Lecture Notes on Huffman Coding and Horn Formulas

1 Huffman Coding

Suppose that you must store a string of 130 million characters of the form A, C, G, and T (say, the map of a chromosome). You would not like to store them as bytes, but pack them as bits. You could represent A as 00, C as 01, G as 10, and T as 11. This would cost you 260 megabits.

Suppose now that you knew that A appears 70 million times in the string, C 3 million times, G 20 million times, and T 37 million times. In this case you may want to invent a more elaborate encoding, perhaps assigning a shorter bit string to A than to C. Any such encoding must obey the *prefix property*, in that *no code (bitstring) is a prefix of another code.* This property enables you to decode the bitstring without backtracking.

You can envision such an encoding as a *binary tree.* For example, the binary tree below correponds to the optimal encoding in the above situation, with A mapped to 0, C mapped to 110, G mapped to 111, and T mapped to 10. To recover the encoding from the tree, just interpret "left" as 0 and "right" as 1. This encoding stores the string in 213 million bits —a 17% improvement over the balanced tree (the encoding 00, 01, 10, 11).

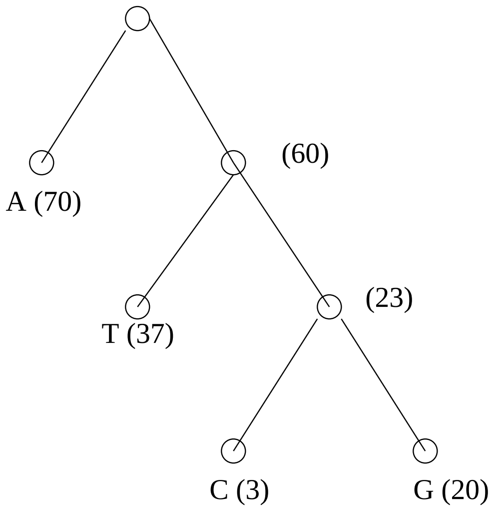

Figure 1: The Huffman tree

Such a tree must have the letters in the leaves (if there are internal nodes that are letters, then we lose the prefix property). The codes are the root-to-leaf paths (by interpreting "left" as 0 and "right" as 1, say). All internal nodes have exactly two children (an internal node with one child could be deleted for a better code); consequently, if we have n leaves, there are $n-1$ internal nodes (prove!). Finally, if we assign frequencies to the internal nodes (the sums of the frequencies of their children), then it is easy to see that the total cost is *the sum of the frequencies of all internal nodes and leaves except the root.* (Proof: Each edge is a bit that is written as many times as the frequency of the node to which it leads.)

There is one final property of this tree that is as important as it is intuitive: *The two letters with the smallest frequencies are together at the bottom, they are children of the lowest internal node in the tree.* Otherwise, if the two bottom leaves were other letters, we could improve the encoding by swapping.

This tells us how to start constructing the optimum tree: *Be greedy.* Take the two lowest frequencies, make them children of an internal node, and delete them from the list of letters. In some sense, this internal node can *replace* these two letters in the problem —in the Figure, the node labeled 23 can be thought of as a 'CG' node, after C and G are deleted from the list of letters. So, after we delete the two leaves, we insert their parent to the list, *and continue the same way.*

It can be shown by induction that this gives the optimum tree: It works for two letters, and if it works for n, then it works for $n+1$: Delete the two least frequent letters, and insert their parent (with frequency the sum of the two frequencies), and repeat. By induction this gives the best tree for the n leaves. And since the optimum costs of the two problems differ by the sum of the two deleted frequencies (recall how we compute the cost), it follows that the overall tree is optimum as well.

