The Maximum Ratio Subtree Problem

We will use mergeable heaps to solve the following optimization problem defined on trees.

We are given a tree T with n nodes, and each node u of the tree stores two positive numbers, a_u and b_u. In general in this problem you don't need the tree to be binary, but to keep things simple, let's assume that it actually is binary.

We will call T' a subtree of T if it is connected and contains a (nonempty) subset of the nodes of T together with the edges between them. In Figure 1 you see some examples of subtrees and one example of a non-subtree.

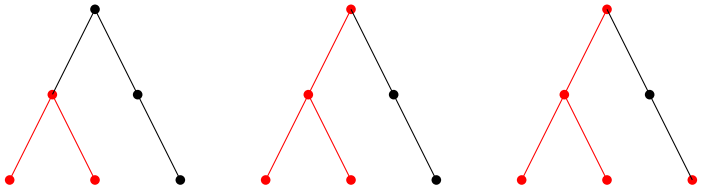


Figure 1: The first two pictures (from the left) show subtrees (marked in red). The last picture is not a subtree because it is not connected.

For a subtree T', let $r(T') = \frac{\sum_{u \in T'} a_u}{\sum_{u \in T'} b_u}$. I.e. $r(T')$ is the ratio of the sum of the a values to the sum of the b values, where both sums are over the nodes of T'.

For any node u of T, let $\Gamma(u)$ be the set of all subtrees of T whose root is u. Let u^* be the root of T. Our goal is to find the value

$$r(u^*) = \max\{r(T') : T' \in \Gamma(u^*)\}.$$

In fact, we will get an algorithm which computes

$$r(u) = \max\{r(T') : T' \in \Gamma(u)\}$$

for all nodes u of T, starting from the leaves, and making its way to the root.

This problem was used by Horn to solve a scheduling problem [Hor72]. In Horn's model, we have jobs represented by nodes of a tree, with the constraint that each job u can only be executed after the jobs on the path from u to the root of the tree are executed. These constraints are called precedence constraints. Suppose that job u takes time b_u, and that we promised the client who asked for job u to be executed that we would refund them a_u dollars for each time unit they have to wait before u is executed. Our goal is to schedule the jobs one after the other so that we minimize the total refunds, i.e. to arrange the jobs in a sequence $u_1, \ldots, u_n$ so that the objective

$$\sum_{i=1}^{n} a_{u_i} \sum_{j=1}^{i} b_{u_j}$$

is minimized. Horn showed that this problem can be solved efficiently if we can find the $r(u)$ values defined above. Check his paper for the details.

In this note we describe Horn's algorithm for computing the $r(u)$ values. More specifically, we describe an efficient implementation of it, due to Galil [Gal80], using mergeable priority queues.

Before we describe the algorithm, let's make one simple but crucial observation.

Proposition 1. *If a, b, c, d are positive numbers, then*

$$\frac{a+c}{b+d} \geq \frac{a}{b} \iff \frac{c}{d} \geq \frac{a}{b}.$$

Similarly, if a, b, c, d, c', d' are positive numbers, then

$$\frac{a+c}{b+d} \geq \frac{a+c'}{b+d'} \iff \frac{c}{d} \geq \frac{c'}{d'}.$$

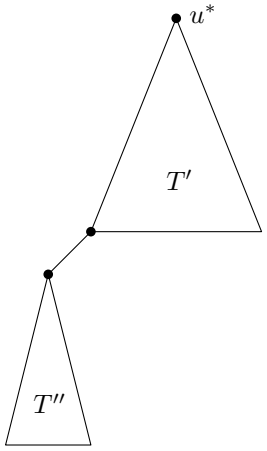
Proof. Let $r = \frac{a}{b}$ and $r' = \frac{c}{d}$. Then

$$a + c = r(b + d) + (r' - r)d.$$

Since $d > 0$, if $r' - r \geq 0$, then $a + c \geq r(b + d)$, i.e. $\frac{a+c}{b+d} \geq \frac{a}{b}$. Conversely, if $r' - r < 0$, then $a + c < r(b + d)$, i.e. $\frac{a+c}{b+d} < \frac{a}{b}$.

