COMPSCI 330: Design and Analysis of Algorithms September 19

Lecture 7: Greedy Algorithms II

Lecturer: Rong Ge *Scribe: Rohith Kuditipudi*

1 Overview

In this lecture, we continue our discussion of greedy algorithms from Lecture 6. We demonstrate a greedy algorithms for solving *interval scheduling* and *optimal encoding* and analyze their correctness.

Although easy to devise, greedy algorithms can be hard to analyze. The correctness is often established via proof by contradiction. We demonstrate greedy algorithms for solving *fractional knapsack* and *interval scheduling* problem and analyze their correctness.

2 Interval Scheduling*

*repeated from Lecture 6

Suppose there are n meeting requests, and meeting i takes time (s_i, t_i)—it starts at s_i and ends at t_i. The constraint is that no two meeting can be scheduled together if their intervals overlap. Our goal is to schedule as many meetings as possible.

Example 1. Suppose we have meetings $(1,3), (2,4), (4,5), (4,6), (6,8)$ that look like this:

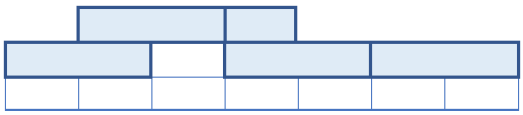

Figure 1: Meetings' intervals.

We should think of intervals as open; that is, $(2,4)$ does not overlap with $(4,5)$. The solution to this instance to schedule three meetings, *e.g.*, $\{(1,3), (4,5), (6,8)\}$. Of course, there can be more than one solutions.

2.1 Algorithm

Let us again follow our guideline for designing greedy algorithms.

(i) We ask first what decisions one needs to make. The answer here is kind of trivial. Clearly, we want to schedule intervals.

(ii) What is the "best" local option? In particular, what is the first meeting to schedule?

Intuitively, I would like to earlier meetings to start with, and the earlier the better. But it is not clear what we mean by "early". Consider two meetings $(1, 4)$ and $(2, 3)$. The first starts early and ends late. It is natural to conclude that one should schedule $(2, 3)$ because it ends early and allows me to schedule other meetings starting at 3 onwards. If I scheduled $(1, 4)$, I could only start another meeting at 4. Ending time is what affects future. (Here, we emphasize that the length of the interval is irrelevant. We simply want to maximize total number of scheduled meetings.)

Once you make the first decision, you can use the greedy rule repeatedly to find a solution. Hence, we have the algorithm:

Algorithm: always try to schedule the meeting with the earliest *ending* time.

It is simple to implement the algorithm. One starts by sorting all intervals by their ending times in ascending order. Then scan the intervals from the one with the earliest ending time, try to schedule the current interval, and if there is a conflict, then skip this interval.

2.2 Analysis

Running time. It is again easy to see that the algorithm runs in $O(n \log n)$ time, since the most expensive step is sorting.

Correctness. We shall focus on analyzing the correctness of this algorithm by first understanding the algorithm more concretely.

Example 2. Let us consider Example 3 again and see what our algorithm will do.

(1) Clearly, $(1, 3)$ is the earliest ending meeting, and the algorithm schedules it.

(2) Then the algorithm looks at $(2, 4)$ and skips it, as it conflicts with $(1, 3)$, which is already scheduled.

(3) It then checks $(4, 5)$, and since there is no conflict, it is scheduled.

(4) Next, it looks at $(4, 6)$, and that conflicts with $(4, 5)$ and thus gets skipped.

(5) Finally, the algorithm checks out $(6, 8)$ and schedules it.

We see indeed the algorithm schedules three meetings, producing an optimal solution.

Let us now try to prove the algorithm. The idea is fairly simple. Suppose OPT schedules more meetings than ALG. Then at the first point where OPT disagrees with ALG, it must have scheduled a meeting that ends later than the one ALG schedules. Whatever OPT did afterwards, the algorithm will do no worse than that.

Theorem 1. The algorithm described in Section 4.1 always produces an optimal solution for the interval scheduling problem.

