

23. Greedy Algorithms

Fractional Knapsack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local criterion that is not depending on the solution of the sub-problems.

Examples: fractional knapsack, Huffman-Coding (below)

Counter-Example: knapsack problem, Optimal Binary Search Tree

The Fractional Knapsack Problem

set of $n \in \mathbb{N}$ items $\{1, \dots, n\}$ Each item i has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$. The maximum weight is given as $W \in \mathbb{N}$. Input is denoted as $E = (v_i, w_i)_{i=1, \dots, n}$.

Wanted: Fractions $0 \leq q_i \leq 1$ ($1 \leq i \leq n$) that maximise the sum $\sum_{i=1}^n q_i \cdot v_i$ under $\sum_{i=1}^n q_i \cdot w_i \leq W$.

Greedy heuristics

Sort the items decreasingly by value per weight v_i/w_i .

Assumption $v_i/w_i \geq v_{i+1}/w_{i+1}$

Let $j = \max\{0 \leq k \leq n : \sum_{i=1}^k w_i \leq W\}$. Set

■ $q_i = 1$ for all $1 \leq i \leq j$.

■ $q_{j+1} = \frac{W - \sum_{i=1}^j w_i}{w_{j+1}}$.

■ $q_i = 0$ for all $i > j + 1$.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the computation of the q_i .

Correctness

Assumption: optimal solution (r_i) ($1 \leq i \leq n$).

The knapsack is full: $\sum_i r_i \cdot w_i = \sum_i q_i \cdot w_i = W$.

Consider k : smallest i with $r_i \neq q_i$ Definition of greedy: $q_k > r_k$. Let $x = q_k - r_k > 0$.

Construct a new solution (r'_i) : $r'_i = r_i \forall i < k$. $r'_k = q_k$. Remove weight $\sum_{i=k+1}^n \delta_i = x \cdot w_k$ from items $k+1$ to n . This works because $\sum_{i=k}^n r_i \cdot w_i = \sum_{i=k}^n q_i \cdot w_i$.

Correctness

$$\begin{aligned}\sum_{i=k}^n r'_i v_i &= r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n (r_i w_i - \delta_i) \frac{v_i}{w_i} \\ &\geq r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} - \delta_i \frac{v_k}{w_k} \\ &= r_k v_k + x w_k \frac{v_k}{w_k} - x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} = \sum_{i=k}^n r_i v_i.\end{aligned}$$

Thus (r'_i) is also optimal. Iterative application of this idea generates the solution (q_i) .

Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

Example

File consisting of 100.000 characters from the alphabet $\{a, \dots, f\}$.

	a	b	c	d	e	f
Frequency (Thousands)	45	13	12	16	9	5
Code word with fix length	000	001	010	011	100	101
Code word variable length	0	101	100	111	1101	1100

File size (code with fix length): 300.000 bits.

File size (code with variable length): 224.000 bits.

Huffman-Codes

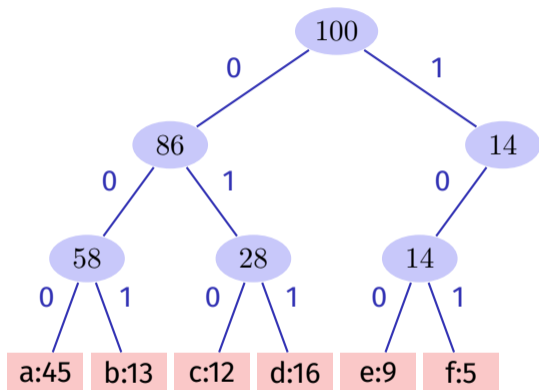
- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal data compression (without proof here).
- Encoding: concatenation of the code words without stop character (difference to morsing).

$af fe \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$

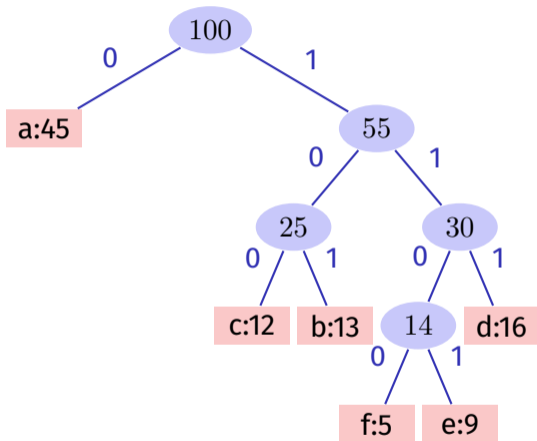
- Decoding simple because prefixcode

$0110011001101 \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow af fe$

Code trees



Code words with fixed length



Code words with variable length

Properties of the Code Trees

- An optimal coding of a file is always represented by a complete binary tree: every inner node has two children.
- Let C be the set of all code words, $f(c)$ the frequency of a codeword c and $d_T(c)$ the depth of a code word in tree T . Define the **cost** of a tree as

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

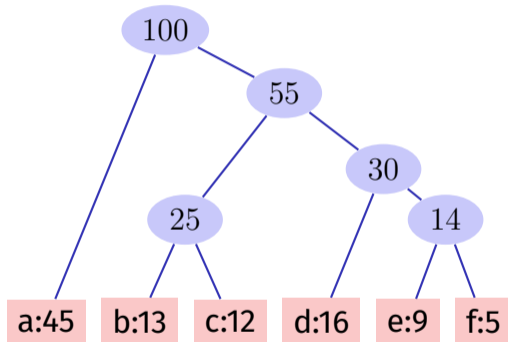
(cost = number bits of the encoded file)

In the following a code tree is called optimal when it minimizes the costs.

Algorithm Idea

Tree construction bottom up

- Start with the set C of code words
- Replace iteratively the two nodes with smallest frequency by a new parent node.



Algorithm Huffman(C)

Input: code words $c \in C$

Output: Root of an optimal code tree

$n \leftarrow |C|$

$Q \leftarrow C$

for $i = 1$ **to** $n - 1$ **do**

 allocate a new node z

$z.\text{left} \leftarrow \text{ExtractMin}(Q)$

// extract word with minimal frequency.

$z.\text{right} \leftarrow \text{ExtractMin}(Q)$

$z.\text{freq} \leftarrow z.\text{left}.\text{freq} + z.\text{right}.\text{freq}$

$\text{Insert}(Q, z)$

return $\text{ExtractMin}(Q)$

Analyse

Use a heap: build Heap in $\mathcal{O}(n)$. Extract-Min in $\mathcal{O}(\log n)$ for n Elements.
Yields a runtime of $\mathcal{O}(n \log n)$.

The greedy approach is correct

Theorem 19

Let x, y be two symbols with smallest frequencies in C and let $T'(C')$ be an optimal code tree to the alphabet $C' = C - \{x, y\} + \{z\}$ with a new symbol z with $f(z) = f(x) + f(y)$. Then the tree $T(C)$ that is constructed from $T'(C')$ by replacing the node z by an inner node with children x and y is an optimal code tree for the alphabet C .

Proof

It holds that

$$f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x) + f(y)) \cdot (d_{T'}(z) + 1) = f(z) \cdot d_{T'}(x) + f(x) + f(y).$$

Thus $B(T') = B(T) - f(x) - f(y)$.

Assumption: T is not optimal. Then there is an optimal tree T'' with $B(T'') < B(T)$. We assume that x and y are brothers in T'' . Let T''' be the tree where the inner node with children x and y is replaced by z . Then it holds that $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$. Contradiction to the optimality of T' .

The assumption that x and y are brothers in T'' can be justified because a swap of elements with smallest frequency to the lowest level of the tree can at most decrease the value of B .