COL351 Assignment 2

Vishwas Kalani, Viraj Agashe

TOTAL POINTS

57 / 60

QUESTION 1

1 Maximum sum 18 / 18

+ **8 pts** Algorithm Correctness
+ **8 pts** Correctness Proof
+ **2 pts** Time Complexity Analysis
+ **4 pts** Algorithm Partial Correct
+ **0 pts** Incorrect Algorithm
+ **3 pts** Partial Correct
+ **4 pts** Correctness Proof Partial Correct
✓ + **18 pts** **Correct Solution**

QUESTION 2

Forex trading 17 pts

2.1 part 1 10 / 10

✓ + **6 pts** **Algorithm Correctness**
✓ + **3 pts** **Correctness Proof**
✓ + **1 pts** **Running Time Analysis Correctness**
+ **5 pts** Algorithm partial
+ **2 pts** Correctness Proof Partial
+ **0 pts** Incorrect Algorithm

2.2 part 2 7 / 7

✓ + **4 pts** **Algorithm Correctness**
✓ + **2 pts** **Algorithm Correctness Proof**
✓ + **1 pts** **Time Analysis**
+ **0 pts** Incorrect Algorithm/Algorithm not given

QUESTION 3

3 Disjoint Paths 22 / 25

+ **2 pts** Identifying we need to use DP
+ **3 pts** Correctly defining the DP variables
+ **10 pts** Correctly defining the recursive(DP) equations
✓ + **15 pts** **Choosing the paths greedily in non increasing order of depth of LCA of the endpoints**
✓ + **5 pts** **Proof of Correctness**
✓ + **5 pts** **Proving $$O(poly(n))$$ Time Complexity**
+ **0 pts** Incorrect/Unattempted
- **3** **Point adjustment**

1 For fixed v, there can be multiple options for OPT(v.left) and OPT(v.right), and for some of them, p may overlap and for some p may not. So we cant use this argument.

1 Maximum Sum

1.1 Algorithm

We propose the following algorithm to find the contiguous sectors with maximum sum of a disc $D = (d_1, d_2, ...d_n)$ storing integers $p_1, p_2, ...p_n$:

1. Construct an array $A = [p_1, p_2, ...p_n]$.
2. Find the maximum sum subarray of the array A using the following dynamic programming algorithm:
   - Initialize a dynamic programming (DP) array `T[]`. Each $T[i]$ represents the maximum sum subarray ending at index i.
   - Set $T[1] = p_1$.
   - For $i \in [2, n]$, fill the DP array using the following relation:
   $$T[i] = \max(A[i], A[i] + T[i-1])$$
   - Finally, $\max(T)$ is the sum of the maximum sum subarray of T.
   - Suppose $T[j]$ is the maximum. We can recover the set of contiguous sectors by finding the first $i < j$ such that $T[i] < 0$ and $T[k] > 0\ \forall k \in (i, j)$. Then $A[i+1], ...A[j]$ is the required subarray (call this subarray D_1)
3. Consider the array $B = [p_2, ...p_n]$. Find the minimum sum subarray of B by using an algorithm similar to the one above:
   - Initialize a dynamic programming (DP) array `S[]`. Each $S[i]$ represents the minimum sum subarray ending at index i.
   - Set $S[1] = B[1]$.
   - For $i \in [2, n-1]$, fill the DP array using the following relation:
   $$S[i] = \min(B[i], B[i] + S[i-1])$$
   - Finally, $\min(S)$ is the sum of the minimum sum subarray of S.
   - Suppose $S[j]$ is the minimum. We can recover the set of contiguous sectors by finding the first $i < j$ such that $S[i] > 0$ and $S[k] < 0\ \forall k \in (i, j)$. Then $B[i+1], ...B[j]$ is the required subarray. Call this subarray D_2.
4. Compute the sum of all elements of the disc D, i.e. $\sum p = p_1 + p_2 + ... + p_n$, and
   $$s^* = \sum p - \min(S)$$
5. If $s^* \leq max(T)$ then D_1 is the required set of contiguous sectors, otherwise, $D \setminus D_2$ is the required set.

1.2 Time Complexity Analysis

We claim that the above algorithm is $O(n)$. We argue this as follows:

- Construction of the arrays A and B take $O(n)$ time respectively.
- **Finding the Max Sum Subarray of A:** The size of the DP table T is n, and since we have already computed $T[i-1]$, we can compute each $T[i]$ in $O(1)$ time. So filling the table takes $O(n)$ time.

- After the computation of the table is done, we can find its maximum in $O(n)$ time, and for this particular index j, we traverse the array to find the first $i < j$ such that $T[i] < 0$ and $T[k] > 0 \; \forall k \in (i, j)$. We may need to traverse the entire array for this, so this step is overall $O(n)$.
- **Finding the Min Sum Subarray of B:** Similarly to above, the size of the DP table S is n, and we can compute $S[i]$ in $O(1)$ time, and filling the table takes $O(n)$ time.
- After the computation of the table S is done, we can find its minimum in $O(n)$ time, and for this particular index j, we traverse the array to find the first $i < j$ such that $S[i] > 0$ and $S[k] < 0 \; \forall k \in (i, j)$. We may need to traverse the entire array for this, so this step is overall $O(n)$.
- Computing the sum of the disc D takes $O(n)$ time. For the final step, we perform some $O(1)$ operations. Under the condition $s^* < max(T)$, we may need to compute $D \setminus D_2$ which is also an $O(n)$ operation.

