Intensive Algorithms

CPSC 366, YALE UNIVERSITY, SPRING 2019

These are lecture notes for CPSC 366b, "Intensive Algorithms," taught by Yang Cai at Yale University during the spring of 2019. These notes are not official, and have not been proofread by the instructor for the course. These notes live in my lecture notes respository at

https://github.com/jopetty/lecture-notes/tree/master/CPSC-366.

If you find any errors, please open a bug report describing the error, and label it with the course identifier, or open a pull request so I can correct it.

Contents

1 January 15, 2019

1.1 Administrata

Started off with basic administrata. Differences between 365 and 366 (pretty much the same but with proofs and a bit more sophisticated material). You probably know what you're getting into. The no collaboaration policy is terrible :(

We'll cover the following topics,

1. Greedy Algorithms
2. Dynamic Programming
3. Divide and Conquer
4. Network Flow
5. Linear Programming
6. NP and NP-completeness
7. Approximation Algorithms
8. *Heuristics*
9. *Randomized Algorithms*
10. *Online Algorithms*

where the italicized items are things we will cover if time permits.

1.2 Greedy Algorithms

A greedy algorithm is one which locally make some optimal choice and stick with it. The important part is defining what "optimal" means. This involves choosing criteria and proving correctness with it.

> **Problem 1.1** (Interval Scheduling)**.** Say you have a conference room with many requests to use it, and you want to satisfy as many of these possibly overlapping requests as possible. The input to this problem is a list of intervals $\{(s_1, f_1), \cdots, (s_n, f_n)\}$ where the i^{th} interval starts at s_i and ends at f_i, so clearly $s_i < f_i$. Intervals i and j overlap if $s_j > f_i$.

```
procedure GREEDY(C)
    A ← ∅
    Sort C
    for c ∈ C do
        if c has no conflicts in A then
            A = A ∪ c
```

The most important question here is "how do we order the intervals?" One way we might do this is by ordering each by the start time, but then we are penalized if there is one

really long interval that starts first. So instead, we could sort the intervals based on the duration. However, if we have a short interval which overlaps two longer intervals, so this is also problematic. We could sort it based on the fewest number of conflicts, but this is also problematic if we have identical overlapping intervals.

The best option is to sort the intervals based on when the *end*. This is intuitive since we want to free up our resource as quickly as possible.

Problem 1.2 (What is the running time of this algorithm?)**.** There are three main steps: the time to sort the list of intervals, which can easily be done in $O(n \log n)$ time; the comparison to other intervals in A, which can be done in $O(1)$ time since the list is already ordered and if an interval conflicts with any already in our ansewr it must conflict with the last one added since for all $j > i$ we know that $f_i \leq f_j$; and adding the interval to the list, which is constant time as well. We do these last two steps n times, so the total run time is $O(n)$ for these parts. Then the whole algorithm runs in $O(n \log n)$ time.

Problem 1.3 (How do we know this algorithm is correct?)**.** Suppose that the intervals are already sorted based on the finish time. Then $f_1 \leq f_2 \leq \cdots \leq f_n$.

Theorem 1.1. *The algorithm outputs a maximal set of non-overlapping intervals.*

You can try to prove that this algorithm find the optimal answer, but it's quite easy to find cases where optimal solution may not be unique (see the identical overlapping intervals from previous). Instead, we can argue that this algorithm is better than any other by showing that it is ahead of any other algorithm.

Theorem 1.2. *Let $A = \{a_1, \cdots, a_k\}$ where $a_i < a_j$ by finish time be the output of our algorithm, and let $B = \{b_1, \cdots, b_\ell\}$ be any sorted other list of non-overlapping intervals (the output of some other algorithm). Then $a_i \leq b_i$ for all i.*

Proof our algorithm stays ahead. We proceed by induction on $j \in \{1, \cdots, \min\{k, \ell\}\}$. We know that a_1 finished first, so b_1 must finish no sooner than a_1, so the claim is true for $j = 1$. For the inductive step, assume that the claim is true for some j. Then $f_{a_j} \leq f_{b_j} \leq s_{b_{j+1}}$. Then b_{j+1} does not overlap with any interval $a_1, \cdots, a_j$, so our algorithm could add b_{j+1} to A (there may be a strictly better choice for us, but we know that we *can* add it). Then $f_{a_{j+1}} \leq f_{b_{j+1}}$. Then by induction the claim is true for all j. ■

Proof of correctness. By way of contradiction, assume that $k < \ell$. By our previous proof, we know that $a_i < b_i$ for all i. Then $f_{a_1} \leq a_{a_2} \leq \cdots \leq f_{a_k} \leq f_{b_k} \leq s_{b_{k+1}}$. This is a contradiction, since our algorithm would have added to b_{k+1} to A. ■

Problem 1.4. What if we tweak the problem to have a payoff associated with it and we want to maximize the payoff? Our greedy algorithm (in fact, *any* greedy algorithm) won't work (but we do know one which uses dynamic programming).