Greedy Algorithms

A general algorithm construction principle ("paradigm") for combinatorial optimization problems.

The idea is simple:

Build the solution **bit by bit** by always choosing what currently looks like the "**best choice**" (without thinking about the rest of the solution). That is, one hopes that local optimization gives global optimization.

More precisely, the method requires a definition of "**best choice**" (also called "**greedy choice**"), as well as proof that repeated use of this ends with an **optimal** solution.

Example: A Simple Scheduling Problem

Input: A collection of booking requests for a resource, each with a start time and end time. **Output:** A largest possible set of non-overlapping bookings.

Here are 12 non-overlapping booking requests.

Is 12 the maximum number for this input? Yes, but it may not be easy to see. An algorithm is needed to find the maximum number. We try the greedy method.

Greedy Choice Proposal

The one that ends first (among those remaining without overlap with already selected ones). As code:

1. Sort activities by increasing end time
2. For each activity a taken in that order:
    - If a overlaps already selected activities:
        - Skip a
    - Else
        - Select a

We want to show the following invariant: There exists an optimal solution OPT that contains the activities selected by the algorithm so far. When the algorithm is finished, correctness follows from the above invariant: The algorithm's selected activities lie within an optimal solution OPT. Because of the algorithm's operation, all non-selected activities will overlap one of the selected ones. Thus, the algorithm's solution cannot be extended and still be a solution. In particular, OPT cannot be larger than the algorithm's selected solution, which is therefore equal to OPT.

Proof of Invariant by Induction:

- Base:
    - Clear before the first iteration of the for loop (since no activities are selected).
- Step:
    - Let OPT be the optimal solution from the induction statement before the iteration. We need to show that there exists an optimal solution OPT' in the induction statement after the iteration.
    - If-case: Here, the algorithm chooses nothing new, so OPT can be used as OPT'.
    - Else-case:
        - Look at the activities sorted by increasing end times: $a_1, a_2, \ldots, a_n$.
        - Let a_i be the algorithm's most recently selected activity and let a_j be the activity that is selected in this iteration.
        - Because of the invariant, OPT contains a_i. In OPT, a_i cannot be the last (because then OPT could be extended with a_j). Let a_k be the next in OPT after a_i. Since a_j is the first activity (in the above sorting) after a_i that does not overlap a_i, $j \leq k$.
        - If $j = k$, OPT can be used as OPT'.
        - Otherwise, we replace a_j in OPT with a_k. This does not cause overlap with other activities in OPT (they either stop before a_i or stop after a_k - in the latter case, they must start after a_k and therefore after a_j) and preserves the size. We therefore have after the replacement a new optimal solution OPT' that satisfies the invariant.

[NB: In the first iteration, a_i does not exist, but we can set $j = 1$ and say something similar.] **Runtime:** Sorting + $O(n)$.

The Knapsack Problem

Knapsack that can carry W kg. Things with value and weight.

Thing no. i	1	2	3	4	5	6	7
Weight w_i	4	6	2	15	7	4	5
Value v_i	45	32	12	50	23	9	15

Goal: take as much value as possible without exceeding the weight limit.

The "Fractional" version of the problem (parts of things can be included in the knapsack) can be solved with a greedy algorithm: select the things by decreasing "utility" = value/weight. A simple replacement argument shows that the optimal solution can only be the one selected by the algorithm.

NB: This greedy algorithm does NOT work for the 0-1 version of the problem (where only whole things can be included). Example of how "greedy choices" cannot simply be assumed to work for all problems (local optimization does not always give global optimization).

Interpreting Bit Patterns

`01101011 0001100101011011...` Bit patterns must be interpreted to have meaning:

Letters Numbers (integer, decimal) Computer instruction (program) Pixels (image file) Amplitude (audio file) . . .

Representation of Characters

A classic representation: **ASCII**. All characters fill 7 bits (**fixed-width codes**).

. . . a: 1100001 b: 1100010 c: 1100011 d: 1100100 . . .

Huffman Codes

Is fixed-width coding the shortest possible representation of a file of characters? It depends on the file's content! Example:

	A	B	C	D	E	F	G
Fixed	000	001	010	011	100	101	110
Variable	0	100	101	110	1110	11110	11111
Freq.	45.000	13.000	12.000	16.000	9.000	5.000	5.000

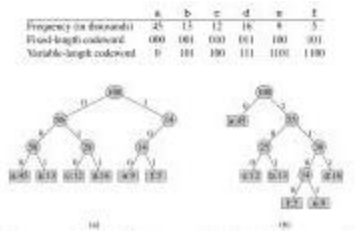
- Fixed-width version: $3 * (45.000 + 13.000 + \cdots + 5.000) = 300.000bits$
- Variable-width version: $1 * 45.000 + 3 * 13.000 + \cdots + 4 * 5.000 = 224.000bits$

Desire: shortest possible representation of a file. Saves space on disk, time on transport over network.