Thus the overall running time of the algorithm is $O(n)$.

1.3 Proof of Correctness

To argue the correctness of the algorithm, first let us consider the sector d_1 of the disc. There are two possibilities for the set of maximum sum contiguous sectors $\mathcal{S}$:

1. $\mathcal{S} = (d_i, ...d_j)$, where $1 \leq i \leq j \leq n$.
2. $\mathcal{S} = (d_i, ...d_1, ...d_j)$, where $1 \leq j < i \leq n$.

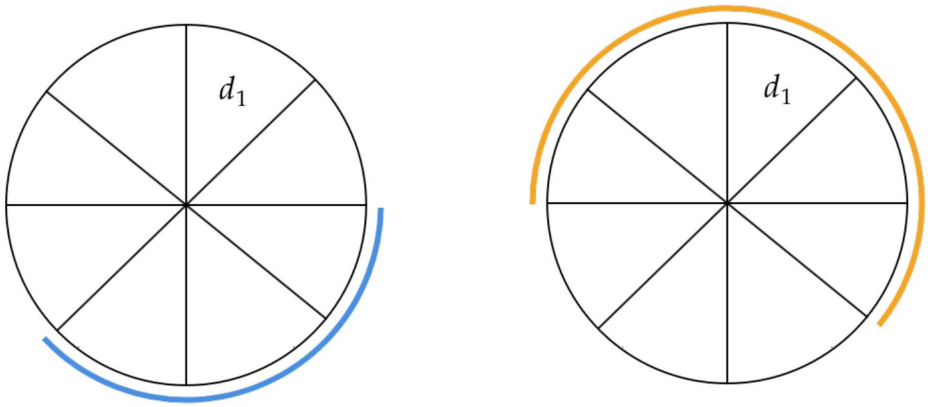

Figure 1: The possibilities for the max sum contiguous sectors

We compute the set of maximum sum contiguous sectors for the above cases, and take the maximum of both. Since these cases are exhaustive, we must get the correct answer. What remains is to prove that the answers for the two cases above have been computed correctly.

1.3.1 Case I

Note that in case $\mathcal{S}$ is of the form $(d_i, ...d_j)$, where $1 \leq i \leq j \leq n$, the max sum contiguous subsectors are the same as the max sum subarray of $A = [p_1, ...p_n]$, since in this case we do not need to consider the "wraparound" of the array. We will prove that our algorithm for computing the max sum subarray of an array A is correct.

Theorem 1. *Each entry of the table $T[i]$ correctly stores the sum of the subarray with maximum sum ending at that index i.*

Proof. Let us proceed to prove by induction. Note that the base case is correct, since $T[1] = A[1]$, and it is the only subarray ending at index 1, and hence must be the maximum sum subarray ending at 1. Suppose that T correctly stores the sum of the subarray with maximum sum ending at that index i for some k. Now, consider the subarray with maximum sum ending at index $k+1$. There are two possibilities:

- It contains only a single element $A[k+1]$
- It is appended to the subarray with maximum sum ending at index k.

Thus, the sum stored in $T[k+1]$ must be the $\max(A[k+1], T[k] + A[k+1])$, which is precisely what the algorithm sets it to. Since $T[k]$ has been computed correctly by the inductive hypothesis, by the principle of mathematical induction, the array $T[\,]$ is computed correctly. ■

Note that any subarray of A must end at some index $i \in [1, n]$. Therefore, taking the maximum of $T[\,]$ yields the sum of the maximum sum subarray of A. Further, note that we choose to append $A[k+1]$ to the max subarray ending at k when $A[k+1] < A[k+1] + T[k]$, i.e. $T[k] > 0$.

Therefore, supposing the maximum of T is at index j. Then, we can obtain explicitly obtain the max sum subarray by finding the first $i < j$ such that $T[i] < 0$ and $T[k] > 0\ \forall k \in (i, j)$. Then $A[i+1], ...A[j]$ is the required subarray. Thus, the algorithm is correct in this case, and D_1 is computed correctly. □

1.3.2 Case II

In the case of $\mathcal{S} = (d_i, ...d_1, ...d_j)$, where $1 \leq j < i \leq n$, we make the following key observation:

$$\sum_{p_1 \in \mathcal{S}} p_1 + \sum_{p_2 \in D \setminus \mathcal{S}} p_2 = \sum_{p \in D} p$$

Since $\sum_{p \in D} p$ is a constant, therefore, when we maximize $\sum_{p_1 \in \mathcal{S}} p_1$, we *minimize* $\sum_{p_2 \in D \setminus \mathcal{S}} p_2$. Since we know that in this case, $d_1 \in \mathcal{S}$, we find the minimum sum subarray of the rest of the disc, i.e. $B = [p_2, ...p_n]$, which will give us $\min\left(\sum_{p_2 \in D \setminus \mathcal{S}} p_2\right)$, using which we can find the required max sum subarray $\mathcal{S}$. We proceed similarly to case I, to show that our computation of the min sum subarray of B is correct.

