COL351 Assignment 2

Harshit Mawandia, Tanish Tuteja

September 2022

Q1

Algorithm:
Let D be the disk with n sectors $\{d_1, d_2, ..., d_n\}$ with d_i storing value p_i. Now lets consider an array P of size n which stores values as $P = [p_1, p_2, ..., p_n]$

```
function KADANE'S-ALGORITHM(Arr)
    $Max \leftarrow 0$
    $MaxEndingAtI \leftarrow 0$
    $i \leftarrow 0$
    while $i < Arr.length$ do
        $MaxEndingAtI \leftarrow MaxEndingAtI + Arr[i]$
        if $Max < MaxEndingAtI$ then
            $Max \leftarrow MaxEndingAtI$
        end if
        if MaxEndingAtI¡0 then
            $MaxEndingAtI \leftarrow 0$
        end if
        $i \leftarrow i + 1$
    end while
    $return(Max)$
end function

$MaxLinear \leftarrow$ KADANE'S-ALGORITHM($P$)
$i \leftarrow 0$
$sum = 0$
while $i < P.length$ do
    $sum \leftarrow sum + P[i]$
    $P[i] = -P[i]$
    $i \leftarrow i + 1$
end while
$MinLinear \leftarrow -$KADANE'S-ALGORITHM($P$)
$Ans = max(MaxLinear, sum - minLinear)$
```

Proof of Correctness:

We have used the $Kadane's Algorithm$ to calculate the maximum sub-array sum of a linear array in $O(N)$ time. This algorithm takes the sum of consecutive elements until it becomes negative, when it is reinitialised to 0 and updates the overall maximum whenever applicable. The maximum sum of an array will always be greater than 0, so when the contiguous sum becomes negative it is reinitialised to 0.

Case 1:
When the maximum sum of the circular array is just a linear subarray. We directly use the $Kadane's\ Algorithm$ to calculate the maximum subarray sum. Let this be S_1
Case 2:
Now, since we have to calculate the maximum contiguous sum of a circular array, there can also be the case where there is a contiguous part of array, $p_i ... p_k ... p_j$, where $i < k < j$, which we do not include in our sum. In this case, $\sum_{i=0}^{j} p_i$ has to be minimum for some i, j. To calculate this minimum sum, we change each p_i to $-p_i$ and then calculate the maximum contiguous sum of P. The maximum returned now is essentially the negative of the minimum contiguous sum of the array. So, if we subtract this part from the sum of all elements, we get the sum of elements $p_{j+1} ..., p_n, p_1 ... p_{i-1}$. Let this be S_2

Now the overall maximum subarray sum S would be the maximum of these two cases:

$$S = max(S_1, S - 2)$$

Time Complexity:
The $Kadane's\, algorithm$ takes $O(N)$ time, which we have used twice. We have also changed every p_i to $-p_i$, which again takes $O(N)$ time. So the overall time complexity:

$$T(N) = 2 * O(N) + O(N)$$

$$\Rightarrow O(N)$$

Q2

(a) We need to find a cycle $(c_{i_1}, c_{i_2}, ..., c_{i_k}, c_{i_{k+1}} = c_{i_1})$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3]...R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

Let $P = R[i_1, i_2] \cdot R[i_2, i_3]...R[i_{k-1}, i_k] \cdot R[i_k, i_1]$

$$\Rightarrow P > 1$$

$$\Rightarrow 1/P < 1$$

$$\Rightarrow log(1/P) < 0$$

$$\Rightarrow log(1/R[i_1, i_2]) + log(1/R[i_2, i_3]) + ... + log(R[i_{k-1}, i_k]) + log(R[i_k, i_1]) < 0$$

Thus, consider a graph $G = (C, E)$, where the vertices $c_i \in C$ correspond to the currencies, and the edges (i, j) are such that $wt(e = (i, j)) = log(1/R[i, j])$
Therefore, we need to find a cycle in G such that the sum of weights is negative.
Now, the algorithm to detect a negative edge sum cycle is as follows:

Algorithm

```
function BF-CHECK(G = (V, E), v, D, P)
    $D[v] \leftarrow 0$
    $P[v] \leftarrow NULL$
    $SingleSourceBF(G, v, D, P)$
    for all $e = (x, y) \in E$ do
        if $D[y] > D[x] + wt(e)$ then
            return True, y
        end if
    end for
    return False, NULL
end function
function NEGCYCLE(G = (V, E), Parents)
    for all $v \in V$ do
        $D[v] \leftarrow \infty$
        $V[v] \leftarrow False$
    end for
    for all $v \in V$ do
        if $V[v] = False$ then
            $P \leftarrow []$
            $hasCycle, cycleNode \leftarrow$ BF-CHECK$(G, v, D, P)$
            if $hasCycle = True$ then
                $Parents \leftarrow P$
                return True, cycleNode
            end if
            for all $v \in V$ do
```

```
                if $D[v] < \infty$ then
                    $V[v] \leftarrow True$
                end if
            end for
        end if
    end for
    return False
end function
$Parents \leftarrow NULL$
$ans, cycleNode \leftarrow$ NegCycle($G, Parents$)
```

Proof of Correctness

We need to prove that there exists a negative weight cycle if and only if there exists a vertex $v \in V$ such that it is possible to improve the minimum distance from v to some other vertex $u \in V$ after n iterations of Bellman-Ford algorithm.

First we prove the $\Rightarrow$ part.
Let D be the distances computed using Bellman-Ford algorithm from a vertex $v \in V$ such that the negative weight cycle is reachable from v. Such a v exists since the cycle will be reachable at least from any of the cycle vertices.
Now, if possible, assume it is not possible to improve D for any vertex. Let $(u_1, u_2, ..., u_m)$ be the vertices of the cycle. Then,

$$D(u_i) + wt(u_i, u_{i+1}) \geq D(u_{i+1}), 1 \leq i < m$$

Also,

$$D(u_m) + wt(u_m, u_1) \geq D(u_1)$$

$$\Rightarrow \sum_{i=1}^{m-1} D(u_i) + \sum_{i=1}^{m-1} wt(u_i, u_{i+1}) + D(u_m) + wt(u_m, u_1) \geq \sum_{i=1}^{m} D(u_m)$$

$$\Rightarrow \sum_{i=1}^{m-1} wt(u_i, u_{i+1}) + wt(u_m, u_1) \geq 0$$

However, this is a contradiction to the fact that the sum of weights of the cycle is strictly negative. Hence, proved it is possible to improve D for some vertex.

Now we prove the $\Leftarrow$ part.
Again, if possible, assume there exists no negative weight cycle. Then, from lecture 15, after i^{th} iteration of Bellman-Ford, $D[v]$ will be optimal $\forall v \in V, level(v) \leq i$. Also, we know the maximum level of a vertex can be the total number of vertices, i.e., n. Thus, after n iterations of Bellman-Ford, D will be optimal for all the vertices.
However this contradicts the fact that $\exists$ vertex $u \in V$ such that the $D(u)$ can be improved. Hence, proved that there exists a negative weight cycle.

Time Complexity

Let S_i be the set of vertices reachable from v_i such that $v \in S_i$ if $v \notin S_j, j < i$. Thus,

$$\sum_{i=1}^{n} |S_i| = n$$

Now, running Bellman-Ford on any v_i would take $O(|S_i|^3)$ time. Thus, total time complexity

$$T = \sum_{i=1}^{n} |S_i|^3$$

$$\leq (\sum_{i=1}^{n} |S_i|)^3$$

$$= O(n^3)$$

(b) We use the parents array computed in part (a) to find the negative weight cycle, if it exists.

Algorithm

```
$cycleNode, Parents \leftarrow$ CHECKFORCYCLE($G$)
$n \leftarrow Parents[cycleNode]$
$Cycle \leftarrow \{cycleNode\}$
while $n \neq cycleNode$ do
    $Cycle \leftarrow Cycle \cup \{n\}$
    $n \leftarrow Parents[n]$
end while
```

Proof of Correctness

Note that on successively taking the parent of a node as calculated with Bellman-Ford, 2 cases arise: either the node appears again or NULL appears, indicating we have reached the parent node.
In the first case, note that the depth of the node can be atmost n. Since after i^{th} iteration the distances of nodes at depth i are optimal, the depth of the node must be optimal. However, this contradicts the fact that we can make an improvement in the distance of the node. Thus, this case is not possible.
Therefore, the node must appear again in the parent array, thus the algorithm computes the cycle correctly.