> Prefix-code: Code word = path in binary tree: 0 ~ go to the left, 1 ~ go to the right. Prefix-free code: no code for a character is the start (prefix) of the code for another character (⇒ decoding unambiguous). So characters correspond to nodes with zero children (leaves).

For a given file (characters and their frequencies), find the best variable-width prefix code. I.e. for $Cost(tree) = |codedfile|$, find tree with lowest cost. Optimal trees cannot have nodes with only one child (all characters in the subtree for such a node can be shortened by one bit). So only nodes with two or zero children exist.

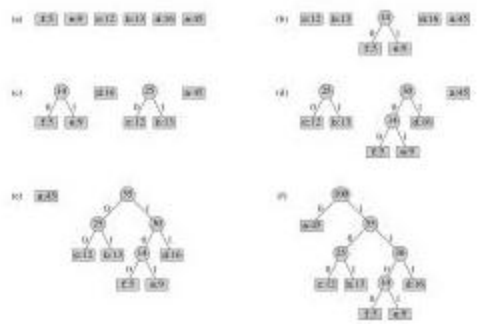
Here is an example of a table, as well as diagrams representing fixed-length and variable-length Huffman Trees:



Huffman's Algorithm

Build up from the bottom (from smallest to largest frequencies) by always making the following "greedy choice": merge the two subtrees with the two smallest total frequencies:

Below is an image of the algorithm being performed to build optimal trees:



Given a table with n characters and their frequencies, Huffman's algorithm makes $n-1$ iterations.

(There are n trees to start, one tree to end, and each iteration reduces the number by exactly one.) By using a (min-)priority queue, e.g. implemented by a heap, each iteration can be performed with:

two `ExtractMin` operations one `Insert` operation $O(1)$ other work.

Each priority queue operation takes $O(logn)$ time. So the total runtime for the n iterations is $O(nlogn)$.

Summary:

Huffman's algorithm maintains a collection of trees F. The weight of a tree is the sum of the frequency in its leaves. In each step, Huffman merges two trees with the smallest weights together, until there is only one tree. We want to prove the following invariant: The trees in F can be merged into an optimal tree. When the algorithm stops, F contains only one tree, which according to the invariant must be an optimal tree.

Correctness

We will prove the following invariant: The trees in F can be merged into an optimal tree. The proof is via induction over the number of steps in the algorithm.

- Basis: No steps are taken. Then F consists of n trees, each of which is just a leaf. These are precisely the leaves in any optimal tree, so the invariant is trivially satisfied.
- Induction step: assume the invariant is satisfied before a step in the algorithm, and let us show that it is satisfied after the step.

Let the trees in F (before the step) be $t_1, t_2, t_3, \ldots, t_k$ where the algorithm merges t_1 and t_2 together. That is, with respect to weight, the following holds for the trees: $t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_k$, According to the induction assumption, the trees can be merged into an optimal tree. Let T be the top of this tree—i.e., a tree whose leaves are the roots of $t_1, t_2, t_3, \ldots, t_k$.

- Case 1: The roots of t_1 and t_2 are siblings in T.
    - Here, the new collection of trees (after the step, where t_1 and t_2 are merged) can still be built together into the same optimal tree mentioned in the invariant before the step.
    - So the invariant holds again after the step.
- Case 2: The roots of t_1 and t_2 are not siblings in T.
    - Here, we will find another top-tree T' (i.e., another merging of the trees in F), which also gives an optimal tree, and where t_1 and t_2 are siblings. For this optimal tree, we are in Case 1 and thus finished.
    - We find T' from T as follows:
        - Look at a leaf in T of greatest depth. Since $k \geq 2$ (otherwise Huffman's algorithm was finished), the leaf has a parent. This parent has at least one subtree, so it has two (in optimal trees, no nodes have one subtree, as noted earlier). Its other subtree must be a leaf, otherwise the first leaf is not a deepest leaf.
        - So there are two leaves in T that are siblings and both are of greatest depth. Let these contain the roots of t_i and t_j, with $i < j$.

Possible situations:

1. 2. 3. . . . i j i j i j i, j

Action that gives T' from T: None (since we are in Case 1) Swap t_2 and t_j Swap t_1 and t_j Swap t_1 and t_i, as well as t_2 and t_j

For a swap of t_1 and t_i, since t_i's root has at least the same depth as t_1's root, and the total frequency of t_i is at least as large as the total frequency of t_1, there will be more characters in the file that get shorter codewords than there are characters that get longer codewords. Furthermore, the changes in codeword length are the same for both lengthening and shortening (namely, the difference in depth between t_1 and t_i). So the coded file's length does not increase by swapping t_1 and t_i, i.e., the tree cannot get worse by swapping t_1 and t_i. Similarly can be shown for a swap of t_1 and t_j, and for a swap of t_2 and t_j. Since the tree was optimal before the swap, it is also after. And t_1 and t_2 are now siblings.