The proof for the second claim is analogous. □

Consider the figure below, where T' and T'' are subtrees of some larger tree, and, say, T' is rooted at u^*. In this picture, the parent of the root node of T'' lies in T'. By Proposition 1, if $r(T'') \geq r(T')$, then $r(T' \cup T'') \geq r(T')$, so if T' maximizes $r(T')$ over all subtrees in $\Gamma(u^*)$, then so does $T' \cup T''$. In other words, if $r(u^*) = r(T')$ and $r(T'') \geq r(T')$, then $r(u^*) = r(T' \cup T'')$. Conversely, if $r(T'') < r(T')$, then $T' \cup T''$ cannot be optimal, i.e. $r(T' \cup T'') < r(T') \leq r(u^*)$.



This motivates an algorithm which keeps merging trees as long as the r-value increases. To describe it formally, we need one more piece of notation. For a subtree T', we let $B(T')$ be the set of nodes of T which are not in T' but whose parent is in T'. This is sort of a "boundary" around T'. (See Figure 2).

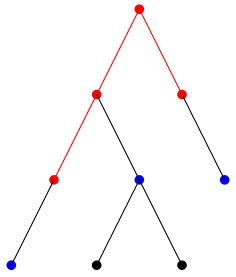


Figure 2: The blue nodes are the boundary set $B(T')$ for the subtree T' shown in red.

We first describe the algorithm on a high level. Then we will discuss implementation. In the algorithm description below $\tilde{T}_u$ will eventually equal a subtree rooted at u such that $r(\tilde{T}_u) = r(u)$, i.e. an optimal subtree rooted at u. If there are many such subtrees, $\tilde{T}_u$ will be the one with the maximum number of nodes. (This makes it unique, although this fact is not immediately obvious.)

1. For each leaf node u, set $\tilde{T}_u$ to just be the node u, and $r(u)$ to be $\frac{a_u}{b_u}$. (There is no other option for the leaves.) Call the leaf nodes "processed".

2. Take any internal node u all of whose children have been processed. Let its children be v and w. Initialize $\tilde{T}_u$ to u, $r(u)$ to $\frac{a_u}{b_u}$, and $B(\tilde{T}_u)$ to $\{v, w\}$.

3. As long as $B(\tilde{T}_u)$ is not empty, take the node $v \in B(\tilde{T}_u)$ with the largest $r(v)$. If $r(v) < r(u)$ or if $B(\tilde{T}_u)$ is empty, call u "processed". If, otherwise, $r(v) \geq r(u)$, then reset $r(u)$ to $r(\tilde{T}_u \cup \tilde{T}_v)$, $\tilde{T}_u$ to $\tilde{T}_u \cup \tilde{T}_v$, and reset $B(\tilde{T}_u)$ to $B(\tilde{T}_u \cup \tilde{T}_v)$.

4. Repeat steps 2. and 3. until all nodes are processed.

For the correctness of this algorithm, we need to argue that when a node u has been processed, then $\tilde{T}_u$ is the subtree rooted at u which satisfies $r(\tilde{T}_u) = r(u) = \max\{r(T') : T' \in \Gamma(u)\}$ and has maximum number of nodes. We sketch the argument. It is obvious that this holds for the leaves, because for any leaf node u, $\Gamma(u)$ contains only one subtree: the single-node tree u itself. For the other nodes, we proceed by induction (on the number of steps of the algorithm). When we are processing a node u we know that all nodes below it have already been processed. Let T^* be the subtree rooted at u such that $r(T^*) = \max\{r(T') : T' \in \Gamma(u)\}$ and T^* has the maximum number of nodes among such subtrees. Our goal is to show that when u is already processed, $\tilde{T}_u = T^*$.

