

CSc 445: Homework 4 Solutions (rev 2)

April 1, 2008

1. Monthly Software Purchases

Given n distinct growth rates c_1, \dots, c_n , we sort the growth rates into descending order. This can be achieved using, say, Heapsort with a max heap. As we have amply demonstrated previously, such a sort requires time $O(n \log n)$.

Now we must show this yields an optimal cost. Since the structure of this solution is so elementary (it's simply sorted), we can shortcut the analysis by directly showing that a sorted solution must be optimal. This takes the place of the usual optimal-substructure and greedy-choice proofs.

Theorem: A permutation of c_1, \dots, c_n consisting of the growth rates in descending order is an optimal solution (i.e., has a minimum cost).

Proof: (By contradiction) Suppose that $\mathcal{E}^* = (e_1, \dots, e_n)$ is a permutation of c_1, \dots, c_n , and has minimum cost, but does *not* rank the growth rates in descending order. That is, for some index i , with $1 \leq i < n$, $e_i < e_{i+1}$. Let's denote the cost of \mathcal{E}^* with the notation $|\mathcal{E}^*|$, i.e.,

$$|\mathcal{E}^*| = \$100 + \$100 \cdot e_2 + \$100 \cdot e_3^2 + \dots + \$100 \cdot e_n^{n-1} = \$100 \sum_{k=1}^n e_k^{k-1}.$$

If we exchange the order of e_i and its successor, we obtain a modified permutation $\hat{\mathcal{E}}$, and the difference in cost between these two solutions (using the above notation) is

$$|\mathcal{E}^*| - |\hat{\mathcal{E}}| = \$100(e_2 + e_3^2 + \dots + e_i^{i-1} + e_{i+1}^i + \dots + e_n^{n-1} - e_2 - e_3^2 - \dots - e_{i+1}^{i-1} - e_i^i - \dots - e_n^{n-1}) \quad (1)$$

$$= \$100(e_i^{i-1} + e_{i+1}^i - e_i^i - e_{i+1}^{i-1}) \quad (2)$$

$$= \$100(e_i^{i-1}(1 - e_i) + e_{i+1}^{i-1}(e_{i+1} - 1)) \quad (3)$$

$$> \$100 \cdot e_i^{i-1}((1 - e_i) + (e_{i+1} - 1)) \quad (4)$$

$$= \$100 \cdot e_i^{i-1}(e_{i+1} - e_i) \quad (5)$$

$$> 0.$$

Lines (4) and (5) follow because, by hypothesis, e_i is smaller than e_{i+1} , and because each growth rate is greater than unity. Since the difference in costs is positive, this contradicts the assumption that $|\mathcal{E}^*|$ is the minimum cost, which completes our proof. \square

(Pedantic remark: In the name of clarity, line (1) above implicitly assumed that i was more than 3 and less than $n - 1$, but of course this restriction is unnecessary.)

2. Greedy Set Cover Lemma (corrected)

The greedy approach to the given “set cover” problem (which as described seems to be the *dominating set* problem) chooses at each iteration the unchosen vertex adjacent to the *largest* number of uncovered vertices, due to its greed.

As a warmup, we treat the first iteration separately, because it is easiest to understand. In the first iteration, there are n_0 uncovered vertices, and the greedy algorithm therefore takes a vertex w having the highest degree. Let Δ denote the number of vertices covered by selecting w . Note that it is not possible to cover any more than Δ new vertices at each iteration, and so the size k of an optimal cover must be at least n_0/Δ vertices. (If each iteration of an optimal series of choices covered Δ new vertices, then $k = n_0/\Delta$, otherwise $k > n_0/\Delta$.) As stated, in the first iteration of the greedy algorithm, we cover Δ vertices, and so

$$n_1 = n_0 - \Delta \leq n_0 - \frac{n_0}{k}.$$

Now consider the t -th iteration, at which there remain n_t uncovered vertices. At this point, for all we know, the greedy algorithm possibly has been making lousy choices, but we can be sure of one thing: starting from this iteration, it must be possible to cover the remaining n_t nodes in k more iterations (since by the definition of the optimal solution, it is possible to cover all n nodes in k iterations). As a consequence, the average degree of the remaining n_t uncovered nodes must be at least n_t/k . (There is some algebraic detail omitted here for clarity, but available on request.) If this were false, then we would not be able to cover the remaining nodes in k more iterations, which is absurd. This is an average value, and although some uncovered vertices might have degree lower than this average, it is impossible for *all* the uncovered vertices to have below-average degree: there must be at least one uncovered vertex y at or above this average. And that’s the beauty of greed: because our algorithm is greedy, we will on the next iteration choose y (or another with degree at least as high as y ’s) and so the number of uncovered vertices is thus reduced by at least this average amount, or perhaps more, i.e.,

