CS 4820 Supplementary Notes

Dominick Twitty

January 29, 2014

1 How to Answer a Homework Problem

You are given the following problem.

> Katie is a college student with a misguided scheduling idea. She figures the best way to succeed in school is to take as many classes as possible. She has a list of start and end times for all the classes she can take. Given that we only care about her schedule for a single day, give an efficient algorithm that ensures Katie takes as many classes as possible.

(Note that this problem is a disguised version of the Interval Scheduling Problem presented in Section 4.1 of the book and in the lecture on Monday, January 27. The proofof correctness presented below is similar to the one in the book and quite different from the one given in Professor Steurer's lecture and on Piazza.)

1.1 Problem Statement

For purposes of formality, you should restate the problem in mathematical terms. Explicitly state your inputs, assumptions, and goals. This is where you turn concepts into concrete terms. That being said, getting bogged-down in terminology can distract from the intentions of the algorithm. You only need to be formal enough to reason about and prove your thoughts.

1.1.1 Example Problem Statement

We are given a set I of n intervals. For any interval $i \in I$, let $s(i)$ be the start time and $f(i)$ be the end time. We assume that for all i, $s(i)$ and $f(i)$ are integers satisfying $s(i) < f(i)$. We say that two intervals i and j are compatible if the second one starts after the first one finishes, i.e. $\max\{s(i), s(j)\} > \min\{e(i), e(j)\}$, and otherwise we say they *overlap*. We wish to construct a maximum-size schedule $S \subseteq I$ such that no two elements of S overlap.

1.2 Commentary / Building the Algorithm

This optional section is where you might comment on aspects of the problem. This is a great place to discuss how your algorithm was developed. Since the section is optional, you can get a perfect score without including it. On the other hand, if commentary is provided, it can prove useful in preventing the grader from overlooking a subtlety in your algorithm that is crucial for its correctness, or in helping the grader to determine how much partial credit to award in the event your solution isn't fully correct.

1.2.1 Example Commentary

We notice that it is easy to create counterexamples when we choose intervals ordered by start time or length. However, if we order by finishing time, we disregard long intervals in favor of the shorter intervals they overlap.

1.3 Algorithm

Typically, the clearest way to explain an algorithm is in English, accompanied by notation and terminology that you have already defined above. A clear explanation followed by annotated pseudo-code is often effective. Use abstract data types instead of data structures unless the choice of data structure is important to the analysis of the algorithm.
On the other hand, solutions that consist of a long piece of pseudo-code with no accompanying explanation tend to be basically indecipherable by anyone but the author (and usually indecipherable by the author as well, after a few days pass). Moreover, our experience is that solutions like this usually turn out to have inaccuracies that render them incorrect.

1.3.1 Example Algorithm

We use a greedy algorithm that repeatedly selects the interval with the earliest finish time among those that are compatible with the intervals previously selected. In more detail, the algorithm maintains a set S of pairwise compatible intervals (initially empty), and a set P containing all intervals that do not overlap any element of S. As long as P is non-empty, we select the interval in P with the earliest finish time, add this interval to S, and delete all of its overlapping intervals from P. When P becomes empty we return S as our answer.

Algorithm 1 Maximizing Schedule Size

function MAX_SCHEDULE(I)
    $S \leftarrow \emptyset$ ▷ The final schedule
    $P \leftarrow I$ ▷ The set of intervals that do not overlap an element of S
    while $|P| > 0$ **do**
        $x \leftarrow i \in P$ with lowest $f(i)$
        $S \leftarrow S + \{x\}$
        $P \leftarrow \{y \mid y \in P$ and y does not overlap $x\}$
    return S

1.4 Proof of Correctness

All algorithms must be formally proven correct. That is, the output of the algorithm must be proven to match the conditions given in your problem statement. This of course assumes that your problem statement accurately models the original problem. You may use any proof method within reason and do not need to give the axioms behind every step of your proof. However, spacing out your proofs is generally easier to read and debug than tight paragraphs.

1.4.1 Example Proof

We individually prove that the algorithm satisfies each condition of the problem statement.