Theorem 2. *Each entry of the table $S[i]$ correctly stores the sum of the subarray with minimum sum ending at that index i.*

Proof. Let us proceed to prove by induction. Note that the base case is correct, since $S[1] = B[1]$, and it is the only subarray ending at index 1, and hence must be the minimum sum subarray ending at 1. Suppose that S correctly stores the sum of the subarray with minimum sum ending at that index i for some k. Now, consider the subarray with minimum sum ending at index $k+1$. There are two possibilities:

- It contains only a single element $B[k+1]$
- It is appended to the subarray with minimum sum ending at index k.

Thus, the sum stored in $S[k+1]$ must be the $\min(B[k+1], S[k] + B[k+1])$, which is precisely what the algorithm sets it to. Since $S[k]$ has been computed correctly by the inductive hypothesis, by the principle of mathematical induction, the array $T[\,]$ is computed correctly. ■

Note that any subarray of B must end at some index $i \in [1, n-1]$. Therefore, taking the minimum of $S[\,]$ yields the sum of the minimum sum subarray of B. Further, note that we choose to append $B[k+1]$ to the max subarray ending at k only when $B[k+1] > B[k+1] + S[k]$, i.e. $S[k] < 0$.

Therefore, supposing the minimum of S is at index j. Then, we can obtain explicitly obtain the min sum subarray by finding the first $i < j$ such that $S[i] > 0$ and $S[k] < 0\ \forall k \in (i, j)$, and thus $B[i+1], ...B[j]$

is the required subarray. Thus, D_2 is computed correctly, and the algorithm is correct. $\square$

We have individually verified that D_1 and D_2 are correct. Further by our observation, D_2 being the subarray with minimum sum of B implies that $\mathcal{D} \setminus D_2$ is the set of contiguous sectors with maximum sum containing d_1. We simply select the set which has the maximum sum out of both these. Since the two cases are exhaustive, the algorithm computes the correct solution and we obtain the set of contiguous sectors with maximum sum. ■

1 Maximum sum **18 / 18**

+ **8 pts** Algorithm Correctness
+ **8 pts** Correctness Proof
+ **2 pts** Time Complexity Analysis
+ **4 pts** Algorithm Partial Correct
+ **0 pts** Incorrect Algorithm
+ **3 pts** Partial Correct
+ **4 pts** Correctness Proof Partial Correct

✓ **+ 18 pts Correct Solution**

2 Forex Trading

2.1 Detection of Positive Gain Cycle

2.1.1 Algorithm

We propose the following algorithm for the detection of a positive gain cycle in the weighted network $G = (V, E, R)$, where $R(i, j) = wt(i, j)$:

1. Construct an auxillary graph $G^* = (V, E, R^*)$, where the updated weights are related to the old weights as:
$$R^*(i, j) = \log\left(\frac{1}{R(i, j)}\right)$$
2. Starting from any currency c_i (since the graph is assumed to be strongly connected), compute $n = |V|$ iterations of the Bellman-Ford algorithm, and let D be the vector of distances. If there is an updation in D in the nth iteration, then there exists a positive gain cycle.
3. Otherwise, no positive gain cycle exists.

2.1.2 Pseudocode

```
function  checkNegCycle():
    For each v in V:
        D[v]= infinity
    D[c] = 0
    For i = 1 to n:
        For each (x,y) in E:
            If (i < n):
                If (D[y] > D[x] + wt(x,y)) then
                    D[y] = D[x] + wt(x,y)
            Else:
                If (D[y] > D[x] + wt(x,y)) then
                    return True
    return False
```

2.1.3 Time Complexity Analysis

We claim that this algorithm has a running time of $O(mn)$, where $m = |E|$ and $n = |V|$. We argue this as follows:

- Construction of the auxillary graph G^* simply requires copying the original graph and changing the weights of the edges on the original graph. This takes time proportional to the size of the graph, i.e. $O(m + n)$.
- Each iteration of the Bellman-Ford algorithm takes $O(m)$ time. During the nth iteration, we only perform some extra $O(1)$ checks. Thus, performing n iterations takes $O(mn)$ time.

Thus, the overall running time of the algorithm is $O(m + n + mn) = O(mn)$.

2.1.4 Proof of Correctness

First, note that we are required to find a cycle which satisfies,

$$R(i_1, i_2) \times R(i_2, i_3) \times ... \times R(i_k, i_1) > 1$$

This is equivalent to saying,

$$\log\left(\frac{1}{R(i_1,i_2)}\right)+\log\left(\frac{1}{R(i_2,i_3)}\right)+...+\log\left(\frac{1}{R(i_k,i_1)}\right)<0$$

This follows by the monotonicity of the logarithm, as well as the following properties:

- $\log(xy)=\log(x)+\log(y)$
- $\log(1/x)=-\log(x)$

Since in our auxiliary graph G^* the edge weights are $R^*(i,j)=\log\left(\frac{1}{R(i,j)}\right)$, checking for this condition is equivalent to finding a negative weight sum cycle in G^*, for which we use the Bellman-Ford algorithm. To prove correctness of our algorithm, we prove the following theorem:

Theorem 3. *A connected graph G has a negative weight cycle if and only if there is an update to the distance vector D in the n-th iteration of the Bellman-Ford Algorithm.*

Proof. [$\Rightarrow$] Suppose there is an update to the distance vector D in the nth iteration of Bellman-Ford. Let the vertex for which the distance was updated be v^*. Suppose on the contrary that there is no negative weight cycle in the Graph G. Note that in this case, we have proved in class that execution of i iterations of Bellman-Ford ensures that D is correct upto level i. Further, we know that the maximum depth (level) of a vertex in a tree with n vertices is $n-1$, and so $Level(v^*)\in[0,n-1]$. By definition of $Level(v)$, we must have that $D[v^*]=dist(c,v,G)$ (shortest distance), however, since $D[v^*]$ is updated in the nth iteration, the new distance $D'[v^*]<D[v^*]<dist(c,v,G)$, which is a contradiction. Thus our initial assumption was wrong, and there exists a negative weight cycle in G.

