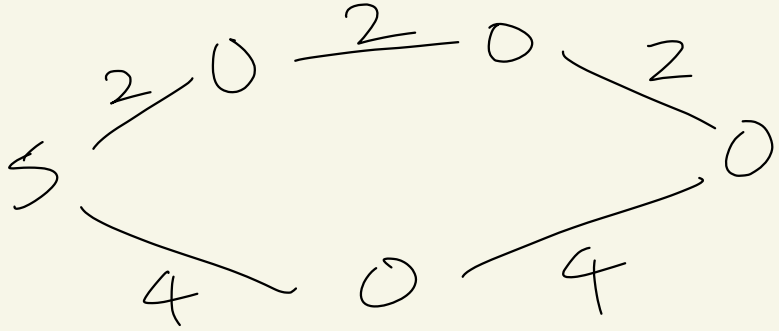
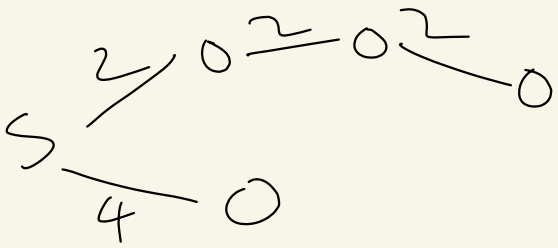
20190650 김관호 HW3

1. NO, here is counterexample.

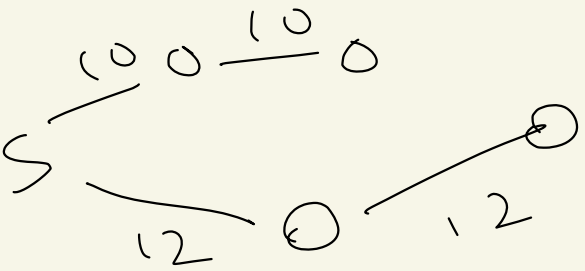


T is



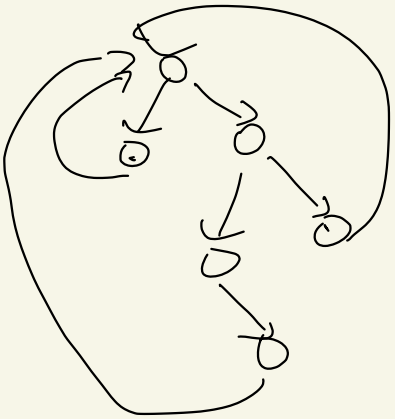
but when $c=8$ is added to all edge,

T' will be



Litterent from T.

2.



(a) If we use Dijkstra algorithm, we have to search all edges excep edge V to U in worst case.
Even more, we put all edge in our priority queue every single search.
so time complexity is $O(|E| + |E| \log |E|) = O(|E| \log |E|)$
where $|E|$ is the number of edges of new tree.

(b)
Since it's directed binary tree, there is unique path from U to V without making any cycle.
Then, there are two case of U, V.

- depth(U) < depth(V)
  In this case, we should just know simple path from U to V.
- depth(U) ≥ depth(V)
  In this case, the path shud be

$v \to root \to v$.

So, we need to know the shortest path

$v \to root$

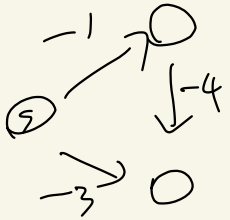
we can get that using BFS containing path length. Each time arriving at root node, update min_distance with BFS' path length. In that manner, we can get the shortest path from v to root.

And then use DFS from root or v to v since the path is simple.

So, the time complexity in worst case is have BFS and DFS further.

$O(|V|+|E|)$

3.



Since there is negative edge, we cannot use Dijkstra algorithm. Even it all edges are negative and say there's no cycle, Dijkstra will spend enormous time. Above graph is an example, if we find shortest node in that time, it doesn't guarantee that is shortest path to that node because of existence of negative edges, so we choose another algorithm, It was shown in Lecture Note.

Let V, E are the set of nodes, edges of G

```
for all v ∈ V
    dist(v) = ∞
dist(s) = 0 , cycle_nodes = [ ]
for i = 0 to i = |V| - 1 do          ... |V| times
    for all (u,v) ∈ E do             ... |E| times
        if dist(v) > dist(u) + length(u,v) then
            if i = |V| - 1 then cycle occured
                cycle_nodes.append(v)
            continue
        dist(v) = dist(u) + length(u,v)
```

check negative cycle's node

Continued..

for all node in cycle_nodes do
    dist(node) = -∞

- correctness

By doing that, we find shortest path cost for valid nodes. And if cycle occured, then put -∞ for all nodes which are connected from the cycle. And unreachable node, they are +∞ because we initally put ∞ for all nodes.