Lemma 1.1. *At the start and end of every iteration of the while loop, the set P is equal to the set of all intervals that are compatible with every element of S.*

Proof. We prove the lemma by induction. In the base case (the start of the first loop iteration) we have $S = \emptyset$, $P = I$ and the lemma holds vacuously. During each loop iteration when we remove an element x from P and add it to S, we remove all elements of P that overlap x. Note that every element removed from P overlaps an element of S (namely, x) and that every element y remaining in P is compatible with every element of S: y is compatible with x because it was not removed during the present loop iteration, and it is compatible with the other elements of S by the induction hypothesis. Thus, at the end of the loop iteration (and hence also at the start of the next one) it remains the case that P equals the set of all intervals that are compatible with every element of S. □

Lemma 1.2. *No two intervals in S overlap.*

Proof. Let x, y be two intervals in S and suppose that x was added after y. At the start of the loop iteration in which x was added to S, it was the case that x belonged to P and y belonged to S. By Lemma 1.1 this means that x and y do not overlap. □

To prove the optimality of S, we assume that S may not be optimal, and that we have an optimal solution O, and $|O| \geq |S|$. We sort S and O by chronological order (it does not matter if we sort by s or f) and present them as follows:

$$S = s_1, s_2, s_3, \ldots, s_k$$
$$O = o_1, o_2, o_3, \ldots, o_m$$

We first prove that the r^{th} element of S finishes at least as early as the r^{th} element of O, for $r = 1, \ldots, k$.

Lemma 1.3. *For all $r \leq k$, $f(s_r) \leq f(o_r)$.*

Proof. The proof of the lemma is by induction. We take the inequality $f(s_r) \leq f(o_r)$ as our induction hypothesis.
Base case: Our hypothesis holds when $r = 1$ because we greedily chose the interval with earliest finishing time.
Induction step: We assume that $r > 1$ and $f(s_{r-1}) \leq f(o_{r-1})$. Then we have

$$f(s_{r-1}) \leq f(o_{r-1}) < s(o_r) \tag{1}$$

where the second inequality holds because o_{r-1} and o_r are compatible. At the time the algorithm selects s_r, no interval in S finishes later than $f(s_{r-1})$ and hence, by inequality (1), every interval in S is compatible with o_r. Lemma 1.1 now implies that $o_r \in P$ at this time. The greedy algorithm chooses s_r as the interval in P with the earliest finishing time, so $f(s_r) \leq f(o_r)$, which confirms the induction hypothesis and completes the proof. □

Lemma 1.4. *S is an optimal solution. That is, $|S| = |O|$.*

Proof. The proof is by contradiction. Assume, to the contrary, that O contains at least $k+1$ elements. Using inequality (1), we know that $f(s_k) < f(o_{k+1})$. Since no element of S finishes later than $f(s_k)$, it follows that o_{k+1} is compatible with every element of S. By Lemma 1.1, we may conclude that o_{k+1} belonged to P at the end of the final loop iteration, contradicting the loop's termination condition. □

1.5 Running Time Analysis

Every algorithm must be analyzed for time complexity. Try to bound the running time as tightly as possible. When asked to design an algorithm always assume that it must run in polynomial time, unless otherwise directed. In general, we will not deduct points for an algorithm with suboptimal running time, as long as the running time is still polynomial in the input size and your running time bound is as tight as possible *for your own algorithm.*

For instance, it is possible to solve the Interval Scheduling problem in time $O(n \log n)$ but the algorithm presented above runs in time $O(n^2)$. The example analysis given below, which proves the running time bound of $O(n^2)$, would be considered perfect. A bound of $O(n^3)$ would receive a minor deduction because it is not tight, and a bound of $O(n \log n)$ would receive a more significant deduction (unless the algorithm was modified) because it would be invalid.

1.5.1 Example Analysis

There are at most n iterations of the while loop because the size of P strictly decreases with each iteration. During an iteration we must perform a minimization over the elements of P (requiring $O(n)$ time), add an element x to S in $O(1)$ time, and test every remaining element of P to see whether it overlaps with x. Testing whether two intervals overlap requires $O(1)$ time and P has at most n elements, so finding and removing the elements that overlap with x takes $O(n)$. In total, then, the algorithm performs at most n loop iterations with $O(n)$ work per iteration, resulting in a running time of $O(n^2)$.