[$\Leftarrow$] Suppose that there exists a negative weight cycle $v_0v_1...v_k$, with $v_k=v_0$ in the graph G. Formally, we state this as:

$$\sum_{i=1}^{k}wt(v_{i-1},v_i)<0$$

Let the elements of the vector D after $n-1$ iterations of Bellman-Ford be written as $D[v_i]=d_{n-1}(v_i)$. Suppose that there is no update possible on the nth iteration, i.e. $d_{n-1}(v)\leq d_{n-1}(u)+wt(u,v)$ for all $u,v,\in V$. Then, summing up the inequality for all the vertices on the cycle, we obtain:

$$\sum_{i=1}^{k}d_{n-1}(v_i)\leq\sum_{i=1}^{k}d_{n-1}(v_{i-1})+\sum_{i=1}^{k}wt(v_{i-1},v_i)$$

However, notice that since $v_0=v_k$, the summations on $d_{n-1}(v_i)$ and $d_{n-1}(v_{i-1})$ are exactly the same, and we get,

$$\sum_{i=1}^{k}wt(v_{i-1},v_i)\geq 0$$

This is a contradiction. Thus our assumption that no update takes place at the nth step is incorrect. □

Since our proposed algorithm simply checks whether an update occurs on the nth step of Bellman-Ford, which is a necessary and sufficient condition for the existence of a negative weight cycle, the algorithm is correct. ■

2.1 part 1 10 / 10

- ✓ **+ 6 pts** **Algorithm Correctness**
- ✓ **+ 3 pts** **Correctness Proof**
- ✓ **+ 1 pts** **Running Time Analysis Correctness**
- **+ 5 pts** Algorithm partial
- **+ 2 pts** Correctness Proof Partial
- **+ 0 pts** Incorrect Algorithm

2.2 Printing the Positive Gain Cycle

2.2.1 Algorithm

We expand upon the algorithm from the previous part to print out the positive gain cycle:

1. Construct an auxillary graph $G^* = (V, E, R^*)$, where the updated weights are related to the old weights as:
$$R^*(i,j) = \log\left(\frac{1}{R(i,j)}\right)$$
2. Starting from any currency c_i, compute $n = |V|$ iterations of the Bellman-Ford algorithm, and let D be the vector of distances.
3. If there is an update in D (for some vertex v) in the nth iteration, then there exists a positive gain cycle. Go backward from v using the parent links until a cycle is found, i.e. until either v is reached, or some vertex appears twice.
4. Otherwise, no positive gain cycle exists.

2.2.2 Time Complexity Analysis

We claim the algorithm is $O(mn)$, which is cubic ($O(n^3)$) in the worst case. We analyse the time complexity of the algorithm as follows:

- Running n iterations of the Bellman-Ford algorithm takes $O(mn)$ time as shown in the previous part.
- Further, following parent links from a vertex till a vertex is repeated can take at most $O(n)$, which is the number of vertices in the graph.

Therefore, the algorithm is $O(mn)$. We know that in a complete graph, the number of edges is maximum and is $m = O(n^2)$. Therefore, in the worst case, this algorithm is $O(n^3)$, which is cubic as required.

2.2.3 Proof of Correctness

To argue the correctness of the algorithm, we will prove the following theorem:

Theorem 4. *If the distance to a vertex v changes in the n-th iteration of the Bellman-Ford algorithm, then we must detect a negative weight cycle on following parent pointers from v.*

Proof. Suppose to the contrary that we do not detect a cycle on following the parent pointers from the vertex v. Then, on following the parent pointers we must reach the vertex c_i (the source), and we must have $d_n(c_i) = 0$, otherwise, it must have a parent as well, implying that c_i lies in a cycle, and following the parent pointers we must again reach c_i which contradicts that we did not detect any cycle.

Suppose we get the following sequence of parent pointers:

$$v \to u_1 \to u_2 \to ... \to u_k \to c_i$$

Since this sequence of parents is a path, we must have all vertices in the path are unique. Further, notice that since $d(v)$ was updated in the n-th iteration, it must be true that

$$d_{n-1}(v) > d_{n-1}(u_1) + wt(u_1, v)$$

Similarly,

$$d_{n-2}(u_1) > d_{n-2}(u_2) + wt(u_2, u_1)$$

Otherwise, $d(v)$ would have been updated in the $(n-1)$-th iteration. Proceeding inductively, we see that we must have,

$$d_0(u_k) > d_0(c_i) + wt(c_i, u_k)$$

For n such iterations to have been possible, we must have $k = n - 1$ and thus the length of the path from v to c_i is $n + 1$. However, $|V| = n$, and so by Pigeonhole principle, atleast one of the vertices on this path are repeated, and so it cannot be a path. This contradicts our initial assumption and there must lie a cycle which can be detected by following the parent pointers from the vertex v.

