Dayou Du
N13825075
dd2645@nyu.edu

Homework 8 - SA
Fundamental Algorithms

2017-11-22

Ex 7.1

a)

1. Consider any edge(u, v)
2. As the definition of edge direction, we discovers v belongs to $Adj[u]$ before we discovers u belongs to $Adj[v]$
3. Thus, when we find edge (u, v) for the first time (when we processing $Adj[u]$):
   (a) If u is an ancestor of v, then (u, v) will become an tree-edge
   (b) If u is an descendant of v, then as the procedure of BFS, we have discovered u when we process $Adj[v]$, which is contradict with precondition $2.$. **Impossible**
   (c) If u is nether a ancestor of v, nor a descendant of v, then (u, v) is a cross-edge.

Thus, the non-tree edges are all cross edges.
I got this question CORRECT.

b)

Consider the two situations $3.(a)$ and $3.(c)$ in question a.

- situation $3.(a)$: (u, v) is a tree edge, and the level of v is one more than the level of u
- situation $3.(c)$:
  1. level of v is smaller than level of u. Then we discovered v before we discovered u, contradict with edge direction. **Impossible**
  2. level of v is more than one levels deeper than u. Then we discovered the parent node of v later than we discover u, because the parent node of v's level is one or more than one levels deeper than u. So v will become as a child of u when we process $Adj[u]$. **Contradiction.**
  3. Thus, the level of v is the same of one more than the level of u.

I got this question CORRECT.

Ex 7.5

a)

$F(0) = 0$
$F(k) = k + F(k-1) + F(k-2) + .. + F(0) = k + \sum_{i=0}^{k-1} F(i), k > 0$
I got this question CORRECT.

b)

$F(k) - F(k-1) = 1 + F(k-1), k > 0$
$F(k) = 2F(k-1) + 1,\ k > 0$
$F(k) = 2^0 + 2^1 + 2^2 + ... + 2^{k-1} + 2^k F(0) = 2^k - 1, k > 0$
The formula above stands true for $F(0) = 0$, thus we have:
$F(k) = 2^k - 1, k \geq 0$
I got this question CORRECT.

c)

$G(k) = k - 1 + G(k-1), k > 0$
$G(0) = 0$
I got this question CORRECT.

d)

$G(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2}, n > 0$
The formula above stands true for $G(0) = 0$, thus we have:
$G(n) = \frac{n(n-1)}{2}, k \geq 0$
I got this question CORRECT.
Note: the question c in the solution is the questions c&d here

e)

Calling the $DFS(k)$ in this question is just like calling $WWKWWSolver(k)$: The procedure on parameter k will call the procedure with parameters in $[0..k-1]$.

That is, the first DFS version in question a is like the $WWKWWSolver$ without saving intermediate results: The “smaller” procedure will be executed more than one times.

The second DFS version in question c is like the $WWKWWSolver$ with saving intermediate results: Each “smaller” procedure will only executed once.
I got this question CORRECT.
Note: the question d in the solution is the questions e here

Ex 7.7

- If we use a pre-order mark and a post-order mark, the program can detect the cycle when we meet again with a vertex that has been pre-order entered but not post-order exited.
- If we use a pre-order mark and without a post-order mark, the program cannot detect cycle since it cannot distinguish between the points with more than one incoming edge or there's a loop.
- If we use a post-order mark and without a pre-order mark, the program will not stop when there's a loop.

I got this question CORRECT.

Ex 7.8

The program stopped, and the vertices on the cycle are not printed ($Blocked$ is not empty).
I got this question CORRECT.

Ex 7.10

a)

$u.pre - u.post$ = the number of vertices from $root$ to u(exclude u) $-$ the number of the descendants of u
The solution does NOT give the answer to this question. It only gives the answer to question b and c

b)

u is a descendant of $v \Leftrightarrow u.pre > v.pre$ and $u.post < v.post$
I got this question CORRECT.

c)

proof:
$\Rightarrow$:
Let u be a descendant of v, then

- u has not been pre-order touched by the time v is pre-order touched, thus $u.pre > v.pre$
- v has not been post-order exited by the time u is post-order exited, thus $u.post < v.post$

$\Leftarrow$:
Proof by contradiction: Assume that we have u, v where $u.pre > v.pre$ and $u.post < v.post$ and u is not a descendant of v.
Let r be the lowest common ancestor(LCA) of u and v

- If r is the same as v, then u is a descendant of v. **Contradict**
- If r is the same as u, then v is a descendant of u, $u.pre < v.pre$. **Contradict**
- If r is neither the same as u. nor the same as v (u and v are in difference subtrees rooted at different children of r). Since $u.pre > v.pre$, then the subtree for u is on the right of the subtree for v, which gives $u.post > v.post$. **Contradict**

Thus, given $u.pre > v.pre$ and $u.post < v.post$, u is a descendant of v.
I got this question CORRECT.

7.Xa

Algorithm 1 Ex 7.Xa

```
procedure BFSTopSort(G(V, E))
    Create array InDegree[V] where for v ∈ V, InDegree[v] is used to store the indegree of v
    Set all elements in InDegree as 0
    for v ∈ V do
        for u ∈ Adj[v] do
            InDegree[u] ← InDegree[u] + 1
        end for
    end for
    Create empty set Ready
    Create empty set Unready
    for v ∈ V do
        if InDegree[v] == 0 then
            Ready.insert(v)
        else
            Unready.insert(v)
        end if
    end for
    while Ready is not empty do
        v ← DeleteFrom(Ready)
        for u ∈ Adj[v] do
            InDegree[u] ← InDegree[u] − 1
            if InDegree[u] == 0 then
                Unready.delete(u)
                Ready.insert(u)
            end if
        end for
        Print(v)
    end while
    if Unready is not empty then
        Detected Loop!
    end if
end procedure
```

I got this question CORRECT.

