Chapter 22 Solusion

frc6.com
i@frc6.com
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;
}
```

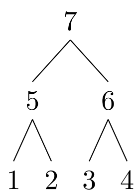
in-degree: $\Theta(V + E)$ by maintaining a counting table: each entry of the table is the counter for in-degree of the specific vertex.

```
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)
    {
```

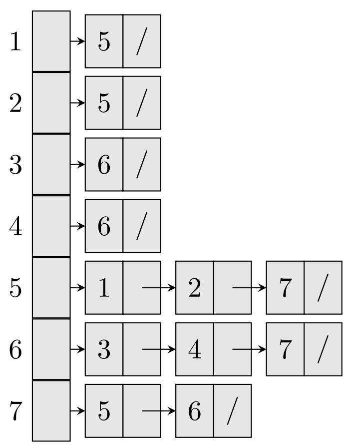
```
        for (int v : graph.adj[i])
        {
            ++degree[v];
        }
    }
    return degree;
}
```

22.1-2

Consider the following binary tree:



We have the following adjacency-list representation:



We have the following adjacency-matrix representation:

	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:

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

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 $v \in [1, n]$ $\square$

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.