2.2 part 2 7 / 7

- ✓ **+ 4 pts** Algorithm Correctness
- ✓ **+ 2 pts** Algorithm Correctness Proof
- ✓ **+ 1 pts** Time Analysis
- **+ 0 pts** Incorrect Algorithm/Algorithm not given

3 Disjoint collection of Paths

3.1 Algorithm

Before describing the algorithm, let us state some definitions.

Definition 5. *Level of a vertex*: The level of any vertex in the binary tree is defined as the number of edges in the shortest path from that the root of the binary tree to that vertex. By definition, level of the root is 0.

Definition 6. *Level of a path*: The level of a path is defined as the minimum level of all the vertices present in that path in the binary tree. For instance the level of any path passing through the root would be 0.

Without loss of generality we can assume that the vertices in the binary tree are labelled from 1 to n, and the number of paths in the given set S is k. Let us also define some tables and functions and the information they store:

1. Vertex Level Table: This is a table of size n, where the i-th item $L[i]$ denotes the level of the vertex i in the binary tree with root node r ($L[r] = 0$).
2. The paths in the binary tree have been given in the form of their end points. Calculate the lowest common ancestor of the end points and append the path of both the vertices to the lowest common ancestor to get all the vertices in any path P.
3. Path Level Table: This is a table of size k, where $L_p[P]$ denotes the level of the path P in the binary tree.
4. Children Table: This is a table $T'[\,]$ of size n. Initially, the i-th entry $T'[i]$ for the i-th vertex of the tree represents it's children nodes. That is:
   - For any vertex i with children i_1 and i_2, $T'[i] = (i_1, i_2)$.
   - For a vertex with with single i_i child, $T'[i] = i_1$.
   - For a leaf node $T'[i] = -1$

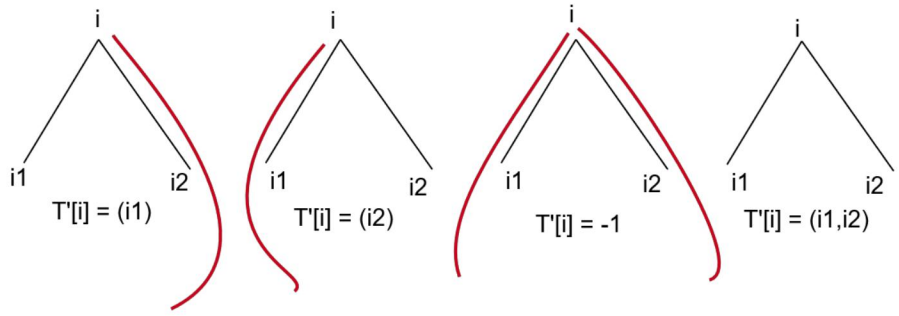

Figure 2: Values for $T'[i]$ as per paths including the vertex i

During the execution of the algorithm, for a given collection of paths which are covered by the algorithm, any entry in table $T'[i]$ represents whether any path from a level higher than that node can enter the sub-tree rooted at it or not without overlapping with already considered paths.

If no path can enter both left and right subtree of vertex i without overlapping with already considered paths then $T'[i] = -1$. Otherwise $T'[i] = (i_1, i_2)$, (i_1) or (i_2) depending on the location of the paths considered. Refer the above figure 2.

5. DP Table: The DP Table $T[\ ]$ is a table of size n. During the algorithm, each$T[i]$ is set such that it represents the maximum cardinality of a subset of paths from set S which lie entirely in the sub-tree rooted at the vertex i. We expect that after the algorithm is complete, $T[r]$ is the required solution, where r is the root of the tree.

6. Path List: For every vertex i, W_i is a list of paths such that for every path $P \in W_i$, $Lp[P] = L[i]$, i.e. the level of every path in this set is equal to the level of i. For a vertex v, this list corresponds to the set of paths passing through it but not it's parent.

7. Path function: The function $f(W_j, S_{j_1}, S_{j_2})$ calculates the number of paths belonging to W_j which can be added to the optimal solution from our algorithm for S_{j_1} and S_{j_2} such that the solution for node j is also a disjoint collection of paths. The computation of the function f is described in Section 3.1.1.

We now propose the following algorithm for finding a maximum cardinality subset of paths of S such that each pair of paths in this subset is edge-disjoint:

- Traverse the tree and populate the vertex level table L, and the children table T'.
- Traverse all the paths and complete the path level table L_p and the path list W_v for all vertices v.
- Begin iterating from the bottom most level of the tree which contains all the leaves. For any vertex j in the binary tree :
  - $T[j] = 0$ if the node is a leaf
  - $T[j] = f(W_j, S_{j_1}, \phi) + T[j_1]$ if the node has 1 child j_1 and the S_{j_1} represents the sub-tree rooted at j_1
  - $T[j] = f(W_j, S_{j_1}, S_{j_2}) + T[j_1] + T[j_2]$ if the node has 2 children j_1 and j_2 and S_{j_1},S_{j_2} represent the sub-tree rooted at j_1,j_2 respectively.
- For every path P in W_j added to the optimal set using the function f in this step of the algorithm, for the vertex j, appropriately modify $T'[\alpha]$ for all vertices $\alpha \in P$, since inclusion of path P in the collection will by definition modify $T'[\alpha]$.
- On reaching the top level, that is the root, return the value $T[r]$ as the answer.