$$n_{t+1} \leq n_t - \frac{n_t}{k}.$$

3. Maintaining Shelters along the Appalachian Trail

We obtain the optimal maintenance schedule by the obvious greedy approach: the first maintenance crew c_1 maintains the first shelter s_1 along with all the shelters (if any) within ten miles of s_1 . The next crew c_2 is responsible for the very next shelter after that, along with all the shelters within the following ten miles, and so on. Given a list of n shelter locations s_1, \dots, s_n we easily could construct a list of crews and crew-to-shelter assignments, but those details are uninteresting.

First we argue that this problem has optimal substructure. Suppose there are n shelters, s_1, \dots, s_n , and an optimal assignment \mathcal{A}^* of m crews c_1, \dots, c_m to maintain them. Let s_i be the last shelter that crew c_1 maintains. Then the rest of the schedule \mathcal{A}^* for shelters s_{i+1}, \dots, s_n must be an optimal subsolution for maintaining (just) shelters s_{i+1}, \dots, s_n ; otherwise, if there were a better subsolution \mathcal{B} for this subset of shelters—one using fewer than $m - 1$ crews—we could join schedule \mathcal{B} with that of the first crew, and thus maintain all the shelters with fewer than m crews, contradicting the optimality of \mathcal{A}^* .

Second, we observe that this problem has the greedy-choice property. Suppose there are i shelters within ten miles of the first shelter s_1 . Our greedy approach is to force the first crew, c_1 , maintain all the shelters s_1, \dots, s_i . No valid solution could ask c_1 to maintain shelter s_{i+1} because it is too far away from the first shelter they service, and presumably the volunteers would perish from exhaustion, or (more likely) quit the trail club and cease providing free labor, both of which would be disastrous. If some optimal solution were to choose shelter s_h instead of s_i as the last shelter under the care of c_1 , with $h < i$, it would be perfectly valid and cromulent to switch responsibility of shelters s_{h+1}, \dots, s_i to crew c_1 , and the solution would not have any greater cost. Thus a greedy choice leads to a solution with cost that is no greater than the optimum.

4. Making Change, Greedily (corrected)

(a) US Currency

Suppose we must make n cents of change. Our algorithm is to do the following: If $n > 0$ then give $q = \lfloor n/25 \rfloor$ quarters. We still owe $n' = n - 25q$ cents of change. If $n' > 0$ then give $d = \lfloor n'/10 \rfloor$ dimes. We still owe $n'' = n' - 10d$ cents of change. If $n'' > 0$ then give $c = \lfloor n''/5 \rfloor$ nickels. We still owe $n''' = n'' - 5c$ cents of change. If $n''' > 0$ then give n''' pennies.

To show optimality, we begin with the greedy choice property. In this situation, that means that when making n cents of change, the optimal solution includes one coin of value x where x is the largest coin value with $x \leq n$. Suppose \mathcal{S}^* is the optimal solution. If \mathcal{S}^* contains x then we are done. We now consider alternative cases:

If $0 < n < 5$ then a solution must consist only of pennies, so \mathcal{S}^* contains x . If $5 \leq n < 10$ then x is a nickel; we have supposed that \mathcal{S}^* lacks nickels, so x is all pennies, and we can replace five pennies with a nickel, improving on the optimum. If $10 \leq n < 25$ then x is a dime; we have supposed \mathcal{S}^* lacks dimes, so x is all pennies and nickels, and some combination of at least two coins can be replaced by a dime, also improving on the optimum. Lastly, if $25 \leq n$ then x is a quarter. But supposing that \mathcal{S}^* lacks quarters, x is thus a combination of pennies, nickels, and dimes. If the number of dimes is two or fewer, then any dimes and additional coins adding up to 25 can be replaced by a quarter. If the number of dimes is three or more, then three dimes can be replaced by a quarter and a nickel. In either case, we improve on the optimum. The above contradictions indicate that any solution lacking a greedy choice is not in fact an optimal solution.

The problem clearly has optimal substructure. If the optimal solution to making n cents of change involves coin y worth z cents, then the remainder of the optimal solution must also be an optimal subsolution to the subproblem of making $n - z$ cents of change; for if it were not and there were a solution using fewer coins, we could combine that solution with coin y to improve on the solution to the problem of making n cents of change, contradicting the notion that we were working with the optimum.

