CSC 373: Algorithm Design and Analysis
Lecture 5

Allan Borodin

January 16, 2013

Some pictures are from Jeff Erickson's lecture notes.

Lecture 5: Announcements and Outline

- Announcements
  1. I will now provide the password for Allan Jepson's lecture notes.
  2. If you have any intention of applying for a USRA, I believe the deadline in this Friday. This is worth doing!
  3. There is a lecture this Friday and then a tutorial on Monday

- Outline for today
  1. Finish up Huffman coding
  2. Greedy algorithms for the makespan problem
  3. Reviewing the greedy algorithm paradigm

Review: Prefix binary codes as binary trees

- Such an encoding is equivalent to a full ordered binary tree T; that is, a rooted binary tree where
  - Every non leaf has exactly two children
  - With the left edge say labeled 0 and the right edge labeled 1
  - With every leaf labeled by a symbol in Γ
- Then the labels along the path to a leaf define the string encoding the symbol at that leaf. The goal is to create such a tree T so as to minimize

$$cost(T) = \sum_i f_i \cdot (\text{depth of } s_i \text{ in } T)$$

- Equivalently we are minimizing the expected symbol length, namely

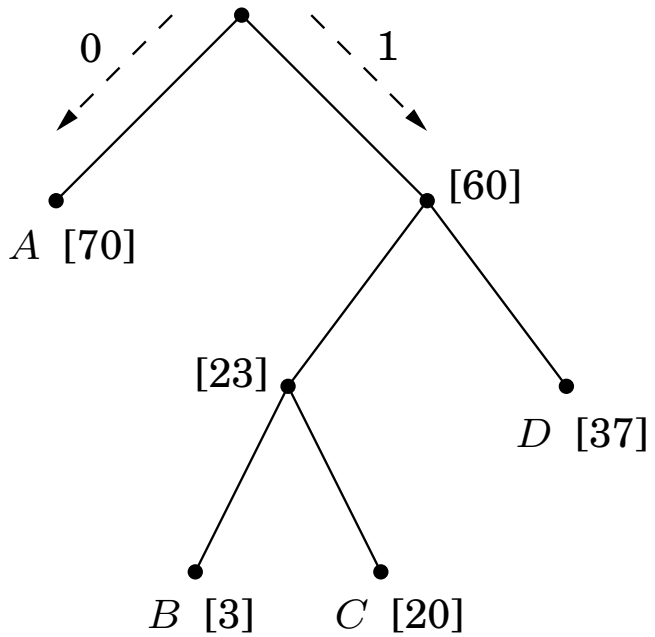
$$\mathbf{E}_{s\in\Gamma}[|\sigma(s)|] = \sum_i p_i \cdot (\text{depth of } s_i \text{ in } T)$$

  where $p_i = \frac{f_i}{\sum_i f_i}$ is the probability of s_i.
- The intuitive idea is to greedily combine the two lowest frequency symbols s_1 and s_2 to create a new symbol with frequency $f_1 + f_2$.

An example of Huffman coding in DPV

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Codeword
A	0
B	100
C	101
D	11