3.1.1 Computation of Path Function

Three types of paths (*blue, red* and *green*) are possible in the set W_j.

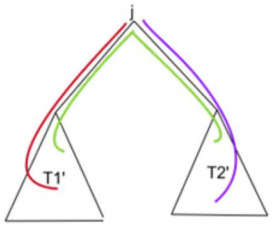

Figure 3: Different types of paths possible in W_j

For an node j with the root of its left and right subtree being j_1 and j_2 respectively, we will first check if

there is a path with j as the end point and j_1 as the next node in path, which doesn't overlap with the any path in S_{j_1} (using the table T' we have constructed). If there exists such a path P_1 then we will increase $f(W_j, S_{j_1}, S_{j_2})$ by 1 and update $T'[v]$ for $v \in P_1$ by including the path.

Similarly we will first check for a path with j as the end point and j_2 as the next node in path, which doesn't overlap with any path in S_{j_2} (using the table T' we have constructed). If there exists such a path P_2 then we will increase $f(W_j, S_{j_1}, S_{j_2})$ by 1 and update $T'[v]$ for $v \in P_2$ by including the path. If we could not find both *blue* and *red* paths then we will look for the remaining *green* paths, and see if there exists such a path with no overlap in S_{j_1} and S_{j_2}.

If there exists such a path then we will increase $f(W_j, S_{j_1}, S_{j_2})$ by 1 and update $T'[j]$ by including the path. $f(W_j, S_{j_1}, S_{j_2})$ will be 2 if we have both *blue* and *red* paths, it will be 1 if we have either a *blue* or *red* path, it will again be 1 if we have a *green* path and it will be 0 if we don't have any appropriate path.

$$0 \leq f(W_j, S_{j_1}, S_{j_2}) \leq 2$$

Checking overlap of a path using T': To check if a path $P = v_1, v_2, \cdots v_m$ overlaps with any other path in a subtree S, we make use of Theorem 8. Consider any 2 adjacent vertices v_k and v_{k+1} in path such that $L[v_k] < L[v_{k+1}]$, it must satisfy $v_{k+1} \in T'[v_k]$ (i.e. the edge (v_k, v_{k+1}) does not lie in any pre-existing path in the subtree S). Thus we can traverse the path once to check its overlap in a given sub-tree. If the path satisfies the above constraints then we can add the path or not according to the calculation of $f(W_j, S_{j_1}, S_{j_2})$ mentioned above and thus update $T'[v]$ for $V \in P$.

3.2 Proof of Correctness

Theorem 7. *For any path P in a binary tree, there is a unique vertex $v \in P$ such that level of path P is equal to the level of the vertex v*

Proof. We will first prove the existence of this vertex v and then justify its uniqueness.

Existence: The level of path is defined as the lowest level of any vertex in the path therefore this guarantees the existence of such a vertex.

Uniqueness: For any vertex $v \in P$, the neighbours of v in P will either be parent or child of P therefore adjacent vertices in P can't have equal levels. Let us say that two vertices $v_1 \in P$ and $v_2 \in P$ have same levels which is also equal to the level of the P. By previous observation, they can't be adjacent in P, and as they are at the top most level, thus the path sub-path from v_1 to v_2 will contain all the vertices with level below v_1 and v_2. There also exists a path in binary tree passing through lowest common ancestor of v_1 and v_2 which contains all the vertices with level above v_1 and v_2. But this is a contradiction as there can't exist 2 paths between v_1 and v_2 in a binary tree. ■

Theorem 8. *Given a binary tree T and a vertex v, with the table $T'[v']$ containing correct entries for all $v' \in T(v)$, where $T(v)$ is the tree rooted at v. Then, a path P contained in $T(v)$ is edge-disjoint with any P' contained in the subtree(s) of $T(v)$ if and only if for all adjacent vertex pairs v_{m_1} and v_{m_2} of P, with $L[v_{m_1}] < L[v_{m_2}]$, we have $v_{m_2} \in T'[v_{m_1}]$.*

Proof. [⇒] Suppose the path P is edge-disjoint with all paths P' contained in subtrees of $T(v)$. Then, consider any 2 adjacent vertices v_k and v_{k+1} in path such that $L[v_k] < L[v_{k+1}]$. If $v_{k+1} \notin T'[v_k]$, then by correctness of T' for the vertex v_k (since $L[v_k] < L[v]$ and P is contained in $T(v)$), the edge (v_k, v_{k+1}) belongs to some path P' contained in the subtree rooted at v_k. This is a contradiction, and thus we must have for all vertices v_{m_1} and v_{m_2} of P, with $L[v_{m_1}] < L[v_{m_2}]$, we have $v_{m_2} \in T'[v_{m_1}]$.

[⇐] Suppose it is true that for all vertices v_{m_1} and v_{m_2} of P, with $L[v_{m_1}] < L[v_{m_2}]$, we have $v_{m_2} \in T'[v_{m_1}]$.

To the contrary, say that $\exists$ some path P' contained in a subtree of $T(v)$ which overlaps with the path P. Let the overlapping edge is (u, v). Suppose $L[u] < L[v]$. Since $(u, v) \in P'$, by definition we must have $u \notin T[v]$, but since $(u, v) \in P$, by our assumption we must have $u \in T[v]$. This is a contradiction, and so we must have that there does not exist a path P' contained in a subtree of $T(v)$ which overlaps with P. ■

3.2.1 Correctness of Algorithm

We can use induction and greedy exchange argument to prove the correctness of algorithm. Let us apply induction on number of nodes in the sub-tree rooted at a node for proving correctness of tables T and T'.