Proof. Assume algorithm schedules k meetings $\mathsf{ALG} = (i_1, i_2, \ldots, i_k)$. Further, suppose for a contradiction that OPT is a better solution producing $t > k$ meetings, and they are $\mathsf{OPT} = (j_1, j_2, \ldots, j_t)$. Let both solutions solutions be sorted in starting time; that is, $s_{i_1} < s_{i_2} < \ldots < s_{i_k}$ and $s_{j_1} < s_{j_2} < \ldots < s_{j_k}$.

Let p be the first meeting where $i_p \neq j_p$. By the design of the algorithm, we have that i_p ends before j_p, and therefore i_p ends before j_{p+1} starts. Hence, the new solution

$$\mathsf{OPT}' = (i_1, i_2, \ldots, i_p, j_{p+1}, j_{p+2}, \ldots, j_t) \tag{1}$$

is also a valid schedule that still schedules t meetings, as many as OPT does. This is similar to the proof of Theorem **??**—OPT' is now closer to ALG. Again, one may repeat this argument. Eventually, we get to an OPT' where $i_p = j_p$ for all $p \leq k$, and the schedules would look like following

$$\mathsf{ALG} = (i_1, i_2, \ldots, i_k)$$
$$\mathsf{OPT}' = (i_1, i_2, \ldots, i_k, j_{k+1}, \ldots, j_t).$$

The first k meetings are exactly the same, but OPT' continues to schedule $j_{k+1}, \ldots, j_t$ after i_k. This contradicts the design of the algorithm because if j_{k+1} is still available to be scheduled at the end of ALG, the algorithm would schedule it. □

3 Huffman Coding

Huffman coding is a particular kind of data-compression algorithm. The focus of this section will be to rederive Huffman coding as a greedy algorithm.

3.1 Problem Statement

Given a long string comprised of characters from an alphabet of size n, find a way to encode these characters into binary codes such that the length of the binary representation of the string is minimized.

Example 3. Suppose we are trying to encode the string"aababc". Note that in this example, $n = 3$ and the alphabet is $\{a, b, c\}$. If we choose our encoding such that $a = 0$, $b = 11$ and $c = 10$, then we can represent the string "aababc" in binary as 001101110. Note that since our goal is to minimize the length of this binary representation, we assigned shortest code to the character that appeared most often in the string (i.e. $a = 0$).

What kinds of encodings are allowed?

Really the only requirement is that our encoding be completely unambigious. In other words, we want an encoding with a *unique* decoding. To illustrate this point, consider the following example of a BAD encoding: $a = 0$, $b = 01$ and $c = 1$. Using this encoding, it's impossible to know whether 01 should be decoded as "ac" or just "b".

The problem with this encoding is that it is not *prefix-free*. Specifically, the encoding for a is a prefix of the encoding for b. Fortunately, so long as our encoding is prefix-free, we will always be able to encode/decode strings with no information loss (i.e. no ambiguity).

3.1.1 Prefix-free Encodings as Binary Trees

Binary trees are a natural way to represent binary encodings. As shown in Figure 2, the encoding for each character in the alphabet is specified by the path from the root of the tree to the character's

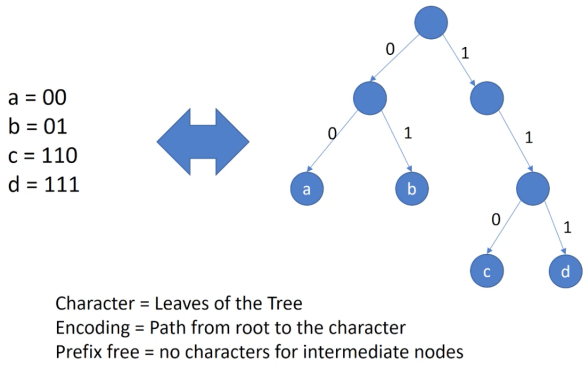

Figure 2: A tree representation of a binary encoding scheme (taken from Lecture 7)

node. So long as only leaf-nodes correspond to characters, the binary tree will represent a prefix-free encoding.

Note that the length of the path from the root to the leaf node corresponding to character i is equal to the length of the encoding for i. Thus, given a binary tree representation of an encoding, we can define the *cost* of the tree as $\sum_i d_i f_i$, where d_i denotes the depth of node i and f_i denotes the number of times character i appears in the string we wish to encode. Note that the cost of a tree is equivalent to the length of the corresponding binary encoding of our string of interest.

