University of Toronto at Scarborough
CSCC73 - Algorithm Design and Analysis

Assignment #1: Greedy Algorithms

Due: Jan 21, 2023 at 11.59pm This exercise is worth 5% of your final grade.

Warning: Your electronic submission affirms that this exercise is your and your partners own work and no one else's, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters, the Code of Student Conduct, and the guidelines for avoiding plagiarism in CSCC73. Please see syllabus for details. Late assignments will not be accepted. If you are working with a partner your partners' name must be listed on your assignment and you must sign up as a "group" on Gradescope. Recall you must not consult **any outside sources except your partner, textbook, TAs and instructor**.

1. In this question we consider a variation of the scheduling problem in which we are given n distinct jobs j_i each of which can be completed independently. Each job has two stages to be completed. The first stage of each job is the bottleneck in the sense that there is only one resource that can handle it and for job j_i takes f_i time. The second stage is less complicated and can be completed in parallel with other jobs as there are many resources available to complete this portion of the job. We will denote the time required to complete stage 2 of job j_i as s_i. [10]

   For example, consider a factory where there is a unique machine that must be used in the production of each product before it can go through the final stages with an employee. Assume that there are enough employees (i.e., n employees) to handle the second stage of all the jobs at the same time. Give a polynomial-time algorithm to return a *schedule* that orders the jobs for stage 1 so that the *completion time* of stage 2 for all the jobs is minimized (ie, the total time for all jobs to be completed is minimized). Prove your algorithm is correct and explain it's complexity.

   Solution:

   It is clear that the working time for completion of the first stage of all the jobs does not depend on the ordering of the jobs. Thus we can not change the time when the last job hands off to stage 2. It is intuitively clear that the last job in our schedule should have the shortest completion time. Hence the *greedy strategy*:

   `Greedy Schedule` G: Run the jobs in order of decreasing second stage time s_i.

   We will assume that this ordering is $j_1, j_2, \ldots, j_n$.

   PROOF. Note that one can use a "greedy stays ahead" or induction argument or an inversion argument. We give the inversion version here.

   We use an inversion argument similar to the inversion argument from class. The jobs in the greedy schedule are sorted by decreasing duration of the second stage. Define an inversion when $i < j$ for two jobs but $s_i < s_j$, ie, their ordering is *inverted* with respect to the greedy ordering.

   Suppose that there exists an optimal solution O such that two jobs o_i and o_j are inverted from the greedy schedule, ie., $i < j$ but the duration of the second stage s_i for o_i is shorter than the second stage s_j for o_j, $s_i < s_j$. As in lecture we can argue that if an inversion exists, there exists consecutive jobs o_k, o_{k+1} that are inverted. Consider swapping these two jobs. Notice that for all $m < k$ and $m > k+1$ the start time for o_m remains unchanged since the first stage time still takes the same.

   Now consider when the second stage for o_k and o_{k+1} finish. If o_k originally started at t_k, it's first stage and second stage times are f_k and s_k so it finished at

   $$t_k + f_k + s_k$$

and o_{k+1} finished at

$$t_k + f_k + f_{k+1} + s_{k+1}.$$

We indicate the job o_k after the swap as o'_k and similarly for o_{k+1} after the swap as o'_{k+1}. After the swap, o'_k finishes at

$$t_k + f_{k+1} + f_k + s_k$$

and o'_{k+1} at

$$t_k + f_{k+1} + s_{k+1}.$$

Recall that $s_k < s_{k+1}$:

after the swap o'_k finishes at

$$t_k + f_{k+1} + f_k + s_k < t_k + f_{k+1} + f_k + s_{k+1}$$

which is before o_{k+1} originally finished. Therefore o'_k does not increase the overall running time of all the jobs in O.

Also notice that o'_{k+1} finishes after the swap at:

$$t_k + f_{k+1} + s_{k+1} < t_k + f_k + f_{k+1} + s_{k+1}$$

which is earlier than it originally finished so again the overall running time has not increased.

Hence this new schedule with one less inversion is also optimal. We can argue inductively that we can remove all inversions to end up with the greedy schedule.

Complexity is bounded by sorting, so $\mathcal{O}(n \log n)$.

Common proof mistakes.