Here is the full algorithm:

Algorithm Huffman(f: array of int);
H: heap of int prioritized by f;
one, zero: array of int, initially nil (these are the parent-to-child pointers of the sought tree)
$i, j, next$: int;
for $i := 1$ to n do $\{$insert$(H, i)\}$,
next:= $n+1$ (this is the index of the next internal node to be constructed)
while $|H| \geq 2$ do
  $\{i :=$ deletemin$(H), j :=$ deletemin(H),
  zero(next):= i, one(next):= j, f(next) := $f(i) + f(j)$,
  insert(H,next), next++$\}$

Now, to decode using the Huffman tree you just run this algorithm:

Algorithm decoder
nextbit: bit, i: int,
$i := 2n - 1$ (initialize the current tree node to the root)
while not end-of-file do
  {read(nextbit)
  if nextbit=0 then i :=zero(i) else i :=one(i)
  if zero(i)=nil then (we have arrived at a leaf)
    { write(i) (or the ith letter), $i := 2n - 1$ (back to root)}}

Coming back to the chromosome example, I am not saying that there is no better way to compress this string. The claim is the we cannot do better *by encoding single letters.* Chances are that there are far better encodings, say by encoding the frequently occurring substring AAGATTA as 010, etc. Once the blocks to be encoded are decided by scanning the text, Huffman's algorithm gives the best encoding. Many applications (including image and video transmission) have Huffman codes as their basic building block.

2 Horn Formulas

There is an interesting last application of the greedy algorithm to solve an important special case of the SAT problem. We know that 2SAT can be solved in linear time, and that 3SAT (and SAT in general) is NP-complete, and so presumably *very* difficult. But consider the following instance:

$$(x \vee \overline{y} \vee \overline{z} \vee \overline{w})\&(\overline{x} \vee \overline{y} \vee \overline{w})\&(\overline{x} \vee \overline{z} \vee w)\&(\overline{x} \vee y)\&(x)\&(\overline{x} \vee \overline{y} \vee w)\&(\overline{z}))$$

It is certainly not 2SAT, but it does have a favorable property: *in each clause there is at most one positive literal.* Such formulas are called *Horn formulas*, and their satisfiability problem can be solved very fast.

Given a Horn formula, the algorithm separates its clauses into two parts: The *implications* (those clauses that do have a positive literal) and the *pure negative clauses* (those that don't). An implication such as the first clause can be rewritten more suggestively as $y\&z\&w \rightarrow x$, with the variable appearing positive being implied by the rest. This is a completely equivalent statement (check it). Thus we have these implications:

$$(y\&z\&w \rightarrow x), (x\&z \rightarrow w), (x \rightarrow y), (\rightarrow x), (x\&y \rightarrow w)$$

and these two pure negative clauses

$$(\overline{x} \vee \overline{y} \vee \overline{w})\&(\overline{z})$$

Notice that we consider the clause (x) as a trivial kind of implication.

Here is the algorithm:

Algorithm stingy(ϕ: CNF formula with at most one positive literal per clause)
Start with the truth assignment t :=FFF$\cdots$F
while there is an implication that is not satisfied do
{make the implied variable T in t}
if all pure negatives are satisfied then return t
else return "ϕ is unsatisfiable"

The idea in the algorithm is to favor F (false) over T (true) —as if T were very expensive. We start with the truth assignment assigning to x, y, z, w the values FFFF, and we only turn a F into a T *only if an implication forces us to* —we ignore the pure negatives at this point. Notice that we never flip a T back to F. Recall that an implication is not satisfied only if all variables to the left of the arrow are T and the one to the right is F.

In our example, the fourth implication forces us to flip x from F to T —the truth assignment becomes TFFF. Then the second implication forces y to be T —we get TTFF. Then the last implication forces w to be T —we have TTFT. Now all implications are satisfied, and the first phase is over.

Thus, in the end of this phase, we have a truth assignment that satisfies all implications, *and has as few T as possible* —all T's were forced. We will eventually come up with such a truth assignment (at worst, we can make all variables T).

We finally look at the pure negatives. If there is a pure negative clause that is not satisfied by the truth assignment we have found, then the formula is unsatisfiable. The reason is that the truth assignment we found has only forced T's; so, if a negative clause needs one of them to be F, it cannot be satisfied. Otherwise, if all negative clauses are satisfied, then obviously we have found a satisfying truth assignment and can report it.

The stingy algorithm can be implemented in *linear time* in the length of the formula (the number of all occurrences of literals in it).

Horn formulas is the second of the three important special cases of SAT that can be solved in polynomial time (the other two are 2SAT, which we have seen, and equations modulo 2, which we will see). Incidentally, the stingy algorithm is the workhorse in PROLOG interpreters (PROLOG, for "programming by logic," is a wonderful language, popular in Europe, that is extremely declarative, and allows you to program by simply specifying the desired properties of the output in a simple logical language).