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.

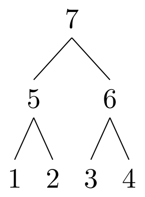
```
std::vector<int> OutDegree(const Graph& graph)
{
    size_t v_size, i;
    v_size = graph.adj.size();
    std::vector<int> degree(v_size);
    for (i = 0; i < v_size; ++i)
    {
        // assume graph.adj[i].size() takes O(n)
        // where n is size of  graph.adj[i]
        degree[i] = graph.adj[i].size();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

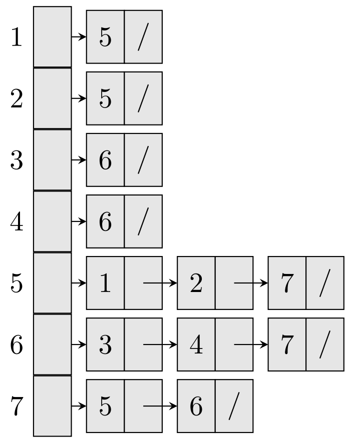
```
std::vector<int> InDegree(const Graph& graph)
{
    size_t v_size, i;
    v_size = graph.adj.size();
    std::vector<int> degree(v_size);
    for (i = 0; i < v_size; ++i)
    {
        for (int v : graph.adj[i])
        {
            ++degree[v];
        }
    }
    return degree;
}
```

22.1-2

Consider the following binary tree:



We have the following adjacency-list representation:



We have the following adjacency-matrix representation:

$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
6 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
\end{array}$$

22.1-3

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

```
AdjListGraph Transpose(const AdjListGraph& graph)
{
    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)
    {
        edge_usage = std::vector<bool>(size, false);
        edge_usage[u] = true;
        for (int v : graph.adj[u])
        {
            if (edge_usage[v] == false)
            {
                target.adj[u].push_back(v);
                edge_usage[v] = true;
            }
        }
    }
    return target;
}
```

22.1-5

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

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

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):

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

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;
    v = 0;
    while (u < size && v < size)
    {
        if (graph[u][v])
        {
            ++u;
        }
        else
        {
            ++v;
        }
    }
    if (u >= size)
        return -1;
    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;
}
```

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.

```cpp
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::kNonWhite;
    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 (int v_idx : graph.adj[u_idx])
        {
            Vertex& v = graph.vertices[v_idx];
            if (v.color == Color::kWhite)
            {
                v.color = Color::kNonWhite;
                v.distance = u.distance + 1;
                v.prev = u_idx;
                q.push(v_idx);
            }
        }
    }
    return discover_result;
}
```

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;
    }
```

```
        return discover_result;
}
```

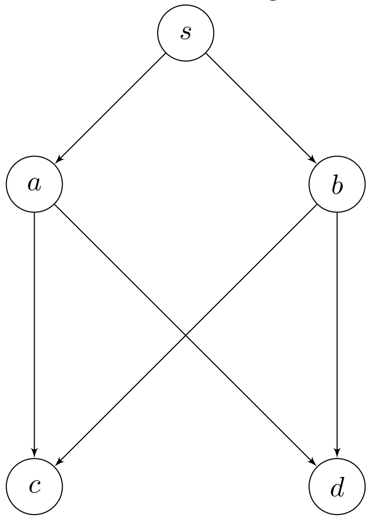
22.2-5

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.

```
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;
}
```

```cpp

int Diameter(Graph& graph)
{
    int u_idx, v_idx;
    std::queue<int> q;
    // let vertex 0 be the root node
    graph.vertices[0].depth = 0;
    graph.vertices[0].prev = -1;
    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])
        {
            if (v_idx != u.prev)
            {
                Vertex& v = graph.vertices[v_idx];
                v.depth = u.depth + 1;
                v.prev = u_idx;
                q.push(v_idx);
            }
        }
    }
    return DiameterAux(graph, 0);
}
```

22.2-9

```cpp
std::list< std::pair<int, int> > TraverseEdge(Graph& graph, int src_idx)
{
    int u_idx, v_idx;
    std::list< std::pair<int, int> > traverse_result;
    std::list< std::pair<int, int> >::iterator result_it;
    std::queue<int> q;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
    }
    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.

```
void DFSVisit(Graph& graph, int u_idx, int& time)
{
    Vertex& u = graph.vertices[u_idx];
    ++time;
    u.discovery_time = time;
    u.color = Color::kNonWhite;
    for (int v_idx : graph.adj[u_idx])
    {
        Vertex& v = graph.vertices[v_idx];
        if (v.color == Color::kWhite)
        {
            v.prev = u_idx;
            DFSVisit(graph, v_idx, time);
        }
    }
    ++time;
    u.finishing_time = time;
}

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

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 $v.d < v.f < u.d < u.f \Longrightarrow (u, v)$ is a cross edge.

We claim (u, v) is a cross edge $\Longrightarrow v.d < v.f < u.d < u.f$. 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. □

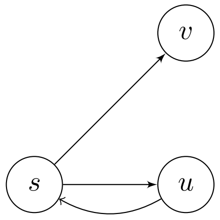
22.3-7