- With an exchange argument type proof, the argument should start with an arbitrary schedule S (which should be an optimal one) and use exchanges to show that this schedule S can be turned into the schedule the greedy algorithm produces without making the overall completion time worse. It does not work to start with the algorithm's schedule G and simply argue that G cannot be improved by swapping two jobs. This argument would show only that a schedule obtained from G in a single swap is not better; it would not rule out the possibility of other schedules, obtainable by multiple swaps that are better.
- To make the argument work smoothly, one should try to swap neighbouring jobs. If you swap two jobs i and j that are not neighbouring, then all the jobs between the two also change their finishing times.
- In general, it does not work to phrase and exchange argument as a proof by contradiction. The problem is that there could be many optimal schedules, so there is no contradiction in the existence of one that is not G. note that when we swap adjacent, inverted jobs, it does not necessarily make the schedule better; we only argue that such swamps do not make it worse.

2. Timing circuits are a crucial component of VLSI chips. Here's a simple model of such a timing circuit. Consider a complete balanced binary tree with n leaves, where n is a power of 2. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

   The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

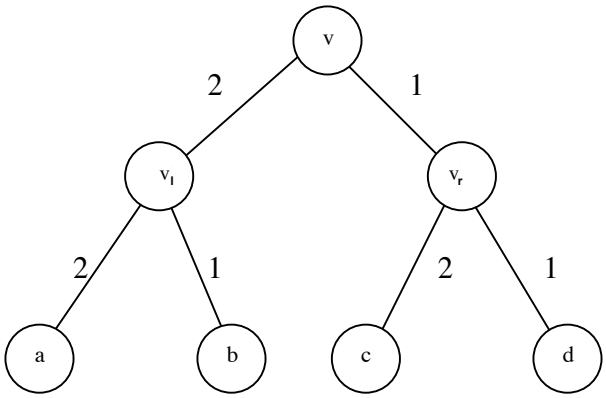

Figure 1: An instance of the zero-skew problem.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to received the signal at the same time. To make this happen, we will have to *increase* the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

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.

Example. Consider the tree in Figure 1, in which letters name the nodes and numbers indicate the edge lengths. The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

Solution:

Let v denote the root and v_l and v_r it's two children. Let d_l denote the maximum root to leaf distance over all leaves that are descendeants of v_l and let d_r denote the maximum root to leaf distance over all leaves that are descendants of v_r.

Notice that:

- If $d_l > d_r$ then we add $d_l - d_r$ to the length of the v to v_r edge and add nothing to the length of the v to v_l edge.
- If $d_l < d_r$ then we add $d_r - d_l$ to the length of the v to v_l edge and add nothing to the length of the v to v_r edge.
- If $d_l = d_r$ we add nothing.

We now apply this procedure recursively to the subtrees rooted at each v_r and v_l.

Let T be the complete binary tree in the problem. We first develop two basic facts about the optimal solution.

(i) Let w be an internal node of T, and let e_l and e_r be two edges directly below w. If a solution adds non-zero length to both e_r and e_l then it is not optimal.

Proof. Suppose that $\delta_l > 0$ and $\delta_r < 0$ are added to the lengths of e_l and e_r respectively. Let $\delta = min(\delta_l, \delta_r)$. Then the solution which adds $\delta_l - \delta$ and $\delta_r - \delta$ to the lengths of these edges must also have zero skew and uses less total length.

(ii) Let w be a node in T that is neither the root nor a leaf. If a solution increases the length of every path from w to a leaf below w then the solution is not optimal. **Proof**. Since w is not the root, there exists an edge e from w to $w's$ parent. Since every path from w to a leaf has at least one edge increased by some δ_i let $\delta = min_i\delta_i$. Now consider decreasing each path by exactly δ and adding δ to e. The set of path lengths remain unchanged, but the overall total length is reduced implying that the solution is not optimal.

(iii) The greedy algorithm produces the unique optimal solution.

Proof. Consider any other solution and let v be any node of T at which the solution does not add lengths in the way that the greedy algorithm does. We use the notation from the problem and assume without loss of generality that $d_l \leq d_r$. Suppose that the solution adds δ_l to the edge (v, v') and δ_r to the edge (v, v'').

If $\delta_r - \delta_l = d_r - d_l$, then it must be that $\delta_l > 0$ or else the solution would do the same thing as the greedy algorithm: in this case, by (i) it is not optimal. If $\delta_r - \delta_l < d_l - d_r$, then the solution will still have to increase the length of the path from v to each of its leaves in order to make the tree zero-skew; so by (ii) it is not optimal. Similarly, if $\delta_r - \delta_l > d_l - d_r$, then the solution will still have to increase the length of the path from v' to each of its leaves in order to make the tree zero-skew; so by (ii) it is not optimal.

Complexity: Augmenting all the internal nodes to have the longest path lengths take linear time (in the number of leaves) as we need only do an in order traversal. For the remainder of the algorithm, each interval node is visited onces, so again linear time.