Time Complexity

After the algorithm from part (a), we are only traversing over the parents array once, thus the total time

$$T = O(n) + O(n^3) = O(n^3)$$

Q3

Algorithm

```
function CheckPaths($S, E$)                                    ▷ $O(nk)$
    $count \leftarrow 0$
    for all $s \in S$ do
        $canAdd \leftarrow True$
        for all $m \in s$ do
            if $m \notin E$ then
                $canAdd \leftarrow False$
                break
            end if
        end for
        if $canAdd = True$ then
            $E \leftarrow E \setminus s$
            $count \leftarrow 1$
            break
        end if
    end for
    return $count, E$
end function

function Disjoint-Paths($node, S$)
    if $node = NULL$ then
        return $0, \emptyset$
    end if
    $opt_l, E_l =$ Disjoint-Paths($node.left, S$)
    $opt_r, E_r =$ Disjoint-Paths($node.right, S$)
    if $node.left \neq NULL$ then
        $S_l.insert((node, node.left))$
    end if
    if $node.right \neq NULL$ then
        $S_r.insert((node, node.right))$
    end if
    $leftPoss, E_l \leftarrow$ CheckPaths($S, E_l$)                       ▷ $O(nk)$
    $rightPoss, E_r \leftarrow$ CheckPaths($S, E_r$)                      ▷ $O(nk)$
    $X = E_l \cup E_r$                                                     ▷ $O(n)$
    $combPoss, X \leftarrow$ CheckPaths($S, X$)                           ▷ $O(nk)$
    $total \leftarrow leftPoss + rightPoss + combPoss + opt_l + opt_r$
    return $total, X$
end function

$MaxPaths \leftarrow$ Disjoint-Paths($T.root, S$)
```

Proof of Correctness

We will prove the algorithm by induction, inducting on the depth. Consider a node v. w.l.o.g Assume both left and right child of v exist.
By I.H., the algorithm gives the correct answer for subtrees rooted at $v.left$ and $v.right$. Now, consider the subtree rooted at v. Let $opt(v)$ be the optimal solution for the subtree rooted at v.

Now, if $\exists$ path $p \in opt(v)$ such that p overlaps with some other $q \in opt(v.left) \cup opt(v.right)$. Thus, we would have to replace q with p, and thus there is no improvement over $|opt(v.left)| \cup |opt(v.right|)$

Thus, for an improvement, we need to consider paths non overlapping with $opt(v.left)$ and $opt(v.right)$. Moreover, since each improving path must contain at least one of $(v, v.left), (v, v.right)$, we can at most improve by 2.

We can improve by 2 if $\exists p, q \in S$ such that p and q are in left and right subtrees of v, and p does not overlap with $opt(v.left)$ and q does not overlap with $opt(v.right)$.
We can improve by 1 if $\exists p \in S$ such that p is in subtree of v, and p does not overlap with $opt(v.left) \cup opt(v.right)$ Otherwise, we cannot improve the $|opt(v.left)| + |opt(v.right)|$ solution.
In this way, from the optimal solutions of the two subtrees, we can calculate the optimal solution for the tree.

Time Complexity

We have two functions $CheckPaths$ and $Disjoint - Paths$.
Lets say $|S| = k$. Lets say the number of nodes in the Tree $T = n$. We can say that the maximum length of a path would be $n - 1$.
Now for $CheckPaths$ function we iterate over the set S and for each edge of the elements in S. So the Time Complexity T_{CP} would be :

$$T_{CP} = O(k) \cdot O(n)$$
$$\Rightarrow T_{CP} = O(kn)$$

The $Disjoint - Paths$ function recursively calls itself on all the children of the current node. So it is called n number of times where n is the total number of nodes in the three. In this function we call the $CheckPaths$ function and the $Set - Union$ function, which are of order of time $O(nk)$ and $O(n)$ respectively.
So the overall Time Complexity T woudl be:

$$T(node) = T(node.left) + T(node.right) + O(nk) + O(n)$$
$$\Rightarrow T(root) = O(n) \cdot (O(nk) + O(n))$$
$$\Rightarrow T(root) = O(n) \cdot O(nk)$$
$$\Rightarrow T(root0 = O(n^2k)$$