Base Case : If the node is a leaf then $T[v] = 0$ and $T'[v] = -1$ because there are no paths possible in empty sub- tree rooted at that node .

Induction Hypothesis : Let us assume $T[v]$ stores the optimal answer and $T'[v]$ is correctly updated with all the paths belonging to the subtree rooted at v, for any vertex v which has fewer than g nodes in the sub-tree rooted under it.

Induction step : Consider a vertex j with g nodes in the sub-tree rooted under it, and its children j_1 and j_2 with g_1 and g_2 being the size of corresponding sub-trees. We also know that $g_1 < g$ and $g_2 < g$. Therefore by induction hypothesis we have optimal values at $T[j_1]$, $T[j_2]$,$T'[j_1]$ and $T'[j_2]$.

For any path P in W_j, if the path P overlaps with any paths in S_{j_1} or S_{j_2} (using T' to check overlap as discussed in 3.1.1, as per induction hypothesis T' is correctly updated for all vertices in both left and right sub-trees), then we can't simultaneously include the path and the other paths with which it overlaps in the solution set. The inclusion of path P in the solution set can cause removal of 1 or more than 1 paths which overlap with it from the solution set but will result in inclusion of just 1 path . If it leads to removal of more than 1 path then we have a more optimal solution which is to not include P. If it leads to removal of just 1 path P_1, then since $Lp[P] < Lp[P_1]$, any path with a level lower than P which overlaps with P_1, will overlap with P as well and there can be paths with lower level which overlap with P, but don't overlap with P_1. Therefore we will not include such paths P in our solution set.

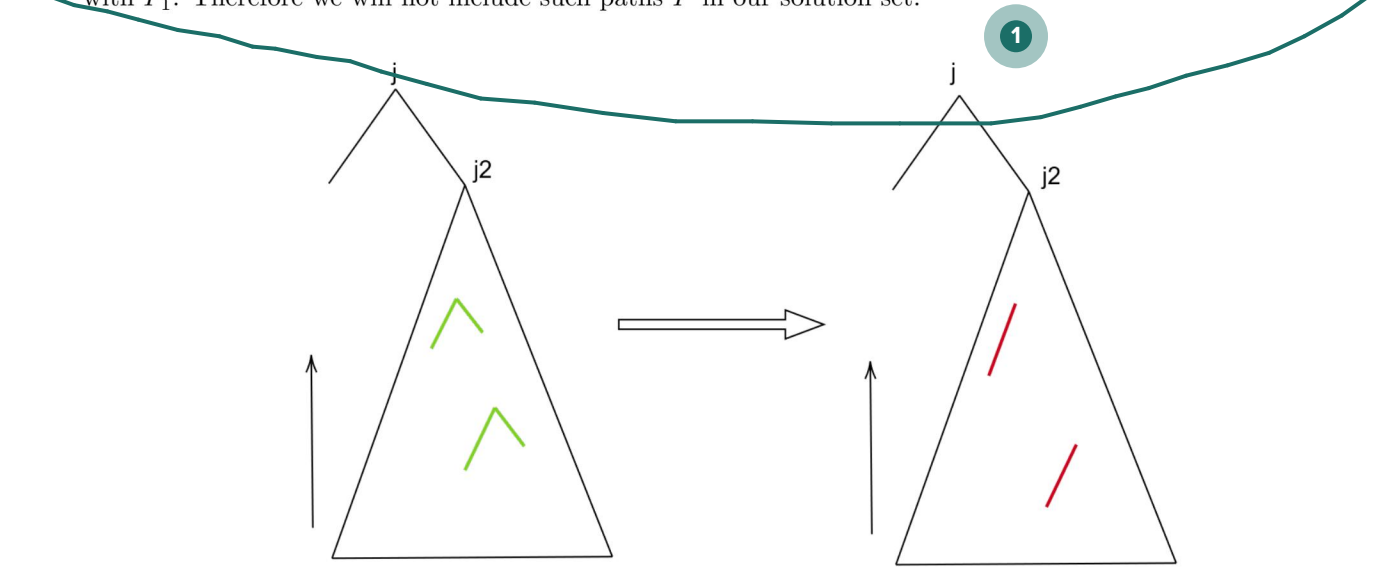

Figure 4: Greedy Replacement Argument

We can greedily argue about the order of adding paths from W_j to $T'[j]$. Let us call the paths in W_j which just enter the left sub-tree as *red* paths, the paths in W_j which just enter the right sub-tree as *blue* paths and those which enter both subtrees as *green* paths as shown in the Figure 4. We insert both *red* and *blue* if possible because they increase the cardinality by 2 giving the most optimal answer, but then we give priority to insertion of *red* or *blue* paths over insertion of *green* paths.
Consider including a possible *green* path at a vertex j instead of *red* or *blue* path. For any path P' with $Lp[P'] < L[j]$, if P' overlaps with the *red* or *blue* path at j, then it will also overlap with *green* path at j, since the *green* path shares the edge from vertex j with both types of paths. If P' is such that it enters the left subtree of j, then it will not overlap with the *blue* path but will overlap with *green* path thus affecting the optimality. The same holds for its entry in the right subtree. Therefore for any node j' such that $L[j'] < L[j]$, the size of solution set that is $T[j']$ on including a *green* path at $j \leq$ the size of solution set on including *red* or *blue* paths.