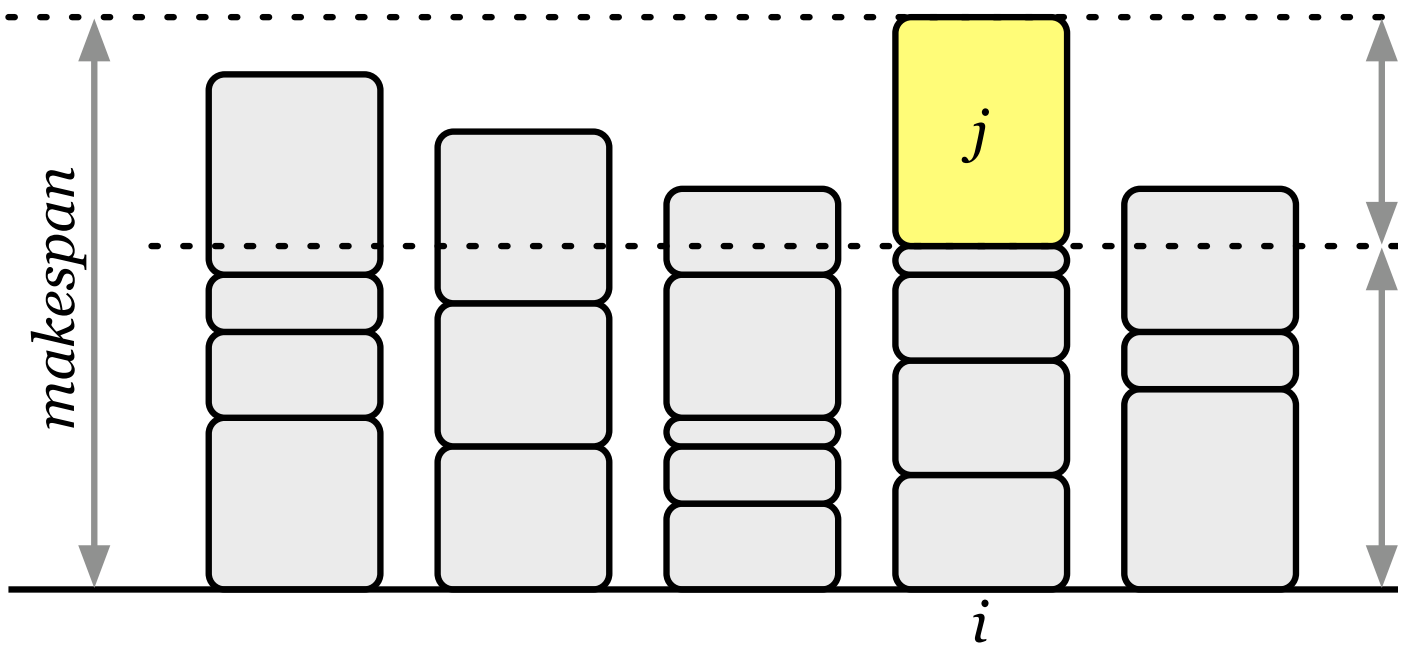
The Huffman algorithm as in DPV text

Code for Huffman coding

Procedure *Huffman*($\mathbf{f}$)
Input: An array $\mathbf{f}[1 \ldots n]$ of frequencies with $f_1 \leq f_2 \ldots \leq f_n$
Output: An encoding tree with n leaves
Let H be a priority queue of integers, ordered by $\mathbf{f}$
For $i : 1, \ldots, n$
  $insert(H, i)$
For $k : n+1, \ldots, 2n-1$
  $i = deletemin(H)$, $j = deletemin(H)$
  create a node numbered k with children i, j
  $f[k] = f[i] + f[j]$
  $insert(H, k)$

The makespan problem

- The input consists of n jobs $\mathcal{J} = J_1, \ldots, J_n$ that are to be scheduled on m identical machines.
- Each job J_k is described by a processing time (or load) p_k.
- The goal is to minimize the latest finishing time (maximum load) over all machines.
- That is, the goal is a mapping $\sigma : \{1, \ldots, n\} \to \{1, \ldots, m\}$ that minimizes $\max_k \left(\sum_{\ell:\sigma(\ell)=k} p_\ell \right)$.



Online greedy algorithm for makespan

- Suppose we think of the jobs coming in as a stream of jobs $J_1, J_2, \ldots$.
- An online algorithm must assign each job immediately to a machine before the next job arrives.

Graham's online greedy algorithm for makespan

- Consider input jobs in the order as they arrive in an online setting
- Schedule each job J_j on any machine having the least load thus far.

Graham's online greedy algorithm for makespan

- Consider input jobs in the order as they arrive in an online setting
- Schedule each job J_j on any machine having the least load thus far.

- We will see that the **approximation ratio** for this algorithm is $2 - \frac{1}{m}$ for all $m > 1$.
- That is, for any sequence of jobs $\mathcal{J}$,

$$Greedy(\mathcal{J}) \leq (2 - \frac{1}{m})OPT(\mathcal{J}).$$

  - *Greedy* denotes the makespan (i.e. the cost) of the above greedy algorithm.
  - *OPT* stands for the cost of any (say, optimal) solution.

