Greedy Algorithms

Adnan Aziz

Based on CLRS, Ch 16.

Algorithms for optimization usually go through a sequence of steps with choices at each step.

In DP all/some choices are explored. In the greedy algorithms, only once choice is made, one that looks best. This approach is not always optimum, but in some special cases it works.

Example of the greedy method: Activity Selection Problem

$S = \{a_1, a_2, \ldots, a_n\}$ activities which use a single shared resource (lecture hall, CPU, network connection, etc.).

Activity a_i has a start time s_i and finish time f_i, $0 \leq s_i < f_i < \infty$.

If selected, a_i takes places in interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.

The *Activity Selection Problem* (ASP) is to find a maximum-sized subset of S in which all activities are compatible.

For example, if $S = \{(1,4), (3,5), (0,6), (5,7), (3,8), (5,9), (6,10), (8,11), (8,12), (2,13), (12,14)\}$ then $\{a_3, a_9, a_{11}\}$, $\{a_1, a_4, a_8, a_{11}\}$, and $\{a_2, a_4, a_9, a_{11}\}$ are compatible.

DP springs to mind—if we select a_k, find an optimum ACS for $\{a_i \mid f_i \leq s_k\}$, and for $\{a_j \mid f_k \leq s_j\}$.

Notation: Let $S_{ij} = \{a_k \in S \mid f_i \leq s_k \text{ and } f_k \leq s_j\}$, i.e., S_{ij} is the set of activities that start after a_i finishes and finish before a_j start.

Introduce dummy activities a_0 and a_{n+1} with $f_0 = 0$ and $s_{n+1} = \infty$.

Let $c[i, j]$ be the number of activities in max-sized subset of mutually compatible activities in S_{ij}.

Assume $a_1, \ldots, a_n$ are sorted by finish time, so $c[i, j] = 0$ for $i \geq j$.

Now if a_k is selected, $c[i, j] = c[i, k] + c[k, j] + 1$ (assumption is that $i < k < j$). So the recursion is

$$\begin{aligned} c[i, j] &= 0 \text{ if } S_{ij} = \emptyset \\ &= \max_{i<k<j, a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} \text{ if } S_{ij} \neq \emptyset \end{aligned}$$

Straightforward to solve in time $\Theta(n^3)$: $\Theta(n^2)$ entries, each takes $\Theta(n)$ time to fill in.

Can do better. Use the following:

Theorem 1 *If a_m is activity in S_{ij} with minimum finish time, i.e., $f_m = \min\{f_k \mid a_k \in S_{ij}\}$, then*

1. *a_m is used in some maximum sized subset of mutually compatible activities in S_{ij}, and*
2. *S_{im} is empty.*

Claim 2: holds because we chose a_m to have smallest finish time.

Claim 1: consider any max sized subset A_{ij} of mutually compatible activities and swap the element with smallest finishing time in A_{ij} with a_m. The set is still compatible, and of the same size.

Results in the following algorithm: choose activity with the smallest finish time, recurse on remaining compatible activities.

Takes $\Theta(n)$ time if the a_is are sorted in order of finish time; otherwise $\Theta(n \cdot \lg n)$.

Comments

Greedy does not always result in optimum solution: choosing the action with start time/least duration/least overlap with other actions would not have lead to optimum solution to ASP.

0-1 knapsack problem

n items, each with value v_i and weight w_i. Can store no more than W pounds in the knapsack. What items to take?
fractional knapsack problem: n items, each with value v_i and weight w_i. Can store no more than W pounds in the knapsack; fractional quantities are acceptable. What items to take?
Both versions have optimum substructure property:

- 0-1: If we take item j, remaining solution is optimum for $W - w_j$ weight and items $\{1, \ldots, j-1, j+1, \ldots, n\}$.
- fraction: Take w of $j \Rightarrow$ take optimum for $W - w$ from remaining $n-1$ items + $w_j - w$ pounds of j.

However the first cannot be solved using a greedy approach: consider 3 items with values $60, 100, 120$ respectively, and weights $10, 20, 30$, respectively. Knapsack capacity is 50.

0-1 Greedy approach would select Item 1 (highest value-to-weight ratio); then regardless of whether Item 2 or 3 was chosen, the resulting solution would be inferior to choosing just Items 2 and 3.

Fraction Greedy approach would select Item 1, then Item 2, then 20 pounds of Item 3; this is optimum.

Huffman codes

Huffman coding: data compression. Exploits the fact that some characters are more common than others.

Example—data consists of 100,000 character file. Suppose the only characters are a, b, c, d, e, f, and that the number of occurrences of each is given in the table below.

char	a	b	c	d	e	f
count (1000s)	45	13	12	16	9	5
fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

The fixed length coding requires $300,000$ bits; the variable length coding uses 224,000 bits.

