CS 577: HW 5

Daniel Ko

Summer 2020

§1 Joe is a project manager in a software company called SuperDuo. Joe found out that programmers usually produces the highest quality codes when they are working in pair. However, the productivity of each pair of programmers is the speed of the slower programmer. So Joe needs to figure out the best strategy for pairing the programmers to generate the maximum sum of productivity for his project. Suppose that the number of programmers in Joe's project is even.

§1.1 Give a greedy strategy that maximizes the sum of the productivity of all pairs. You should describe the greedy strategy first and then write it in the form of an algorithm. Also you should analyze the computing complexity of the greedy algorithm.

The greedy strategy is to first sort the programmers in ascending order of productivity. Then, we pair the first two programmers, then the next two, etc. We sum up all the slower programmer in each pair, which will be our maximum sum of productivity.

```
Data: A list gamer of even size n, containing the productivity level of each programmer
Result: The maximum sum of productivity for pair programming
def maxProd(gamer):
    sortedGamer ← Sort gamer in ascending order of productivity
    totalProd ← 0
    for int i = 0; i < n; i+=2:
        totalProd += sortedGamer[i]
    return totalProd
```

The time complexity of maxProd is $O(n \log n)$

Proof. We first sort the list gamer of size n. Using an efficient sorting algorithm such as merge sort, this will take $O(n \log n)$ time. Then, we sum up every other element in the list, which takes $O(n)$ time. The rest of the computations take constant time. Hence, the total time complexity is $O(n \log n + n) = O(n \log n)$ □

§1.2 Prove that this greedy strategy indeed generates the maximum sum of productivity.

We will use a "Greedy Stays Ahead" argument to prove this greedy strategy.

§1.2.1 Define Your Solution

Let $A = \{a_1, a_2, \cdots, a_{n/2}\}$ denote the pairs containing the productivity values of two programmers that our algorithm will generate, i.e. $a_i = (p_\alpha, p_\beta)$ where p_α and p_β are the productivity values of α and βth lowest productive programmer in this ith pair respectively. Let $O = \{o_1, o_2, \cdots, o_{n/2}\}$ denote the pairs containing the productivity values of two programmers that an optimal algorithm will generate. Let O be ordered in ascending productivity, i.e. if $i > j$, the productivity of pair o_i will be equal to or higher than pair o_j.

§1.2.2 Define Your Measure

For our algorithm, let $\delta(a_1), \cdots, \delta(a_{n/2})$ denote the total productivity of pairs from a_1 to a_i. For example, $\delta(a_3)$ will be the sum of the productivity (the lower productivity between the two programmers) of pairs a_1, a_2, and a_3. Formally, we can express this function as

$$\delta(a_i) = \sum_{j=1}^{i} \min(a_j)$$

Similiary for the optimal algorithm, the $\delta(o_1), \cdots, \delta(o_{n/2})$ denote the total productivity of pairs from o_1 to o_i.

§1.2.3 Prove Greedy Stays Ahead

We show by induction that our greedy stays ahead.