(b) Radix- c coins

A greedy strategy to make n cents of change is to try each of the values $j = k, k-1, \dots, 1$, in turn, stopping as soon as $c^j \leq n$. Then we issue a coin of size c^j and repeat the process next making change for $n - c^j$ cents, of course until the remainder is zero.

In this formulation, the problem still has optimal substructure as argued in part (a). We need only show the greedy choice property. Let $j \leq k$ be the largest value such that

coin c^j no larger than n . Our greedy algorithm would include a coin of denomination c^j in the solution, but suppose (for a contradiction) that there is an optimal solution for n does not include a coin of size c^j . Let a_i be the number of coins of size c^i in the solution, i.e.,

$$\sum_{m=0}^{j-1} a_m c^m = n \geq c^j.$$

If, for any i , $a_i > c$ then we could replace the c coins of denomination c^i , collectively worth $c \cdot c^i = c^{i+1}$ cents, with a single coin of denomination c^{i+1} , and that would be a better solution. However, since we are assuming we already have an optimum solution, we may assume that $a_i \leq c - 1$ already, and so

$$n = \sum_{m=0}^{j-1} a_m c^m \leq \sum_{m=0}^{j-1} (c-1)c^m = (c-1) \frac{c^j - 1}{c-1} = c^j - 1 < c^j.$$

The contradiction $n \geq c^j$ and $n < c^j$ implies that no optimal solution can lack a coin of size c^j , and thus we have proved the greedy choice property.

(c) Bad Examples

One example where greed does not work is if we lost our nickels. If US coins were of denomination 1, 10, 25 then thirty cents change should be three dimes, but a greedy strategy makes for a quarter and five pennies.

Another example is suppose dimes became more costly, and American coins were worth 1, 5, 11, 25. Making change for 22 cents works fine, but 33 cents requires a non-greedy strategy (three 11 cent pieces, obviously), whereas greedy yields a quarter, a nickel, and three pennies.

5 Optimal storage on tapes

Solution Greedy algorithms are broadly based either on selection of a subset or on ordering of a set of elements based on some choice function that is greedy. This problem is a classical example of the ordering paradigm.

The two key things to observe are that tape is rewound every time a movie is to be accessed and each movie is equally likely to be accessed. This means we can write the following formally:

Minimize the access time $t(\pi)$ where,

$$t(\pi) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{k_j}$$

The greedy choice that aims to minimize $t(\pi)$ above, is simply that at any point of time while constructing the permutation π of the movies, pick the movie with the least length on tape to store.

In other words, store the movies in non decreasing order of their lengths. Time to do this is $O(n \lg n)$ to sort the movies and linear time to write them.

Why is this optimal? As in most proofs for proving optimality, we use proof by contradiction by introducing a bad subsequence in our permutation.

Theorem 1 If $l_1 \leq l_2 \leq \dots \leq l_n$, then the ordering $i_j = j, 1 \leq j \leq n$, minimizes

$$t(\Pi) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{kj}$$

over all possible permutations of the i_j .

Proof 1 Let $\Pi = i_1, i_2, \dots, i_n$ be any permutation of the index set $1, 2, \dots, n$. Then,

$$t(\Pi) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{kj} = \sum_{k=1}^n (n - k + 1)l_{ik}$$

If there exist a and b such that $a < b$ and $l_{i_a} > l_{i_b}$, then interchanging i_a and i_b results in a permutation Π' with

$$t(\Pi') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1)l_{ik} \right] + (n - a + 1)l_{i_b} + (n - b + 1)l_{i_a}$$

Subtracting $t(\Pi')$ from $t(\Pi)$, we obtain

$$\begin{aligned} t(\Pi) - t(\Pi') &= (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) \\ &= (b - a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the l_i 's can have minimum t .

6 Finding second smallest number efficiently

Solution We construct a heap of the given elements using up just less than n comparisons. Then we find the second smallest using $\lg n$ comparisons for the `getMax()` operation. Overall, we get $n + \lg n + \text{constant}$.

Alternatively, split the n numbers into groups of 2, perform $n/2$ comparisons successively to find the largest using a tournament-like method.

The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones the maximum pipped. This will yield $\lg n - 1$ comparisons for a total of $n + \lg n - 2$.

7 Basketball Seating

Solution a Assume ALG does *not* admit all groups of size $\leq n$. That is, there is some group g_i of size $\leq n$ that ALG does not admit.

