CS 677: Assignment 5

Terence Henriod

April 21, 2014

Abstract

In this assignment, various greedy algorithm problems and related ones are presented.

1. A search engine company needs to do a significant amount of computation every time it recompiles its index. For this task, the company has a single large supercomputer, and an unlimited supply of high-end PCs.

   They have broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Lets say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

   Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

   Lets say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly ***El Goog*** can generate a new index.

   Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

   Note: to prove that your greedy strategy yields the optimal solution, you have to prove that the problem has the greedy-choice property.

   Solution: Intuitively, no matter what order we schedule for the jobs to run, the last job must wait for $\sum_{i=1}^{n} p_i$ before finalizing, then it must finalize. It would be a good idea to minimize the completion time of this last job, we should put the shortest finishing time job last (or another way, the job with the longest finishing time first), in order to minimize $\sum_{i=1}^{n} p_i + f_n$. This is somewhat counter intuitive because we disregard the possibility of maximizing the parallelism, but as will be shown, this particular greedy strategy leads to the optimal solution.

   A greedy algorithm that would solve the problem in polynomial time would choose jobs with the longest finishing time first, disregarding the pre-processing times. In order to accomplish this, we can simply sort the jobs using an $n \lg n$ sorting algorithm, and feed the jobs into the system with the sorted order.

   The Algorithm

```
Algorithm INDEX-REBUILD($J(0 \ldots n-1)$)
(* Input: A list/array of jobs for index recompilation *)
1.  RANDOMIZED-QUICKSORT($J, 0, n-1$)
2.  (* This is the same as regular quicksort except instead of comparing elements directly, it only
    compares $J[i].f$s *)
3.  for $i \leftarrow 0$ to $n-1$
4.          feed a job to the super-computer
5.          wait for the job to finish pre-processing
```

```
6.          send the job to a PC after pre-processing to finish
7.    stop (∗ After jobs complete at the PCs, the index is recompiled. ∗)
```

Greedy Choice
This algorithm is greedy because it makes the local optimum choice of selecting the longest finishing time job first, regardless of preprocessing times.

Greedy Choice Property
To first prove the greedy choice of shortest pre-processing time first, consider the completion times c_i for each job, where $c_j = \sum_{i=1}^{n} p_i + f_j$ since, clearly, the pre-processing completion times of each job depend on the jobs that pre-process before them.

Suppose that in the optimal solution the jobs run in some order where the k^{th} job has the longest completion time c_k, and is not first job in the sequence. Also consider the shortest job, identified as the j^{th} job. To keep things simple, consider a set comprised of just a pair of jobs (i.e. $n = 2$). In the "optimal" solution, if the completion time of the job with the longer finishing time is the job with the higher completion time, then the greedy solution improves on it by reducing the completion time of the job with te longer finishing time and still is not any worse because $\sum_{i=1}^{n} p_i + f_j < \sum_{i=1}^{n} p_i + f_k$, so even if the change results in the job with the shorter finishing time becoming the job with the higher completion time, it is still better than the optimal solution. Clearly, the greedy choice is a valid strategy.

In an inductive manner, this logic can be expanded to a set of any size by ordering the jobs by arranging them pair-wise in order to achieve the best pair-wise absolute completion times, and repeating until no improvements can be made, because finding a way to reduce the completion time of onw job does not result in the increase of the completion time of another job in a way that will negate the benefits of the change.

The greedy choice of choosing the job with the longest finishing time first has improved on what we had originally assumed to be the optimal solution (for the purposes of this proof, the above logic could be used to directly prove/derive the *actual* the optimal solution).

Optimal Substructure
As is intuitive, the optimal algorithm for ordering n jobs will be the same algortihm to use to order $n - 1$ jobs after the first one has been selected. This is just a formality to justify our use of a greedy strategy.

Polynomial Time
This algorithm is easily seen to be a polynomial time algorithm. It used a modified efficienct sorting algorithm, like quick sort, which is a $\Theta(n \lg n)$ sorting algorithm. After the sorting, we can feed each of the n jobs in to the computers into the supercompmputer and then into a PC (2 operations for each job), for a cost of $\Theta(2n) = \Theta(n)$ operations. The resulting algorithm will then be $\Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$, which is polynomial.

2. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream I consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a schedule for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

   Now, because you are just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r:

   (*) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.

   Note that this constraint is only imposed for time intervals that start at 0, not for time intervals that start at any other value.

   We say that a schedule is *valid* if it satisfies the constraint (*) imposed by the link.

   The problem. Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

   Example. Suppose we have $n = 3$ streams, with
   $(b_1, t_1) = (2000, 1)$,
   $(b_2, t_2) = (6000, 2)$,
   $(b_3, t_3) = (2000, 1)$,
   and suppose the links parameter is $r = 5000$. Then, the schedule that runs the streams in the order 1, 2, 3 is valid, since the constraint (*) is satisfied:
   $t = 1$: *the whole first stream has been sent, and* $2000 < 5000 * 1$
   $t = 2$: *half of the second stream has also been sent, and* $2000 + 3000 < 5000 * 2$.
   Similar calculations hold for $t = 3$ and $t = 4$.

   (a) Consider the following claim:
   There exists a valid schedule if and only if each stream i satisfies $b_i \leq r * t_i$.
   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

   Solution: The claim is false. It is possible that a stream could violate $b_i \leq r * t_i$ on its own, but be part of a schedule where the average b_{total} is $\leq r * t_{total}$, i.e. $\sum_{i=1}^{n} b_i \leq r * \sum_{i=1}^{n} t_i$. Consider a set of streams being defined by $(b_1, t_1) = (1500, 1)$ and $(b_2, t_2) = (500, 1)$, with $r = 1000$. In this case, stream 1 does not satisfy the $b \leq r * t$ requirement alone, since $1500 > 1000 * 1$. However, we could schedule the jobs so that the stream order is $1, 2$, in which case we have $b_{total} = 500 + 1500 = 2000$ and $t_{total} = 1 + 1 = 2$, giving $2000 \leq 1000 * 2$, which satisifes (*) from the problem statement, and is therefore a valid schedule.

