CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 9: Huffman Coding

We study a well-known greedy algorithm that gives an optimal prefix code.

Compression

Suppose a text has 26 letters $a, b, c, \ldots, z$.

A standard way to represent the letters using bits is to use $\lceil \log_2 26 \rceil = 5$ bits for each letter,

e.g. $a = 00000$, $b = 00001$, $c = 00010$, ..., $z = 11010$.

So, we use five bits to represent each letter.

In general, we cannot do much better if each letter appears equally likely.

What if we scan the text once and notice that the letters appear with quite different frequencies? E.g. "a" appears 10% of the time, "b" 2%, "c" 3%, "d" 2%, "e" 12%, etc.

Can we hope to do better by using variable-length encoding scheme?

The idea is to use fewer bits for more frequent letters, and more bits for less frequent letters, so that the average number of bits used is fewer.

Prefix Coding

When we use fixed-length bit strings to represent the letters, it is easy to decode.

Say, if we use five bits to encode the 26 letters, we just need to read five bits at a time, to decode one letter at a time.

It is not as clear how to decode if we use variable-length encoding.

Say, suppose we encode the five letters as $a = 01$, $b = 001$, $c = 011$, $d = 110$, $e = 10$. Then, when we read a compressed text such as 00101110, it could be decoded as "bce", but it could also be decoded as "bad".

To avoid ambiguity in decoding, we will construct prefix codes, so that no encoded string is a prefix of another encoded string.

In the above example, the ambiguity arises because the string representing "a" is a prefix of the string representing "c", i.e. $a = 01$ and $c = 011$.

Now, suppose we use a prefix code for the five letters, $a = 11$ $b = 000$ $c = 001$ $d = 01$ $e = 10$.

Then, we encode the text "cabed" using the string 00111000101.

When the decoder reads the string 00111000101, it will read from left to right and

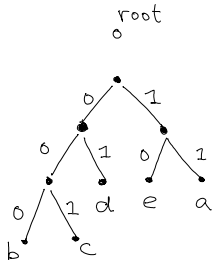
unambiguously decode the text "cabed".

In short, prefix coding allows for easy and efficient encoding and decoding.

Decoding Tree

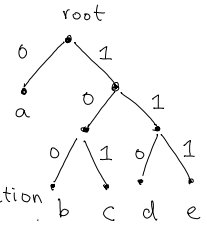
It is useful to represent a prefix code as a binary tree.

In the example above, the prefix code can be represented by the tree



To decode, we start from the root, read a bit, move to the corresponding branch, until we reach a leaf, then we return the letter associated to the leaf, and go back to the root and repeat.

As another example, a=0, b=100, c=101, d=110, e=111, then the tree is



This is useful when the frequency of "a" is very high.

It should be clear that each prefix code has a binary tree representation and each binary tree representation corresponds to a (unique) prefix code.

Objective

Suppose we are given the frequencies of the five letters, say $f_a = 0.8$ $f_b = f_c = f_d = f_e = 0.05$.

Then, the average length of a letter (in the second example) is $\underbrace{0.8 \times 1}_{\text{1 bit for "a"}} + \underbrace{0.05 \times 3}_{\text{3 bits for "b"}} + 0.05 \times 3 + 0.05 \times 3 = 1.4$.

In the tree representation, the length of an encoded string is equal to the depth of the corresponding leaf, and the objective becomes $\sum_i f_i \cdot \text{depth}_T(i)$.

Optimal Prefix Code

Input: n symbols with frequencies $f_1, \ldots, f_n$ so that $\sum_{i=1}^{n} f_i = 1$.

Output: a binary tree T with n leaves that minimizes $\sum_{i=1}^{n} f_i \cdot \text{depth}_T(i)$.

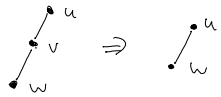
This problem doesn't look so easy as the output space is quite complicated, as there are exponentially many possible binary trees.

It is also not clear how the algorithm can make a decision greedily.

Let's first think about how an optimal solution should look like.

We say a binary tree is <u>full</u> if every internal node has two children.

We start with a simple observation.



Observation The binary tree of an optimal solution is full.

Proof If there is an internal node with only one child, then we can directly connect its child to its parent and decrease the depth of some leaves, getting a better solution. □

Corollary There are at least two leaves of maximum depth that are siblings (having the same parent).