- Analysis.

For first 2-for loop algorithm goes $|V| \cdot |E|$ times, and final for loop is going to be at most $|V|$ times.

So, $O(|V| \cdot |E|)$

4.

We can design greedy algorithm with BFS.

Let's say color's name as their index (= 0, 1, ⋯, d)

At first, choose any node to start. Color that node to 0.

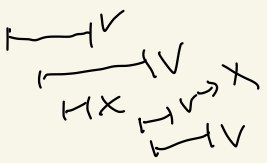
And start BFS from start node.

while BFS, to color node, check which color is used from its neighbors and choose smallest color that not in neighbors color list. ↳ greedy method

- It is correct because we have d+1 colors. If there's a node which has d neighbor (maximum neighbors in graph), and their colors are all different from each other. In that worst case, there is a color not used between 0 and d because number of color used is d. So it guarantees that we can color all node satisfying condition with d+1 different colors.
- while BFS, we check neighbors one more time to choose current node's color. So, |V|+2·|E| time occured

time complexity is $O(|V|+|E|)$ using adjacency list.

5.

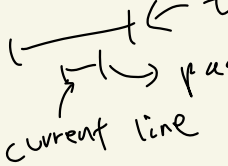


To solve this problem, firstly we need to sort set S' lines by its start point ascending order.

Let's use stack to contain line which will be in set T. Follow rules below.

1. If current line is overlapped by line top of stack, pass that line.

=>



2. Define variable cur_end which contain end point of block of lines.

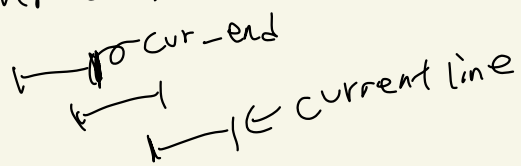
And If current line's end point is bigger than end point of line that is top of the stack.

In that case, two cases begin.

2-1. when current line's start point is bigger than cur_end.

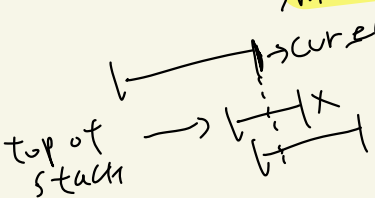
↓↓ example

which is the case below



In this case, push current line in stack and update cur_end with current line's end point

2-2. when current line's start point is smaller or equal than cur_end.



In this case, the line top of stack is overlapped back and forth. so we should pop that line from the stack. And cur_end will be the same.

Finally, lines in the stack are elements of the set T.

- correctness

Since we sort lines by its start point next line's start point is always greater or equal than previous one.
All we need to do is check its end point to check overlap
If overlapped, we skip that.
If not, we'll check if latest line is whether overlapped by lines back and forth.
If latest line is overlapped, remove.
If latest line is not overlapped, put current line in the set and keep going.
In this manner, we can creat smallest subset T we desired.

- Time complexity

Sort => $n \log n$
Sweeping => $+ n$
∴ time complexity is $O(n \log n)$

6.

Similarly #5, we need to sort lines and sweep.

Firstly sort lines by its start point (= L[i]).
And then, choose first line's start point for cur_stab
and set it's end point (= R[i]) to end_range.
Below is pseudo code.

```
stubs = []
end_range = first line's end point, cur_stab = first line's
                                              start point
for next n-1 lines do:
    if line's start point > end_range then
        stubs.append(cur_stab)
        end_range = line's end point
    else
        end_range = min(end_range, line's end point)
    cur_stab = line's start point
stubs.append(cur_stab)   <- for last element
```

Finally, list stubs is smallest set of points that stabs X.

- correctness
  Since we sorted lines its start point, following line's start point must belong to previous line or exceed shortest end point of previous lines.

In tormer case, Update cur-stub to new one and update end_range.
In latter case, select cur-stub to stub point and update cur-stub, end_range.
This is valid because following line's start point must greater than eqal to tormer one and all we have to do is check whether start point of current line exceed the shortest end point of current lines' group.

- Time analysis
  Sort: $n \log n$
  Sweep: n
  $+$
  $\Rightarrow$ time complexity is $O(n \log n)$