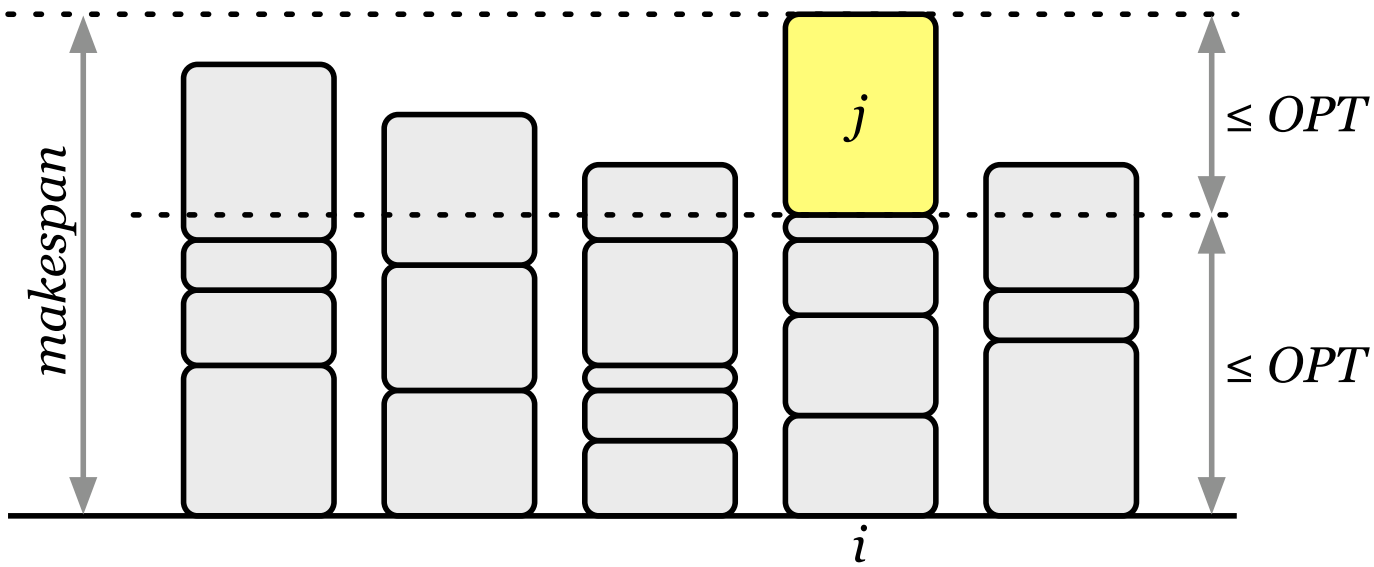
Graham's online greedy algorithm for makespan

- Consider input jobs in the order as they arrive in an online setting
- Schedule each job J_j on any machine having the least load thus far.

Claim

- The **approximation ratio** for this algorithm is $2 - \frac{1}{m}$ for all $m > 1$.
- That is, for any sequence of jobs $\mathcal{J}$, $Greedy(\mathcal{J}) \leq (2 - \frac{1}{m})OPT(\mathcal{J})$.

- **Basic proof idea:**



The proof

- The proof for the approximation follows the approach we used in the interval colouring result.

We will establish some simple "intrinsic bounds" that any solution must satisfy and then analyze the greedy solution in terms of the following intrinsic bounds.

- $OPT(\mathcal{J})$ must be at least

$$B_1 = \max\{p_1, \ldots, p_n\},$$

where p_i is the processing time (load) of J_i.

- $OPT(\mathcal{J})$ must be at least the average machine load

$$B_2 = \frac{(p_1 + \ldots + p_n)}{m}.$$

The proof (continued)

Claim

For any sequence of jobs $\mathcal{J}$, $Greedy(\mathcal{J}) \leq (2 - \frac{1}{m})OPT(\mathcal{J})$.