Can avoid ambiguity in decoding by requiring that no codeword is a prefix of another code work.

Given codebook, it's trivial to encode. For decoding, note there's only one way to recover a code word from the start of the file; get this word, and apply the process to the remainder of the file.

E.g., 001011101 decodes to $0 \cdot 0 \cdot 101 \cdot 1101 = aabe$.

Can represent codebook by a binary tree whose leaves are given characters. Corresponding codeword is given by path from root to leaf.

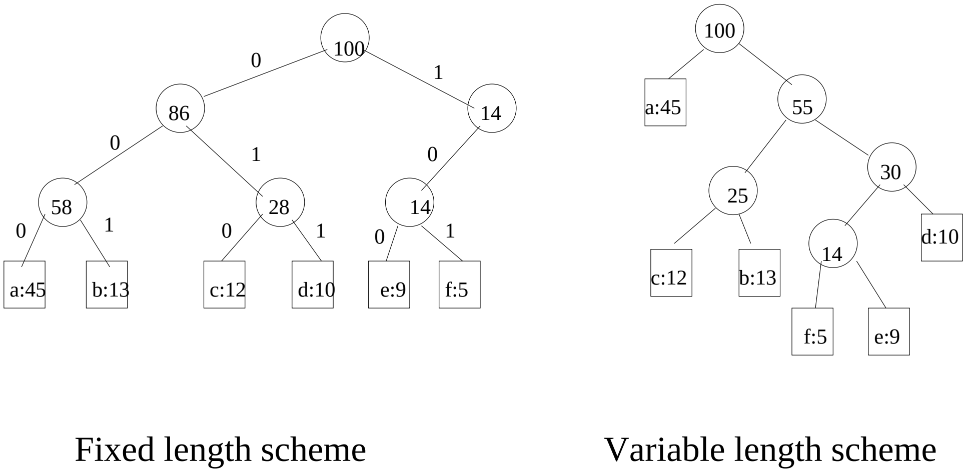
Note that the binary tree is not a search tree.

Let C be a set of n chars, each with frequency $f(c_i)$ in some file. We want to construct a codebook ($\equiv$ tree) of C that minimizes the number of bits needed to represent the file.

Fact: the optimum tree for a code cannot contain a nonleaf node with less than two children. (Otherwise could splice it out, get a shorter code word.)

Given T, the number of bits needed is $\sum_{c \in C} f(c) \cdot d_T(c)$, where $f(c)$ is the frequency of c, and $d_T(c)$ is the depth of c's leaf in T.

Here's an algorithm for creating the codebook.



Fixed length scheme

Variable length scheme

Figure 1: Tree representation of coding schemes

- start with $|c|$ leaves, perform $|c|$ -1 merging operations to create tree

Use a minheap Q, keyed on f to merge the two least frequent objects; result is a new object with frequency that's the sum of the frequencies of the merged objects.

Runtime is $O(n \lg n)$: $n-1$ merges, each takes $O(\lg n)$ time (selection from minheap).

Apply to example yields to the variable length coding in the tree above.

Why does this algorithm work? Can take any optimum tree, replace it by one in which the two deepest siblings are the lowest frequency without increasing cost.

In example below, let x and y have the lowest frequency. WLOG, assume $f(a) \leq f(b)$, and $f(x) \leq f(y)$. Then swapping x with a cannot increase cost, neither can swapping b and y.

Question: in what sense is Huffman greedy?

Cost of a tree is equal to the sum of the costs of its merges—Huffman picks the least cost merge at each step.

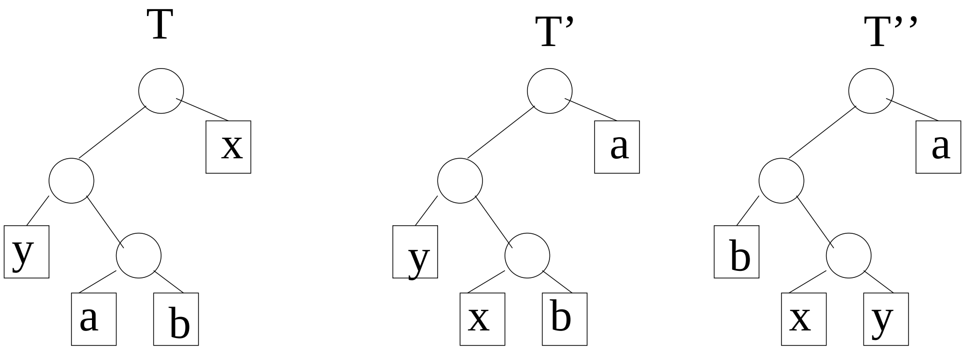


Figure 2: Huffman correctness argument