Argument for Inclusion of such Non-Overlapping Paths: The solution we are getting from our algorithm is $f(W_j, S_{j_1}, S_{j_2}) + T[j_1] + T[j_2]$, let us say a better solution $f(W_j, S_{j_1}, S_{j_2}) + T[j_1] + T[j_2] + \beta$ exists at a vertex j where $\beta > 0$. Let γ paths out of $f(W_j, S_{j_1}, S_{j_2}) + T[j_1] + T[j_2] + \beta$ be such that level of those paths is equal to $L[j]$. This means the remaining $f(W_j, S_{j_1}, S_{j_2}) + T[j_1] + T[j_2] + \beta - \gamma$ paths belong to the left and right subtree.
If $f(W_j, S_{j_1}, S_{j_2}) + \beta > \gamma$, then we will find a better solution for left or right subtree which contradicts the hypothesis.
If $0 < \beta \leq \gamma - f(W_j, S_{j_1}, S_{j_2})$, then as the maximum value of both $f(W_j, S_{j_1}, S_{j_2})$ and γ can be 2 thus the only possible combinations is $\gamma = 2, f(W_j, S_{j_1}, S_{j_2}) = 1$ or $\gamma = 1, f(W_j, S_{j_1}, S_{j_2}) = 0$ or $\gamma = 2, f(W_j, S_{j_1}, S_{j_2}) = 0$. The first two cases would lead to contradiction as both the sub trees have same number of paths as the optimal solution but the solution at node j is better, which is shown in 3.2.1 (replacing all the red/blue paths with green paths should always reduce or maintain the size of solution set of above vertices).
In the third case if $\beta = 2$, we get a similar contradiction, if $\beta = 1, \gamma = 2$ then the number of paths in left and right subtree is $T[j_1] + T[j_2] - 1$, thus at least one of the left or right subtree will have paths equal to the optimal solution. Since we are having $\gamma = 2$ (thus there is a path that goes both to left and right subtree), either left or right subtree has paths equal to optimal solution with a path from vertex j which is a contradiction to $f(W_j, S_{j_1}, S_{j_2}) = 0$.
Thus $\beta = 0$

3.3 Time complexity analysis

Let there be n vertices in the binary tree and k paths in the given set S

1. Labeling each node with its level take $O(n)$ time which involves top down traversal of the binary tree.
2. Finding all the vertices in any path involves the calculation of lowest common ancestor ($O(n)$) and then appending the two paths to the lowest common ancestor ($O(n)$). So for k paths, the step is $O(kn)$
3. Labeling each path with its level takes $O(kn)$ time which involves traversing each of the k paths with maximum length $n-1$ and finding the minimum level from its vertices.
4. Checking whether a path overlaps with any subtree and if it doesn't overlap, whether it is a *red,blue* or *green* path, takes $O(n)$ time as maximum length of a path is $O(n)$.
5. For any vertex j, let us say α_j denote the size of W_j. Thus we know that:

   $$\alpha_1 + \alpha_2 + \cdots \alpha_n = k$$

   Therefore iterating in bottom up manner and checking which paths from the α_j paths to include using **step 4** takes

   $$(\alpha_1 + \alpha_2 + \cdots \alpha_n) \times O(n) = O(nk)$$

Thus overall time complexity is $O(nk)$

3 Disjoint Paths 22 / 25

+ **2 pts** Identifying we need to use DP
+ **3 pts** Correctly defining the DP variables
+ **10 pts** Correctly defining the recursive(DP) equations

✓ **+ 15 pts** **Choosing the paths greedily in non increasing order of depth of LCA of the endpoints**

✓ **+ 5 pts** **Proof of Correctness**

✓ **+ 5 pts** **Proving $$O(poly(n))$$ Time Complexity**

+ **0 pts** Incorrect/Unattempted

- 3 **Point adjustment**

1 For fixed v, there can be multiple options for OPT(v.left) and OPT(v.right), and for some of them, p may overlap and for some p may not. So we cant use this argument.

COL351: Assignment 2

Vishwas Kalani (2020CS10411)
Viraj Agashe (2020CS10567)

September 2022

Contents

1 Maximum Sum **2**
1.1 Algorithm . . . 2
1.2 Time Complexity Analysis . . . 2
1.3 Proof of Correctness . . . 3
1.3.1 Case I . . . 3
1.3.2 Case II . . . 4

2 Forex Trading **6**
2.1 Detection of Positive Gain Cycle . . . 6
2.1.1 Algorithm . . . 6
2.1.2 Pseudocode . . . 6
2.1.3 Time Complexity Analysis . . . 6
2.1.4 Proof of Correctness . . . 6
2.2 Printing the Positive Gain Cycle . . . 8
2.2.1 Algorithm . . . 8
2.2.2 Time Complexity Analysis . . . 8
2.2.3 Proof of Correctness . . . 8

3 Disjoint collection of Paths **10**
3.1 Algorithm . . . 10
3.1.1 Computation of Path Function . . . 11
3.2 Proof of Correctness . . . 12
3.2.1 Correctness of Algorithm . . . 13
3.3 Time complexity analysis . . . 14