- Consider the job that completes last defining the makespan.
- Without loss of generality we can say this is the nth (i.e. last) job.
- Consider the assigned machine just before the assignment.

1. Its load is at most the average load of previous jobs, that is, $B_2 - \frac{p_n}{m}$.
2. After adding p_n to the load, the makespan becomes

$$Greedy(\mathcal{J}) \leq B_2 + \left(1 - \frac{1}{m}\right) p_n \leq B_2 + \left(1 - \frac{1}{m}\right) B_1$$

3. Hence, the greedy makespan $Greedy(\mathcal{J}) \leq \left(2 - \frac{1}{m}\right) OPT(\mathcal{J})$ □

Exercise for your interest

Suppose p_i stands for the *load* and jobs are temporary and only present in some time interval $[t_i^1, t_i^2]$. The goal is to minimize the makespan at every point of time.

Why study proofs? (again)

- Looking at this proof we can see what seems to be causing the biggest gap between an optimal assignment and that of the online greedy algorithm.
- Namely, a job that maximizes the load could be the last job defining the makespan.
- While this doesn't show that the bound is tight, we do have the following tight example:

  - Let the first $m(m-1)$ jobs have unit load while the last job has load $p_n = m$.
  - Then *Greedy* spreads the unit jobs evenly over the m machines (each machine then having load $m-1$) and then is stuck adding p_n to some machine. This forces the makespan to $2m-1$.
  - *OPT* spreads the unit jobs over $m-1$ machines so that it can achieve the makespan m.

The LPT makespan algorithm

- The proof and the tight example suggest a different (not online) greedy algorithm.

Sort the jobs so that the largest come first (and hence the name LPT for longest processing time).

- It can be shown (although we will not do that now) that the approximation ratio for the LPT makespan algorithm (on m identical machines) is $\left(\frac{4}{3} - \frac{1}{3m}\right)$.

- One can also achieve a somewhat better online approximation ratio by not being entirely greedy.

Summarizing the greedy paradigm

- Informally, (most) greedy algorithms consider one input item at a time and make an irrevocable ("greedy") decision about that item before seeing more items.
- To make this precise for any given problem we have to say
  1. how input items are represented
  2. how an algorithm determines the order in which input items are considered.
- Mainly, we need to define the class of orderings of the input items that will be allowed. We cannot allow any ordering of the input set or else one can say take exponential time to compute an "optimal ordering".
- If we say the ordering must be done in say time $O(n \log n)$ (or even $poly(n)$) then we are in the situation of trying to prove that something cannot be done in a given time bound.

One way to formalize how to order

- For a given problem, assume that input items belong to some set $\mathcal{J}$.
- For any execution of the algorithm, the input is a finite subset $\mathcal{I} \subset \mathcal{J}$.
- Let $f : \mathcal{J} \to \Re$ be any function; that is, we do not place any restriction on the complexity or even the computability of the function.
- Then for any actual input set $\mathcal{I} = \{I_1, \ldots, I_n\}$, the function f induces a total order on the input set (where we can break ties using the index of the input items as given).
- In a fixed order the function f is set initially. In an adaptive order, there can be a different function f_i in each iteration i with f_i depending on the items considered in iterations $j < i$.

Jeff Erickson's comment on greedy algorithms

5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well. . . hardly ever.[6]

My view of greedy algorithms

- First, the previous comments are in the context of emphasizing DP algorithms and hence were a deliberate overstating of the point.
- My view of greedy algorithms is that while they may rarely be optimal or as good as more sophisticated algorithms, there are many cases where they work well either in terms of provable approximations or "in practice".
- Moreover, in some cases we imediately need something that works and knowing some basic approaches to a problem becomes a starting point. If nothing esle, greedy algorithms can be a benchmark for comparison against more sophisticated algorithms.
- DP algorithms, once they are formulated, often seem quite apparant. But coming up with a correct DP formulation is often not so obvious. In contrast, coming up with a correct (albeit possibly having poor performance) greedy algorithm is usually easy to do.
- Finally, there are applications (e.g. auctions) where conceptual simplicity is a virtue in itself and to some extent conveys a sense of "fairness".