There are two cases to consider. First assume that $g_i \geq n/2$. Then, since the algorithm is greedy, we know that it must have admitted some group *larger* than g_i ; otherwise, group g_i would have been admitted. therefore, we can conclude that the algorithm seats $\geq n/2$ people, as required.

For the second case, assume that $g_i < n/2$. Since g_i is not admitted, we know that at some point $remaining < g_i < n/2$. Since, $remaining$ is non-increasing, we thus conclude that at least $n/2$ people are seated.

This argument implies that if **ALG** admits all groups of size $\leq n$, then it admits exactly the same number of people as the optimal seating algorithm. Second, if **ALG** admits at least $n/2$ people, we know that the optimal seating algorithm can seat at most n people. Hence, $n/2 > k/2$, as required.

Solution b Consider groups $\mathcal{G} = \{(n+2)/2, n/2, n/2\}$. Notice that the greedy seating algorithm admits the group of size $(n+2)/2$, and then cannot admit any of the other groups. The optimal algorithm admits the two groups of size $n/2$, filling all n seats. Taking limit as $n \rightarrow \infty$ we get

$$\lim_{n \rightarrow \infty} \frac{(n+2)/2}{n} = \frac{1}{2}$$

Solution c Assume **ALG2** does *not* admit all groups of size $\leq n$. That is, there is some group g_i of size $\leq n$ that **ALG2** does not admit.

There are two cases to consider. First, assume that $g_i \geq n/2$. Then, since the algorithm first considers all groups of size $\geq n/2$; otherwise, group g_i would have been admitted in the loop when $j = 1$. Therefore, we can conclude that the algorithm seats $\geq n/2$ people, as required. For the second case, assume that $g_i < n/2$. Notice that when $j = \lceil \lg n \rceil$, $g_i \geq n/2^j$. Therefore, if g_i is not admitted, we can conclude that $g_i > remaining$. That is $remaining < g_i < n/2$. Since $remaining$ is non-increasing, we thus conclude that at least $n/2$ people are seated.

As before for part a, this argument implies that the algorithm **ALG2** is factor 2, as desired.

Solution d Algorithm **ALG2** runs in $\mathcal{O}(m \lg n)$ time.

The following algorithm runs in $\mathcal{O}(m)$ with p.r = 2.

```

Fast-Seats( $G[1..m], n$ )
  admitted  $\leftarrow 0$ 
  remaining  $\leftarrow n$ 
  for  $i \leftarrow 1$  to  $m$ 
    if  $G[i] \geq n/2$  and  $G[i] \leq remaining$ 
      Admit( $i$ )
      admitted  $\leftarrow$  admitted +  $G[i]$ 
      remaining  $\leftarrow$  remaining -  $G[i]$ 
  for  $i \leftarrow 1$  to  $m$ 
    if  $G[i] \leq remaining$ 
      Admit( $i$ )
      admitted  $\leftarrow$  admitted +  $G[i]$ 
      remaining  $\leftarrow$  remaining -  $G[i]$ 
  return admitted

```

8a Longest path and Max ST

Solution: Longest path No, Dijkstra's cannot be modified easily to yield the longest path in polynomial time. At least, it is not known if it can be since the hamiltonian path can be reduced in polynomial time to the longest path problem.

More to the point, Dijkstra's employs the greedy method to find the shortest path. This works because the optimal substructure of the problem is always nicely preserved and the subproblems at any point are non overlapping.

Greedy algorithms work on the basis of choosing the locally best available option at any point of time and the structure of the longest path solution does not suit this.

Solution: MaxST Yes. Greedy algorithm for the minimum spanning tree can be converted into maximum spanning tree with a change to the greedy choice function:

Compute the reciprocal of the edge weights and run minimum spanning tree on this. The set of edges chosen corresponds to the maximum spanning tree in the original graph. Correctness follows using the same argument for minimum spanning tree.

The choice of which edge to add is based on the heap of available edges that have not been considered so far. At each point, the algorithm (Kruskal's) maintains the invariant that there is no cycle formed. If the optimal solution consists of some edge with a greater cost compared to the one given by Kruskal's, then it could not have been included at some point of the algorithm since a cycle would be formed. Which means, there is some other edge in the solution in Kruskal's that is not there in the optimal (since this would form a cycle in the optimal solution). If we now add the unique edge that belongs to solution \mathcal{K} but not \mathcal{K}^* to the set of edges \mathcal{K}^* , then we have a cycle. Removal of any edge on the cycle that is created will leave behind some other tree \mathcal{K}' which has a cost no more than \mathcal{K}^* . Hence, \mathcal{K}' is also a minimum (Maximum) cost tree.