3.2 Greedy Algorithm

In Section 3.1.1, we defined a cost function on binary trees and demonstrated an equivalence between this cost function and our original objective–to minimize the length of the binary encoding of a particular string. At a high level, our algorithm will solve the encoding problem by specifying a procedure for greedily constructing a binary tree with minimal corresponding cost.

3.2.1 Algorithm Description

Our algorithm builds a single binary tree from a forest of n disjoint nodes, each corresponding to a particular character in the alphabet, through a sequence of $n-1$ merge operations. The algorithm is greedy in the sense that it seeks to minimize the immediate cost of each merge operation.

Specifically, let S denote the set of root nodes at any point during the algorithm (note that $|S| = n$ at first). Let f_i and f_j denote the frequencies of characters i and j in a string of interest s. *Merging* nodes i and j consists of adding a new node ij to S such that i and j are both children of ij. Moreover, define the *cost* of merging i and j to be equal to f_{ij}, where $f_{ij} = f_i + f_j$. Merging i and j amounts to increasing the depth of both nodes (a.k.a. increasing the length of the binary encoding of both i and j by one). Thus, intuitively the cost of each merge operation is meant to capture the resulting change in the length of our binary encoding of s. As shown in Figure 3, at each step our algorithm finds the two elements of S with the lowest associated frequency values and merges these elements together. From start to finish, our algorithm will first merge a and c, then merge ac and d, and finally merge acd with b.

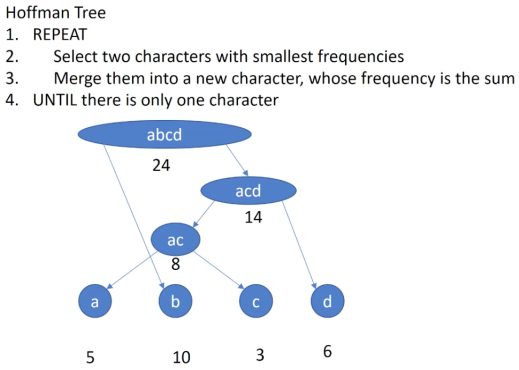

Figure 3: Building the optimal Huffman tree (figure taken from Lecture 7)

3.2.2 Proof of Correctness (By Induction)

Let n denote the size of the alphabet.

Base Case (n=2): algorithm will assign 0 to one character and 1 to the other

Induction: Assume our greedy algorithm returns an optimal solution for n. Now suppose we wish to apply our algorithm to a problem of size $n+1$. Note that once we decide which two nodes to merge first, we will have reduced our original problem to one of size n. Thus, in order to prove the correctness of our algorithm for problems of size $n+1$ we simply need to demonstrate that our algorithm will not initially pick the wrong two nodes to merge.

To do so, let i and j denote the two characters with the lowest frequencies. Note that by definition, the first step of our greedy algorithm will be to merge i and j, meaning that i and j must be neighbors in the Huffman tree returned by our algorithm. Furthermore, let OPT be any Huffman tree in which i and j do not both belong to the same parent node and without loss of generality, suppose that the depth of i is at most the depth of j in OPT. From the fact that i and j are the two characters with the lowest frequencies, it follows that swapping i with the sibling of j in OPT will not increase the cost of OPT. In fact, swapping either i or j and any node with greater depth in OPT will not increase the cost of OPT, implying that one can always construct an optimal Huffman tree by first merging i and j.

3.2.3 Runtime Analysis

Assuming use a data structure such as a heap to keep the nodes sorted in order of frequency at all times, the algorithm will perform $n-1$ merge operations. After each merge operation we will need to maintain the integrity of the heap by performing a "swim" operation (covered in CS 201), each of which will take $\Theta(\log n)$ time. Thus, the overall runtime of the algorithm will be $\Theta(n \log n)$. If we do not use a data structure such as a heap, then before each merge operation we will need to iterate through every node in order to find the two with the lowest frequencies and so the runtime of our algorithm will worsen to $\Theta(n^2)$.

4 Summary

We have discussed the greedy paradigm in algorithm design and showed that it gives algorithms that solve interval scheduling and Huffman coding problems.