSortVisit(Graph<OriginalVertex>& graph, int u,
    std::list<int>& topo_sort_result)
{
    graph.vertices[u].color = Color::kNonWhite;
    for (int v : graph.adj[u])
    {
        if (graph.vertices[v].color == Color::kWhite)
            TopoSortVisit(graph, v, topo_sort_result);
    }
    topo_sort_result.push_front(u);
}

// component_adj_list is for tranpose graph and might exist repeat elements
void SCCTranposeVisit(Graph<TranposeVertex>& tranpose, int u,
    std::list<int>& component_vertices, std::list<int>& component_adj_list,
    int curr_component)
{
    tranpose.vertices[u].color = Color::kNonWhite;
    tranpose.vertices[u].component = curr_component;
    component_vertices.push_back(u);
    for (int v : tranpose.adj[u])
    {
        if (tranpose.vertices[v].color == Color::kWhite)
            SCCTranposeVisit(tranpose, v, component_vertices,
                component_adj_list, curr_component);
```

```
        else if (tranpose.vertices[v].component != curr_component)
            // default value of component attribute is INT_MIN
            component_adj_list.push_back(tranpose.vertices[v].component);
    }
}

void DpSCCVisit(Graph<SCCVertex>& scc, int u)
{
    scc.vertices[u].color = Color::kNonWhite;
    for (int v : scc.adj[u])
    {
        if (scc.vertices[v].color == Color::kWhite)
        {
            DpSCCVisit(scc, v);
            if (scc.vertices[u].dp_min_scc_value > scc.vertices[v].dp_min_scc_value)
            {
                scc.vertices[u].dp_min_scc_vertex = scc.vertices[v].dp_min_scc_vertex;
                scc.vertices[u].dp_min_scc_value = scc.vertices[v].dp_min_scc_value;
            }
        }
    }
}

void Reachability(Graph<OriginalVertex>& graph)
{
    int u, component_counter;
    std::list<int> topo_sort_result;
    std::list< std::list<int> > component_vertices_collection,
        tranpose_component_adj_list_collection;
    // topo sort
    graph.WhitenAllVertices();
    for (u = 0; u < graph.size; ++u)
    {
        if (graph.vertices[u].color == Color::kWhite)
            TopoSortVisit(graph, u, topo_sort_result);
    }
    // find tranpose
    Graph<TranposeVertex> tranpose = Tranpose(graph);
    // find strongly connected components
    tranpose.WhitenAllVertices();
```

```
66      component_counter = 0;
67      for (int w : topo_sort_result)
68      {
69          if (tranpose.vertices[w].color == Color::kWhite)
70          {
71              component_vertices_collection.emplace_back();
72              tranpose_component_adj_list_collection.emplace_back();
73              SCCTranposeVisit(tranpose, w, component_vertices_collection.back(),
74                  tranpose_component_adj_list_collection.back(), component_counter);
75              ++component_counter;
76          }
77      }
78      // make scc graph
79      Graph<SCCVertex> scc(component_counter);
80      u = 0;
81      for (std::list<int>& component_vertices : component_vertices_collection)
82      {
83          scc.vertices[u].vertices.assign(component_vertices.begin(),
84              component_vertices.end());
85          ++u;
86      }
87      u = 0;
88      for (std::list<int>& component_adj_list : tranpose_component_adj_list_collection)
89      {
90          std::vector<bool> repeat_counter(scc.size, false);
91          for (int v : component_adj_list)
92          {
93              if (repeat_counter[v] == false)
94              {
95                  scc.adj[v].push_back(u);
96                  repeat_counter[v] = true;
97              }
98          }
99          ++u;
100     }
101     // init dp_min_scc_vertex and dp_min_scc_value for each scc vertex
102     for (SCCVertex& scc_v : scc.vertices)
103     {
104         scc_v.dp_min_scc_vertex = *(std::min_element(scc_v.vertices.begin(),
105             scc_v.vertices.end(), [&graph](int a, int b) {
```

```
106                 return graph.vertices[a].value < graph.vertices[b].value;
107             }));
108         scc_v.dp_min_scc_value = graph.vertices[scc_v.dp_min_scc_vertex].value;
109     }
110     // dp find min for each scc
111     scc.WhitenAllVertices();
112     for (u = 0; u < scc.size; ++u)
113     {
114         if (scc.vertices[u].color == Color::kWhite)
115             DpSCCVisit(scc, u);
116     }
117     // set min attribute for each original vertex
118     for (SCCVertex& scc_v : scc.vertices)
119     {
120         for (int v : scc_v.vertices)
121         {
122             graph.vertices[v].min = scc_v.dp_min_scc_vertex;
123         }
124     }
125 }
```

Approach 2 Perform DFS on G^T by considering vertices in the increasing order of $L(u)$ in the out-most loop. The depth-first forest partition vertices with the same *min* into the same depth-first forest tree where *min* is the root vertex of the tree.

The root has the minimum L value due to the order we consider vertices. The root of the tree of G^T is reachable in G by each vertex in the tree of G^T. The main idea is $u \rightsquigarrow v$ in $G \iff v \rightsquigarrow u$ in G^T. Note that DFS will visit all reachable unvisited vertices. Hence considering vertices in the increasing order of $L(u)$ makes the algorithm work correctly.

```
1  void DFSVisit(Graph<TranposeVertex>& tranpose, int u,
2      Graph<OriginalVertex>& graph, int root)
3  {
4      tranpose.vertices[u].color = Color::kNonWhite;
5      graph.vertices[u].min = root;
6      for (int v : tranpose.adj[u])
7      {
8          if (tranpose.vertices[v].color == Color::kWhite)
9              DFSVisit(tranpose, v, graph, root);
10     }
11 }
12
```

```cpp
void Reachability(Graph<OriginalVertex>& graph)
{
    int u;
    std::vector<int> counting_sort(graph.size);
    // sort vertices in the increasing order of L value in linear time
    for (u = 0; u < graph.size; ++u)
    {
        counting_sort[graph.vertices[u].value] = u;
    }
    // find tranpose
    Graph<TranposeVertex> tranpose = Tranpose(graph);
    // DFS
    tranpose.WhitenAllVertices();
    for (int v : counting_sort)
    {
        if (tranpose.vertices[v].color == Color::kWhite)
            DFSVisit(tranpose, v, graph, v);
    }
}
```