Proof Look at a leaf of maximum depth. If it has no sibling, then the tree is not full. 

□

Suppose we know the shape of an optimal binary tree (which we don't know yet).
Then it is not difficult to figure out how to assign symbols to the leaves.
We should assign symbols with highest frequencies to the leaves with smallest depth, and assign symbols with lowest frequencies to the leaves with largest depth.
Otherwise, if one symbol of higher frequency is of a larger depth than another symbol of lower frequency, then we could "exchange" the two symbols and decrease the objective value.
This exchange argument leads to the following observation.

Observation. There is an optimal solution in which the two symbols with lowest frequencies are assigned to leaves of maximum depth, and furthermore they are siblings.

Proof By the corollary, there are two leaves of maximum depth that are siblings.
By the exchange argument, we can assign them with the symbols with lowest frequencies without increasing the objective value (just exchange with the symbols there).
So, the new solution is still optimal and satisfies the properties as stated. □

Huffman's Algorithm

So far, we have deduced very little information about the optimal solutions.
We just know that there are two leaves of maximum depth that are siblings, and we can assign two symbols with lowest frequencies there.
But we still don't know how the tree should look like (e.g. maximum depth, minimum depth, etc), and we also don't know how to use the frequencies to make decisions.

Huffman figured out that this little information is already enough to design an efficient algorithm.
The idea is to "reduce" the problem size by one, by identifying / combining two symbols with lowest frequencies into one, knowing that they can be assumed to be siblings of maximum depth.

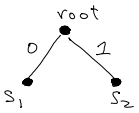
How the tree should look like will become apparent when the problem size becomes small enough, and then we can construct back a bigger tree from a smaller tree one step at a time.

Algorithm: (Huffman Code)

Input: a set S of n symbols, with frequencies $f_1, \dots, f_n$.

Output: an optimal binary tree T, with leaves associated to symbols in S.

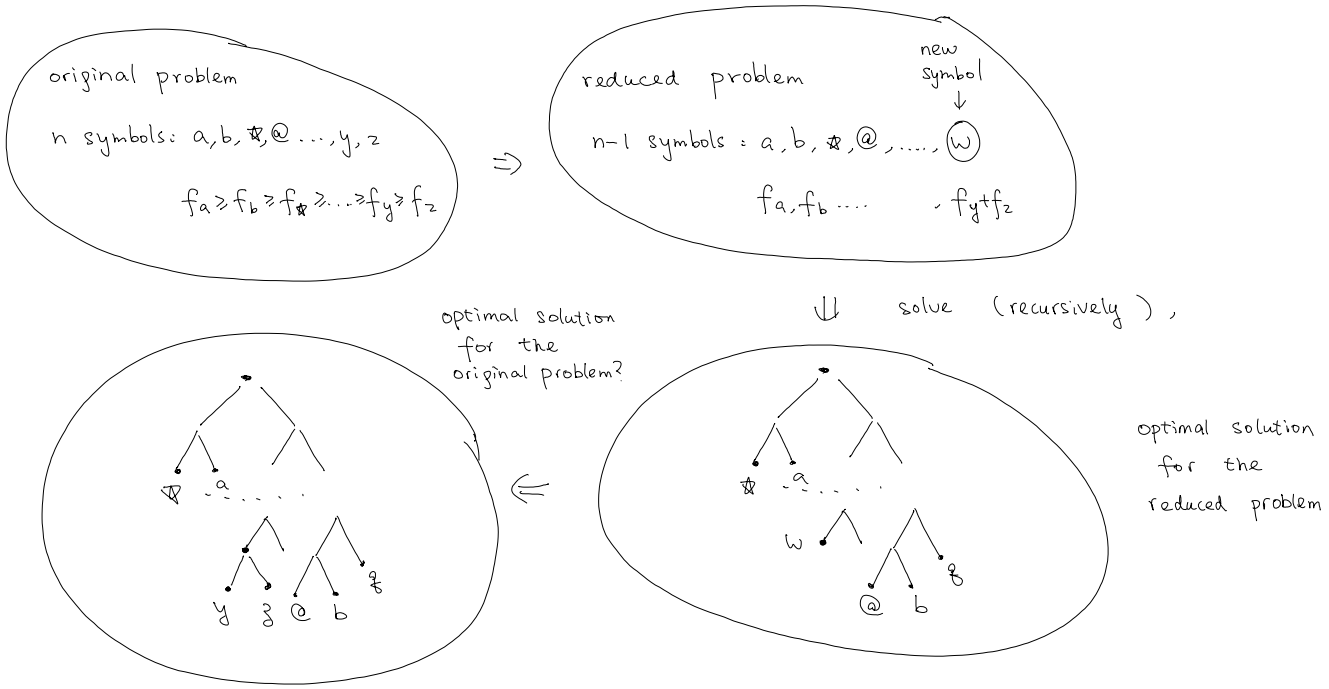
Base case: If $|S| = 2$, encode one symbol using 0 and another symbol using 1, and return the tree.