Proof. Let $P(i)$ be the predicate, "$\delta(a_i) \geq \delta(o_i)$" We define $i \in \mathbb{N}^+$.
Base Case: When $i = 1$, we know that a_1 contains the lowest productivity programmer because we paired up the two lowest productivity programmers in our algorithm. We also know that o_1 contains the lowest productivity programmer because we ordered O in ascending productivity. The least productive pair must contain the lowest productivity programmer because the productivity any pair containing the lowest productivity programmer will the productivity value of the lowest productivity programmer. Hence, the productivity of pair a_1 will be the value of lowest productivity programmer, and similarly, the productivity of pair o_1 will be the value of lowest productivity programmer. Thus, $\delta(a_1) \geq \delta(o_1)$ and $P(1)$ holds.
Inductive step: Suppose $P(k-1)$ holds. We show that $P(k)$ holds. By our algorithm, we know that pair a_k contains the $(2k-1)$th lowest productive programmer and a_k's productivity value will be the productivity value of the $(2k-1)$th lowest productive programmer, i.e. $\min(a_k) = p_{2k-1}$. We claim that we can bound the productivity value of pair o_k by $\min(o_k) \leq p_{2k-1}$. We know that since O is ordered in ascending productivity, the productivity value of pair o_k must be equal or greater to the pairs that came before it. There are $2k-2$ programmers that come before pair o_k. This means that at most, there are $2k-2$ programmers with lower or equal productivity value than productivity value of pair o_k from the pairs behind o_k. If we include the programmer that determines the productivity value of pair o_k, we have $2k-1$ programmers with lower or equal productivity value than productivity value of pair o_k. It follows from the pigeonhole principle that $\min(o_k) \leq p_{2k-1}$ because if $\min(o_k) \geq p_{2k}$ then there would exist some programmer with productivity value equal or less than p_{2k-1} that will not belong in the pairs behind o_k but rather ahead of o_k, which would be a contradiction because O is ordered, i.e. p_{2k} will appear before p_{2k-1}. Using this bound we know that $\min(a_k) \geq \min(o_k)$ and by our induction hypothesis we know that $\delta(a_{k-1}) \geq \delta(o_{k-1})$.

It follows that,

$$\begin{aligned}\delta(a_k) &= \delta(a_{k-1}) + \min(a_k) \\ &\geq \delta(o_{k-1}) + \min(o_k) \\ &\geq \delta(o_k)\end{aligned}$$

$P(k)$ holds as desired. Therefore by induction, $\delta(a_i) \geq \delta(o_i)$ for all $i \in \mathbb{N}^+$. □

§1.2.4 Prove Optimality

Since both O and A have the same number of pairs, $n/2$, it follows from our proof from 1.2.3 that $\delta(a_{n/2}) \geq \delta(o_{n/2})$. Therefore, our algorithm is optimal because the total productivity of A is equal or greater than the total productivity of O.

§2 Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

§2.1 Describe a greedy strategy and write it in the algorithm format. Also analyze the computing complexity of the algorithm.

This algorithm recursively makes both subtrees of a vertex zero skew and then computes the correct cost to add to either the left or right edge to make the entire tree zero skew. We recurse all the way down to the leaves and notice that subtrees where its root is a leaf is already zero skew so we return 0. On the following layers up, we consider the edge length from the current node v and its left and right child, l and r, and the time it takes l to go its leaves and r to go its leaves. We denote the time it takes l and r to go its leaves, δ_l and δ_r respectively. We check whether $\delta_l + \ell_{v,l}$ or $\delta_r + \ell_{v,r}$ is greater and adjust $\ell_{v,l}$ or $\ell_{v,r}$ accordingly to make the subtree zero skew. We continue this process until the entire tree is zero skew.

Data: $T = (V, E, \ell)$, a complete balanced binary tree with an associated edge length ℓ
$v \in V$, a vertex for a subtree to make zero skew
Result: Length from root to leaf (zero skew value); a tree has zero skew and the total edge length is as small as possible

```
def timingCircuit(v):
    if v is a leaf:
        return 0
    δ_l = timingCircuit(l)
    δ_r = timingCircuit(r)
    # Where ℓ_(a,b) is the associated length of edge (a, b) ∈ E
    if δ_l + ℓ_(v,l) > δ_r + ℓ_(v,r):
        Increment ℓ_(v,r) by (δ_l + ℓ_(v,l)) − (δ_r + ℓ_(v,r))
    elif δ_r + ℓ_(v,r) > δ_l + ℓ_(v,l):
        Increment ℓ_(v,l) by (δ_r + ℓ_(v,r)) − (δ_l + ℓ_(v,l))
    return δ_r + ℓ_(v,r)
```

The time complexity of timingCircuit is $\Theta(n)$.