7.Xb

Algorithm 2 Ex 7.Xb

```
procedure DFSTopSortDriver(G(V, E))
    Create a mark array/bitmap Stated[V] with all elements initialized to false/0
    Create a mark array/bitmap Done[V] with all elements initialized to true/1
    for v ∈ V do
        if Started[v] == 0 then
            DFSTopSort(v)
        end if
    end for
end procedure
procedure DFSTopSort(v)
    Stated[v] ← 1
    for u ∈ Adj[v] do
        if Started[u] == 0 then
            DFSTopSort(u)
        else
            if Done[u] == 0 then
                Loop Detected!
            end if
        end if
    end for
    Print(v)
    Done[v] ← 1
end procedure
```

I got this question CORRECT.

Ex 7.Y

Algorithm 3 Ex 7.Y

Input: a tree-liked graph $T(V, E)$
Output: a set of vertices that could be a $center$

```
function FindCenter(T(V, E))
    Create array Degree[V] where for v ∈ V, Degree[v] is used to store the degree of v
    Set all elements in Degree as 0
    for v ∈ V do
        for u ∈ Adj[v] do
            Degree[u] ← Degree[u] + 1
        end for
    end for
    Create empty set Ready,NextRound,Unready
    for v ∈ V do
        if Degree[v] == 1 then
            Ready.insert(v)
        else
            Unready.insert(v)
        end if
    end for
    while Unready is not empty do
        while Ready is not empty do
            v ← DeleteFrom(Ready)
            for u ∈ Adj[v] do
                Degree[u] ← Degree[u] − 1
                if Degree[u] == 1 then
                    NextRound.insert(u)
                    Unready.delete(u)
                end if
            end for
        end while
        Delete all elements in NextRound and put them into Ready
    end while
    return Ready                         ▷ all elements left in Ready can be a center
end function
```

I got this question CORRECT.

Ex 7.Z

a)

Algorithm 4 Ex 7.Za

procedure $DFSDriver(G(V, E))$
    for $v \in V$ **do**
        $v.longestoutof \leftarrow -1$
    end for
    for $v \in V$ **do**
        $DFS(v)$
    end for
end procedure
function $DFS(v)$
    if $v.longestoutof \neq -1$ **then**
        return $v.longestoutof$
    end if
    $v.longestoutof \leftarrow 0$
    for $u \in Adj[v]$ **do**
        $v.longestoutof \leftarrow max\{v.longestoutof, Ecost(u, v) + DFS(v)\}$
    end for
    return $v.longestoutof$
end function

I got this question CORRECT.
Note: In this solution I assumed that the edge cost can not be negative. Thus I initialized the $longestoutof$ fields as -1 as a mark of uncalculated (Perform the same function as the Nil in solution). And we have to specially deal with leaf nodes if the edge cost can be negative.

b)

Algorithm 5 Ex 7.Zb

```
procedure BFS(G(V, E))
    Create array InDegree[V] where for v ∈ V, InDegree[v] is used to store the indegree of v
    Set all elements in InDegree as 0
    for v ∈ V do
        v.longestinto ← 0
        for u ∈ Adj[v] do
            InDegree[u] ← InDegree[u] + 1
        end for
    end for
    Create empty set Ready
    Create empty set Unready
    for v ∈ V do
        if InDegree[v] == 0 then
            Ready.insert(v)
        else
            Unready.insert(v)
        end if
    end for
    while Ready is not empty do
        v ← DeleteFrom(Ready)
        for u ∈ Adj[v] do
            InDegree[u] ← InDegree[u] − 1
            u.longestinto ← max{u.longestinto, v.longestinto + Ecost(v, u)}
            if InDegree[u] == 0 then
                Unready.delete(u)
                Ready.insert(u)
            end if
        end for
        Print(v)
    end while
    if Unready is not empty then
        Detected Loop!
    end if
end procedure
```

I got this question CORRECT.

c)

By apply changes below to transform $G(V,E)$ to $G'(V,E')$, $Ecost(u,v)$ to $Ecost'(u,v)$ we can use the algorithm in question a) to solve question b).

- For each $(u,v) \in E$, add (v,u) into E' (reverse every edge)
- $Ecost'(u,v) \leftarrow Ecost(v,u)$

That is, to solve question b for graph $G(V,E)$ and cost funtion $Ecost(u,v)$, we can call
$SolverToQuestionA(G'(V,E'), Ecost'(u,v))$
and then the fields $u.longestoutof$ is the $u.longestinto$ we need in question b
I got this question CORRECT.
Note: It seems that the question 7.Zc&d in our handout is different with the solutions here.
There questions c&d are merged into one question d in the solution sheet. And the question c in the solution sheet did NOT appear on the homework handout.

Here is the answer to the question c described in the solution sheet:
Yes we can adapt them to get the shortest path. The basic idea is change the initial value to $+\infty$, and use min instead of max to get a "expand" with calculating the smallest path at the same time. Another thing that have to be adapt is separately deal with the "leaf nodes" (initialize them to 0, not $+\infty$)

d)

By making the exactly same transformation in question c.

- $G(V,E) \rightarrow G'(V,E')$: For each $(u,v) \in E$, add (v,u) into E' (reverse every edge)
- $Ecost'(u,v) \leftarrow Ecost(v,u)$

That is, to solve question a for graph $G(V,E)$ and cost function $Ecost(u,v)$, we can call
$SolverToQuestionB(G'(V,E'), Ecost'(u,v))$
and then the fields $u.longestinto$ is the $u.longestoutof$ we need in question a
I got this question CORRECT.