Induction step: Let y and z be two symbols with lowest frequencies, denoted by f_y and f_z.

1. Delete symbols y and z from S. Add a new symbol w with frequency $f_y + f_z$.
2. Solve this new problem (with $n-1$ symbols) recursively and get an optimal tree T'.
3. In T', look at the leaf associated with w, add two leaves to it (so that w becomes an internal node), and associate y and z with the two new leaves.

The scheme is summarized as follows.



Examples: Let's do some examples to be familiar with Huffman's algorithm.

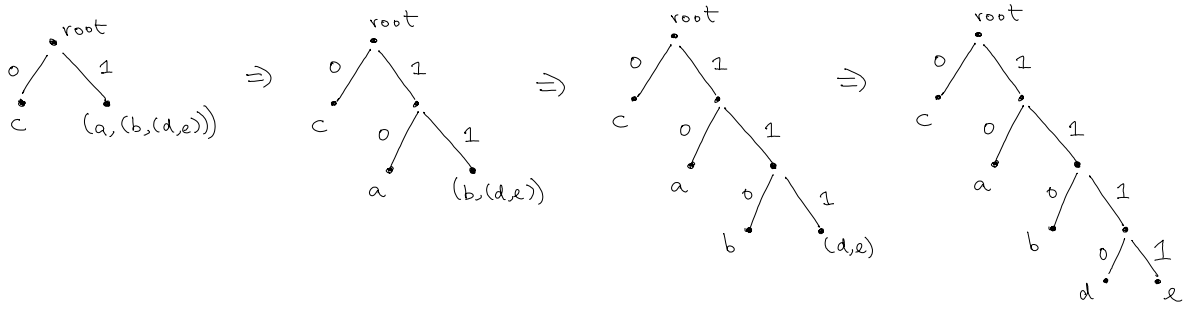
① five symbols: a, b, c, d, e, with $f_a = 0.3$ $f_b = 0.2$ $f_c = 0.4$ $f_d = 0.05$ $f_e = 0.05$

⇒ reduce to four symbols: a, b, c, (d,e) with $f_a = 0.3$ $f_b = 0.2$ $f_c = 0.4$ $f_{(d,e)} = 0.1$

⇒ reduce to three symbols: a, (b,(d,e)), c with $f_a = 0.3$ $f_{(b,(d,e))} = 0.3$ $f_c = 0.4$

$\Rightarrow$ reduce to two symbols : $(a,(b,(d,e)))$, c with $f_{(a,(b,(d,e)))} = 0.6$ $f_c = 0.4$.

construct trees



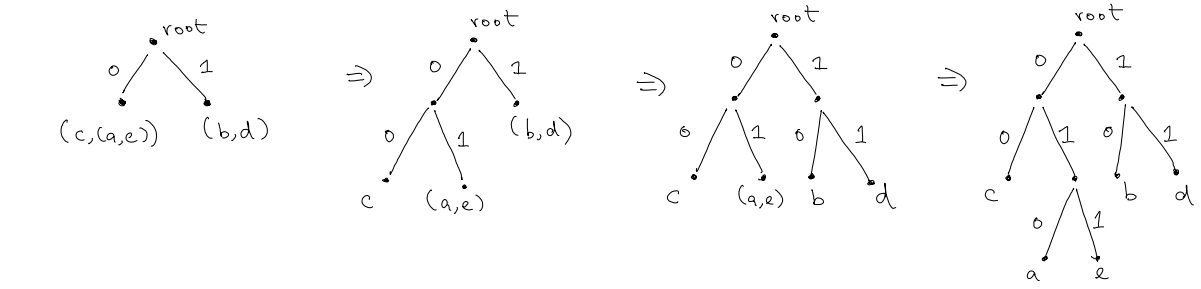
② five symbols : a,b,c,d,e , with $f_a = 0.18$ $f_b = 0.24$ $f_c = 0.26$ $f_d = 0.2$ $f_e = 0.12$