Proof. Each recursive call in our algorithm halves the input size and we have two recursive

calls. The rest of the computations take constant time. We get a recurrence relationship of $T(n) = 2T(n/2) + n^0$ which is $\Theta(n)$ by the master theorem. □

§2.2 Prove that this greedy strategy indeed generates the optimal solution

Lemma 2.1

The algorithm timingCircuit will always generate a zero skew tree.

Proof. We show by strong induction that timingCircuit will always generate a zero skew tree. Let $P(n)$ be the predicate "timingCircuit will generate a zero skew tree that has n leaves". We define $n \in \mathbb{N}^+$.
Base Case: When $n = 1$, timingCircuit returns 0. It is trivially true that a tree with a single node will always have zero skew. Hence $P(1)$ holds.
Inductive step: Suppose for all $1 \leq n \leq k$, $P(n)$ holds. We show that $P(k+1)$ holds. We know that the subtrees generated by the children of v, l and r, will have strictly less leaves than $k+1$. It follows by our strong inductive hypothesis, that timingCircuit(l) and timingCircuit(r) will correctly generate a zero skew subtrees for the subtrees that have l and r has its root, and return the distance from l to its leaves, δ_r, and r to its leaves, δ_l. Since the two subtrees generated by the children v are zero skew, all we need to do so that the entire tree is zero skew is to modify the edge length between (v, l) and (v, r) such that $\delta_r + \ell_{(v,r)} = \delta_l + \ell_{(v,l)}$, i.e. both subtrees have same zero skew if we start from v. This is exactly what we do in the if and else if statement in our algorithm. Hence, timingCircuit generates a zero skew tree that has $k+1$ leaves and $P(k+1)$ holds. Therefore by strong induction, timingCircuit will always generate a zero skew tree. □

We define terms for the following theorem. Let T be the original tree and let ℓ be its associated edge length. Let v be the root of T and let l and r be the left and right child of v respectively. Let T_α be the tree generated by timingCircuit. Let T_ϕ be an optimal tree, i.e. a zero skew tree such that the total edge length is as small as possible. Let the left and right subtree of T_α be L_α, R_α. Similiarly, define L_ϕ, R_ϕ. Let ℓ^α and ℓ^ϕ and the associated edge length of T_α and T_ϕ respectively.

Theorem 2.2

The algorithm timingCircuit will always generate a zero skew tree such that the total edge length is as small as possible.

Proof. From lemma 2.1, we know that timingCircuit will always generate a zero skew tree. It is sufficient to show that timingCircuit will minimize total edge length when generating this tree. We continue with an exchange argument. Suppose the optimal solution, T_ϕ, does not use the edge weights generated by our algorithm, T_α, i.e. total edge length of T_ϕ is strictly less than total edge length of T_α. This means there exists an edge (a, b) such that $\ell^\alpha_{(a,b)} > \ell^\phi_{(a,b)}$. Fix (a, b) such that it is the closet edge from the leaves. Let U be an induced subgraph of T_ϕ such that it only contains edges which its weight has not been increased compared to T. Let the connected component containing b in U be X. Define Y to be the set of edges such that one vertex is in X and one vertex is in $E \setminus X$, where E is the set of all edges in T. Let ζ to be the smallest change on edge length between T_ϕ and T in E, or more formally $\zeta = \min\{\ell^\phi_{(a,b)} - \ell_{(a,b)} \mid (a, b) \in E\}$. Notice we can construct a new tree, T_o, from T_α where we reduce all edges by ζ in E and increase the edge of (a, b) by ζ. T_o will still be zero skew by construction and its total edge weight will be smaller or equal than T_ϕ, making it optimal. Without loss of generality, we can perform this process to all edges (a, b) where $\ell^\alpha_{(a,b)} > \ell^\phi_{(a,b)}$. At the end of the process, we will have edge weights produced by our algorithm, such that it is no greater than that the edge weights of T_ϕ we considered. Therefore, the edge weights produced by our algorithm must also be optimal.

□