COMPSCI 330: Design and Analysis of Algorithms Spring 2019

Lecture 9: Greedy Algorithms III

Lecturer: Rong Ge *Scribe: Shweta Patwa*

1 Example: Horn-SAT

Whether a logical formula is satisfiable or not? Variables take boolean values, and the negation of a variable x is denoted by $\bar{x}$. A Horn-SAT formula is of one of the following three types:

- Type 1: $x \wedge y \implies z$
- Type 2: x
- Type 3: $\bar{x} \vee \bar{y} \vee \bar{z}$

Problem: Given a set of Horn clauses, does there exist an assignment to variables s.t. all the clauses are satisfiable?

First set everything to false.

Set variables in constraints of type 2 to be true.

Look at constraints of type 1 and set variables that must be true.

Must check if all trpe 3 clauses are still satisfied.

If any check is violated, return NO.

Proof. If the algorithm succeeds, then there is clearly an assignment.

If the algorithm fails, then no assignment can satisfy the clauses.
(Proof Sketch.) We prove this by contradiction. Assume there is a satisfying assignment. We consider the following cases.
Suppose the satisfying assignment also sets the same set of variables to true, as in the solution computed by the algorithm. But such an assignment cannot be a satisfying assignment because clauses of type 3 remain unsatisfied.
Thus, one of the variables that were set to true must instead be false. Look at the 'first' variables that were false. The ordering is in terms of how the algorithm proceeds and sets variables to be true. However, we needed the 'first' variables to be true to satisfy clauses of type 2. Therefore, these must remain true.
Next, consider variables that were set to true to satisfy clauses of type 1. These must remain true.
Hence, there cannot be a satisfying assignment. □

2 Example: Encoding Problem

Given a very long string with n characters in alphabet, find a way to encode these characters into binary codes that minimizes the length of the encoding.
Example. "*aababc*", n= 3 and alphabet is $\{a,b,c\}$. If a is '0', b is '11' and c is '10', then the string can be

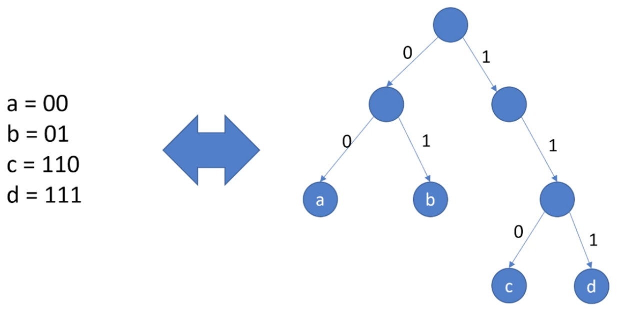
encoded as ‘001101110’.

If we encode all characters to 0, then there is no way to do the decoding successfully because we cannot differentiate between the encoded strings.

Why not use ASCII?
Different characters appear with very different frequency. We do not need the same number of bits to encode each character in the alphabet. We can use varied length encoding to save space. Can help save on the communication you need to transmit strings.

Here is another bad example of encoding: a is ‘0’, b is ‘01’ and c is ‘1’. This is a bad encoding because the code for a is a prefix of the code for b. Cannot differentiate between ac and b. This must be avoided.
<u>Sufficient condition:</u> Prefix-free encoding, i.e., no code should be the prefix of another code.

Can think of prefix-free encoding as a binary tree. Each node will have at most 2 children. One child gets labeled with 0, and the other with 1.



The code for a is 00, b is 01, c is 110 and d is 111. (Aside: right child of root is unnecessary.)

2.1 Prefix tree vs. binary tree

Claim: If all characters are leaf nodes in a tree, then the encoding is a prefix tree, and vice versa.
(Proof sketch.) If there was a character whose encoding is a prefix of some other character’s encoding, then the corresponding vertex must be an intermediate vertex in the tree. But characters are represented by leaves. Thus, for every pair of characters in the alphabet, the encoding for one cannot be a prefix of the encoding for the other.

Claim. All prefix-free encoding allow unique decoding.
Decoding algorithm: Start from the root of the tree

```
for $i = 1$ to $l$
    follow the edge labeled by $a[i]$
    if we reach a leaf, output the corresponding character and return to the root
```

The cost of the tree can be defined as sum of the products of frequency and depth for each character. This gives the cost of the above tree to be $5*2+10*2+3*3+6*3=57$ if the frequencies are $\{5,10,3,6\}$.

2.2 Hoffman Tree Problem

Given a long string with n characters in the alphabet, find a way to encode these characters into binary codes that minimizes the length.
Alternatively, given an alphabet of n characters, character i appears w_i times, and an encoding tree with minimum cost.

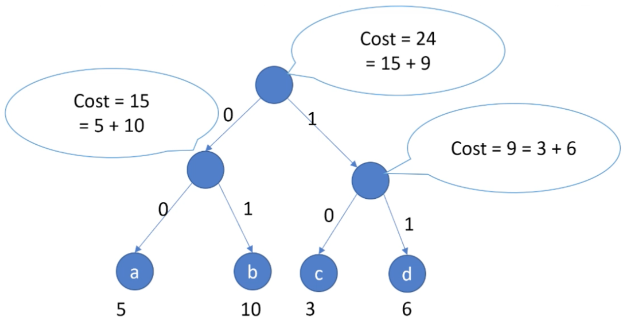


Figure 1: Example: Cost of tree

Claim. Cost of the tree = sum of cost for each node. (Cost of leaf nodes is zero).
(Proof sketch.) Can prove by induction on the size of the tree. Setup the IH as: If the two costs are the same for trees of size k, then prove that it also holds for trees of size $k+1$.

Algorithm to compute Hoffman tree:

```
Repeat
    Select two characters with smallest frequency
    Merge them into a new character, whose frequency is the sum
Until there is only one character left
```

How do we construct a tree by making a sequence of decisions? Initially, we will have n nodes that are leaves of the tree. At every step, we pick two nodes and merge them. Cost of merging at ith step is the total frequency of the two merged nodes. This comes from the second way to compute the cost of the tree that we saw above. The goal for a greedy algorithm then becomes to minimize the cost at each step.

Running time:

- Naive: n iterations because each iteration reduces the number of characters by one. Each iteration takes $O(n)$ time if each step we are doing a linear scan to figure out which two nodes/characters to combine. Overall, the runtime is $O(n^2)$.
- Priority queue/heap: Support finding the minimum element, add element and delete element, all of which can be done in $O(\log n)$.

(Proof sketch.) Use induction: Assume algorithm finds the optimal solution for alphabet of size n, we want to prove that it works for alphabets of size $n+1$.

- Base case: $n = 1$ is trivial. Cost is zero in this case. Can consider $n = 2$.
- Induction step: Suppose algorithm finds a solution that is not an optimal solution. First merge i and j, then apply IH. The only place the algorithm can make a mistake is by messing up on the first merge. Need to argue that it is ok to merge i and j first, and that there is a choice for them that works.

  Consider different cases and prove that the optimal solution we look at must have made an error. If i is at smaller depth than that of j, then we can swap j's sibling with i which would improve the optimal solution. But this should not have been possible.

  See notes for complete proof.