$\Rightarrow$ reduce to four symbols : $b,c,d,(a,e)$ with $f_b = 0.24$ $f_c = 0.26$ $f_d = 0.2$ $f_{(a,e)} = 0.3$

$\Rightarrow$ reduce to three symbols : $c,(b,d),(a,e)$ with $f_c = 0.26$ $f_{(b,d)} = 0.44$ $f_{(a,e)} = 0.3$

$\Rightarrow$ reduce to two symbols : $(c,(a,e)),(b,d)$ with $f_{(c,(a,e))} = 0.56$ $f_{(b,d)} = 0.44$

Construct trees



Correctness Proof

The most natural way to analyze a recursive algorithm is to use induction.

We will use the same terminology as described in the algorithm.

Clearly, the output of the base case when there are only two symbols is optimal.

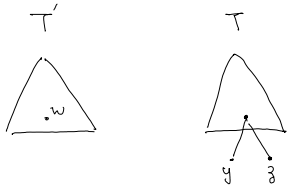
Denote the objective value of the solution returned by Hoffman's algorithm by Sol(n).

By induction, we have computed correctly an optimal binary tree T' for the n-1 symbols,

with y and z replaced by w, and $f_w = f_y + f_z$.

Let $Obj(T')$ be the objective value of T'.

What is the relationship between Sol(n) and $Obj(T')$.



We just add two leaves to T' to form T.

Recall that the objective is $\sum_{i=1}^{n} f_i \cdot depth_T(i)$.

Every other leaf has the same contribution to the objective value of T and T'.

So, just focus on the change of deleting w and then adding y and z back.

Then, $sol(n) = Obj(T') - f_w \cdot depth_{T'}(w) + f_y \cdot (depth_{T'}(w) + 1) + f_z \cdot (depth_{T'}(w) + 1)$

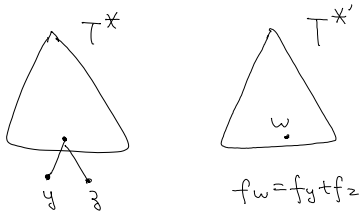
$= Obj(T') + f_y + f_z$, since $f_w = f_y + f_z$ by our construction.

Next, we would like to argue that any optimal solution must have objective value at least $Obj(T') + f_y + f_z$, and this would imply that the solution returned by Hoffman's algorithm is optimal.

Let T^* be an optimal solution for the original problem.

By the lemma using the exchange argument, we can assume that y and z are leaves of maximum depth in T^* and furthermore they are siblings.

We define $T^{*'}$ as obtained from T^* by deleting y and z and define $f_w = f_y + f_z$ where w is the parent of y and z.



By the same calculation above, $Obj(T^*) = Obj(T^{*'}) + f_y + f_z$.

Now, observe that $T^{*'}$ is a solution to the reduced problem of size $n-1$.

By the induction hypothesis, T' is an optimal solution to the reduced problem, and hence it must hold that $Obj(T') \leq Obj(T^{*'})$.

Putting together, we have $Obj(T^*) = Obj(T^{*'}) + f_y + f_z \geq Obj(T') + f_y + f_z = Obj(T) = Sol(n)$, proving that T is an optimal solution.

Implementation

In every iteration, we need to find two symbols with lowest frequencies, delete them and add a new symbol with the frequency as their sum.

A straightforward implementation takes $\Theta(n)$ time to find two symbols with lowest frequencies.

We can use a heap to do these operations in $O(\log n)$ time.

Recall that a heap supports the operations of insert and extract-min in $O(\log n)$ time.

So, each iteration can be implemented in $O(n \log n)$ time, after initially inserting all n frequencies in the heap in $O(n \log n)$ time.

Therefore, the total time complexity is $O(n \log n)$.

Reference : [KT 4.8]