```
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;
```

```
                u.finishing_time = time;
            }
            else
            {
                v_idx = *v_it;
                Vertex& v = graph.vertices[v_idx];
                if (v.color == Color::kWhite)
                {
                    ++time;
                    v.discovery_time = time;
                    v.color = Color::kGray;
                    v.prev = u_idx;
                    path.push({ v_idx, graph.adj[v_idx].begin(), graph.adj[v_idx].end() });
                }
                ++v_it;
            }
        }
}
```

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:

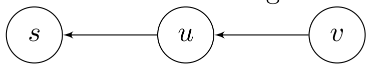
```cpp
struct Edge
{
    int vertex_from;
    int vertex_to;
    enum Type { kTree, kBack, kForward, kCross } type;

    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);
            }
            else if (v.color == Color::kGray)
            {
                result.emplace_back(u_idx, v_idx, Edge::kBack);
            }
        }
    }
    u.color = Color::kBlack;
    ++time;
    u.finishing_time = time;
}

std::list<Edge> DFSUndirected(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)
        {
            DFSVisitUndirected(graph, u_idx, time, result);
        }
    }
    return result;
}
```

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

```cpp
void DFSVisitUndirected(Graph& graph, int u_idx, int& time, int cc)
{
    Vertex& u = graph.vertices[u_idx];
    ++time;
    u.discovery_time = time;
    u.color = Color::kGray;
    u.connecting_component = cc;
    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;
                DFSVisitUndirected(graph, v_idx, time, cc);
            }
        }
    }
    u.color = Color::kBlack;
    ++time;
    u.finishing_time = time;
}

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

22.3-13

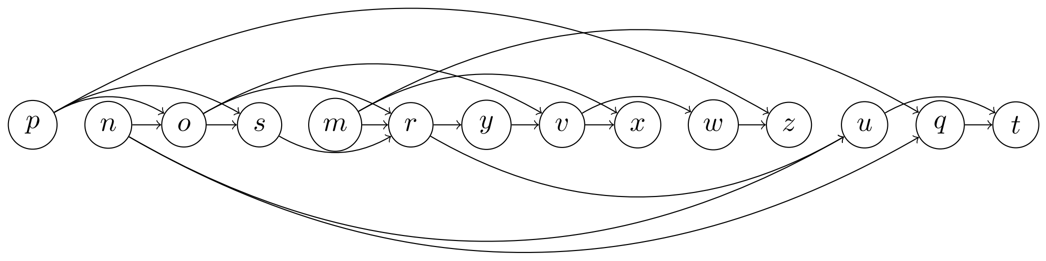
We can run DFS multiple times with source on each vertex on the graph. A directed graph is not signly connected if there exist some forward edges or cross edges that connect vertices in the same depth-first tree (i.e. in the same component). Hence, we have the following algorithm runs in $O(|V|(|V| +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

```
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 (v.color == Color::kWhite)
        {
            v.prev = u_idx;
            DFSVisit(graph, v_idx, time);
        }
        else if (v.color == Color::kBlack)
        {
            return false;
        }
    }
    u.color = Color::kBlack;
    ++time;
    u.finishing_time = time;
    return true;
}

bool DFSWithSrc(Graph& graph, int src_vertex)
{
    int time, u_idx;
    for (Vertex& u : graph.vertices)
    {
        u.color = Color::kWhite;
        u.prev = -1;
    }
    time = 0;
    return DFSVisit(graph, src_vertex, time);
}

bool CheckSinglyConnected(Graph& graph)
{
    int u_idx;
    for (u_idx = 0; u_idx < graph.size; ++u_idx)
    {
        if (DFSWithSrc(graph, u_idx) == false)
            return false;
    }
    return true;
}
```

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:

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

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

```
// graph must be DAG
// 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

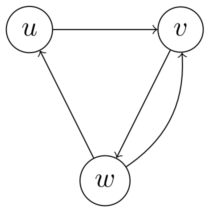
A forest contains multiple free trees (components). By Theorem B.2, $|E| = |V| - 1$ in a free tree. Hence, we can perform DFS on the graph. For each component, if the number of edges is more than the number of vertices, we can determine the graph contains a cycle. Hence, we can bound the running time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

```
            u.prev = -1;
        }
        time = 0;
        for (u_idx = 0; u_idx < graph.size; ++u_idx)
        {
            if (graph.vertices[u_idx].color == Color::kWhite)
            {
                if (DFSVisit(graph, u_idx, time) == false)
                {
                    return false;
                }
            }
        }
        return true;
    }
```

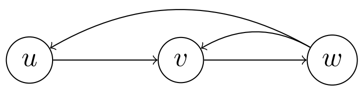
22.4-4

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:



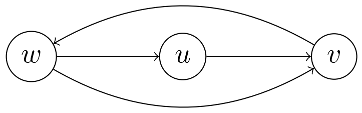
Let u be our source edge of DFS. Then we have the following DFS result and topological sort result which only has two "bad" edges: (w, u) and (w, v).

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])
        {
            --in_degree_counter[v];
            if (in_degree_counter[v] == 0)
            {
                zero_in_degree.push(v);
            }
        }
    }
    return topo_sort;
}
```

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.

Updating...