   (b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

   Solution: A greedy choice that would be good for selecting a schedule for this scenario would be to select the stream with the lowest "bit density" first, and continue doing so from the remaining set to determine the order. This problem exhibits an optimal sub-structure because finding a valid way to schedule n streams must also find a valid way to schedule $n-1$ streams so as to keep

$b_{tot} \leq r * t_{tot}$ for each set. The greedy choice is as good or better than the "optimal" algorithm because if an optimal algorithm were to schedule the streams in some valid order without the lowest "bit desnsity" stream being first, making the greedy choice of scheduling the lowest bit density stream first would not alter the average bit density of the schedule, and would in fact result in a schedule that kept the bit density further below the threshold of the condition than would a different ordeering.

The question says to give an algorithm that simply indicates if there exists a valid scheduling. This is almost trivial since all that is required is to total b_is and the t_is and check if $b_{tot} \leq r * t_{tot}$, and if that is satisfied then there exists a valid scheduling. It is of course polynomial since it requires $2 * (n - 1)$ additions and 1 comparison, which is $2n - 1 = \Theta(n)$.

This is boring and not as good as actually determining the valid schedule. A good algorithm for determining the existence of a valid schedule *and* finding such a valid schedule in polynomial time would look like the following:

```
Algorithm FIND-VALID-SCHEDULE(S( 0 ... n - 1 ), r)
(* Input: A list or array of streams *)
1.  (* An array of the "bit densities" of the streams and an identifier for determining the where
    in the order each stream goes *) D(0 ... n - 1)
2.  $density_{tot} \leftarrow 0$
3.  for $i \leftarrow 0$ to $n - 1$
4.      do
5.          $D[i] = S[i].b/S[i].t$
6.          RANDOMIZED-QUICKSORT$(D, 0, n - 1)$
7.  (* Sort the jobs by bit density *)
8.  for $i \leftarrow 0$ to $n - 1$
9.          $density_{tot} = density_{tot} + D[i]$
10.         if $density_{tot} > r$
11.           then
12.                 return false (* There is not a valid schedule of the $n$ streams *)
13. for $i \leftarrow 0$ to $n - 1$
14.         Run stream D[i].identifier
15. return true (* There is a valid schedule of the $n$ streams *)
```

This is a $\Theta(n * 2) + \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$ algorithm.

3. Exercise 16.1-2 (page 422): Suppose that instead of always selecting first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm and prove that it yields an optimal solution.

   Solution: This algorithm works similarly to the typical greedy greedy algorithm for shecduling the maximum number of activities, just in reverse. That is, this algorithm starts at the "end" of the day (as opposed to the beginning of the day), and picks the activity that starts the latest (as opposed to the one that finishes the earliest), which leaves a sub-problem of selecting the next activity from the set of activities that all finish before the selected one starts (as opposed to leaving activites that all start after the recently selected activity finishes).

   This algorithm is a greedy one because is chooses the local optimum choice at each step: the activity with the latest start time in the set of remaining compatible activities is chosen. Further, as we already know, the problem has an optimal sub-structure: the optimal solution to the problem relies on finding the optimal solution to the sub-problems, which arises after every selection of an activity, which, the algorithm asserts is optimal.

   To show that this solution is optimal, consider the set that includes the most compatible activities using the specified algorithm, call it A_{max}, which, of course, is a sub-set of all considered activities S_k. Let a_i be the activity in A_{max} that has the lastest starting time. Suppose we choose the activity with the latest starting time from S_k, call it a_l. There are two cases of this: either a_l is the same activity as a_i, or a_l is not a_i, and therefore has an earlier start time than a_i. The first case is trivial; if a_l is the same as a_i, then clearly the choice has not detracted from the optimal solution (because it was the same choice as the optimal solution), and produces and activity set the same size as the optimal soluiton (because it is the same). In the other case where , however, we could form a new set that must be the same or better than A_{max}, we could call it $A^{'}_{max}$. We know that $A^{'}_{max}$ is the same or better than A_{max} because a_l is compatible with all of the same activities that a_i is (any activtiy that is compatible with a_i must finish before a_i, so since a_l starts later than a_i, clearly the the same activities are compatible with a_l). $A^{'}_{max}$ might even be better because the later start time might allow for more compatible options (it should be noted that this is not possible because one cannot do better than the theoretical optimum solution). In this case, the resulting set will be of the same size or greater than the optimal set.

   In either case, the greedy choice led to a solution set (A_{max} or $A^{'}_{max}$) that was of the same size as the optimal solution, therefore the greedy algorithm is valid and finds the optimal solution.

   Finally, after selecting an activity, due to the optimal sub-structure of the problem, we can repeat the above logic on smaller sets of activities until there are no activities remaining.

4. Exercise 16.3-1 (page 436): Explain why in the proof Lemma 16.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

   Solution: Since x and b are defined as being the letters with the lowest frequency and the highest frequency, respectively, we can consider their frequencies as the endpoints of an interval of numbers, positive integers if we must be precise. All numbers j_i in the interval of counts must then satisfy $x.freq \leq j_i \leq b.freq$. Clearly, the only way for a number to be in this interval if $x.freq = b.freq$ is for $j_i = x.freq = b.freq$. Thus, since $a.freq$ and $y.freq$ are in this interval, they must satisfy $a.freq = b.freq = x.freq = y.freq$.