We claim that at any point in time when u is still being processed, $\tilde{T}_u$ is a subtree of T^*. This is certainly true initially, when $\tilde{T}_u$ is just u, because we know that T^* must be rooted at u. We proceed by induction on the number of times we add a tree to $\tilde{T}_u$. Suppose that at some point $B(\tilde{T}_u) = \{u_1, \ldots, u_k\}$, where $r(u_1) \geq r(u_2) \geq \ldots \geq r(u_k)$. The algorithm adds $\tilde{T}_{u_1}$ to $\tilde{T}_u$ if $r(u_1) \geq r(u)$. If $r(u_1) < r(u)$ then there is nothing to prove because $\tilde{T}_u$ does not change. So, suppose that $r(u_1) \geq r(u)$ but, towards contradiction, $\tilde{T}_{u_1}$ is not a subtree of T^*. Then we have the following claims:

(a) We must have $u_1 \notin T^*$.

(b) $r(u_1) = r(\tilde{T}_{u_1}) < r(T^*)$

Both claims hold because, otherwise, we can merge $\tilde{T}_{u_1}$ with T^* and, by the second case of Proposition 1, not decrease $r(T^*)$. These two claims lead to a contradiction: the first one implies that $r(T^*) \leq \max\{r(u), r(u_2)\} \leq r(u_1)$ (try to verify this using Proposition 1), which contradicts the second one.

We have then established that $\tilde{T}_u$ is a subtree of T^* after u is processed. It cannot be a strict subtree, because if it were, then there would be some $v \in B(\tilde{T}_u)$ which is a node of T^*. But we know that for all v in $B(\tilde{T}_u)$ we have $r(v) < r(\tilde{T}_u)$, which would imply that $r(T^*) < r(\tilde{T}_u)$. (Again, try to verify this using Proposition 1.) Therefore, $\tilde{T}_u = T^*$, which completes the analysis.

Let us now see how to implement this algorithm. We will not actually keep the trees $\tilde{T}_u$ because we only care about the values $r(u)$. (You can modify the implementation to also compute the trees without increasing the asymptotic running time: this is a useful exercise.) The idea is that every node u will have a binomial max-heap $u.H$ which stores the nodes in $B(\tilde{T}_u)$, and the key of every node v equals $r(v)$, which we will store in $v.r$. Merging $\tilde{T}_u$ and $\tilde{T}_v$ just corresponds to taking the union of $u.H$ and $v.H$. We will also store the sum $\sum_{v \in \tilde{T}_u} a_v$ in $u.num$ and the sum $\sum_{v \in \tilde{T}_u} b_v$ in $u.denom$, so that we can easily compute $r(u) = u.r = \frac{u.num}{u.denom}$.

The code below takes a pointer u to a node in T

```
COMPUTER(u)
 1  u.num = a_u
 2  u.denom = b_u
 3  u.r = u.num/u.denom
 4  if u is a leaf
 5       Initialize u.H to be empty and return
 6  else Let v and w be the children of u
 7       COMPUTER(v)
 8       COMPUTER(w) // If u has only one child v, ignore COMPUTER(w)
 9       Initialize u.H to contain v and w, with keys v.r and w.r.
10       while u.H is not empty
11            v = MAX(u.H)
12            if v.r ≥ u.v
13                 u.num = u.num + v.num
14                 u.denom = u.denom + v.denom
15                 u.r = u.num/u.denom
16                 EXTRACT-MAX(u.H)
17                 u.H = UNION(u.H, v.H)
18            else return
19  return
```

To compute all the r-values, we just call COMPUTER(u^*) once at the root u^*. Notice that the procedure is only called once for every root of T. It is easy to see that the running time is dominated by the total running time of the heap operations. The total number of insertions is bounded by n, since each node is inserted at most once. The number of MAX and EXTRACT-MAX operations is bounded by the number of UNION operations. We can charge any UNION($u.H, v.H$) operation, where v is on a lower level than u, to v. Every node is charged in this way at most once, because we remove it from the heap $u.H$ before the union, so v will never again be returned by a MAX operation and participate in a union. The number of unions is then bounded by n, and, therefore, the total number of heap operations is bounded by $O(n)$. Since each binomial heap operation takes time $O(\log n)$, the total running itme is $O(n \log n)$.

References

[Gal80] Zvi Galil. Applications of efficient mergeable heaps for optimization problems on trees. *Acta Inf.*, 13:53–58, 1980.

[Hor72] W. A. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal on Applied Mathematics*, 23(2):189–202, 1972.