Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Feb. 13, 2023 (Lecture 9)

Greedy Algorithm (Part 2)

Review

- The main idea of greedy algorithm is look some optimal solution locally and then try to extend globally. Usually the greedy algorithm is efficient.
- The greedy algorithm may not achieve optimal solution for the problem.
- We shall arrive at the greedy algorithm by first considering a dynamic programming approach and then showing that we can always make greedy choices to arrive at an optimal solution.

Example 2
Huffman Codes

Huffman Codes

We consider how to encode the data of sequence characters into binary codes efficiently. Suppose we have a 100,000 character data file which contains 6 different characters. We know the frequency of these characters.

We may use fixed length codeword to encode, or use variable length codeword to encode.

The following table shows the details of the example (frequency is in thousands).

Huffman Codes Example

	a	b	c	d	e	f
frequency	45	13	12	16	9	5
fixed-length codeword	000	001	010	011	100	101
variable-length codeword	0	101	100	111	1101	1100

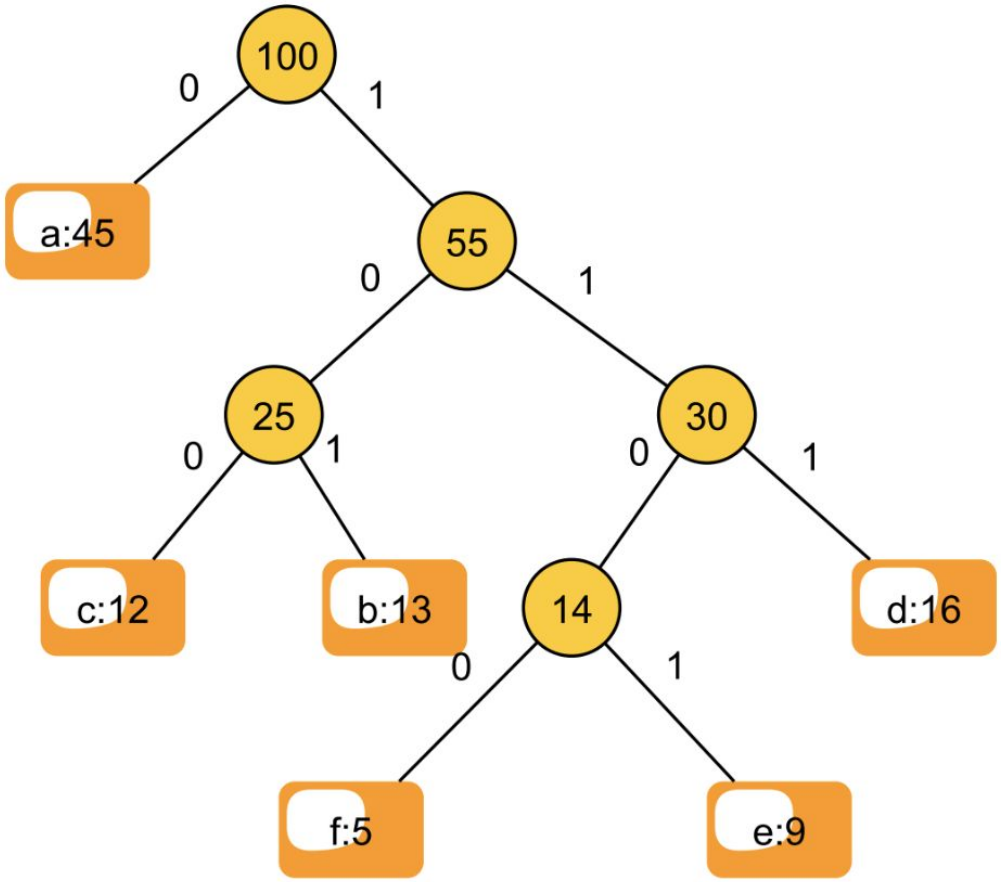
When we use the fixed-length codewords, the encoded file requires 300,000 bits. But if we use the variable-length codewords, the file requires $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 22,400$ bits.

The reason of the efficiency of the variable-length encoding is that we use shorter codewords for more frequent characters.

Huffman Codes Example

- To use the variable-length codewords to encode, we need to define prefix codes, in which no codeword is also a prefix of some other codeword.
- When we use the prefix encoding, we can simply concatenate the codewords together without causing ambiguous.
- A binary tree can be used to help decode the variable length codewords.

Huffman Codes Example



The tree on the left is corresponding to the above example of variable-length codewords. If we have a binary string 001011101, then we can start from the root and following the labeled edges.

Edge 0 connects to the leave a, edges 101 connect to leave b, etc. So it is decodes as *aabe*.

Huffman Codes

Given a tree **T** corresponding to a prefix code, we can easily compute the number of bits required to encode a file.

For each character *c* in an alphabet *C*, let the attribute *c.freq* denote the frequency of *c* in the file and let $d_T(c)$ denote the depth of *c*'s leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character *c*.

The number of bits required to encode a file is thus

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

which we define as the cost of the tree T.

Constructing Huffman Code

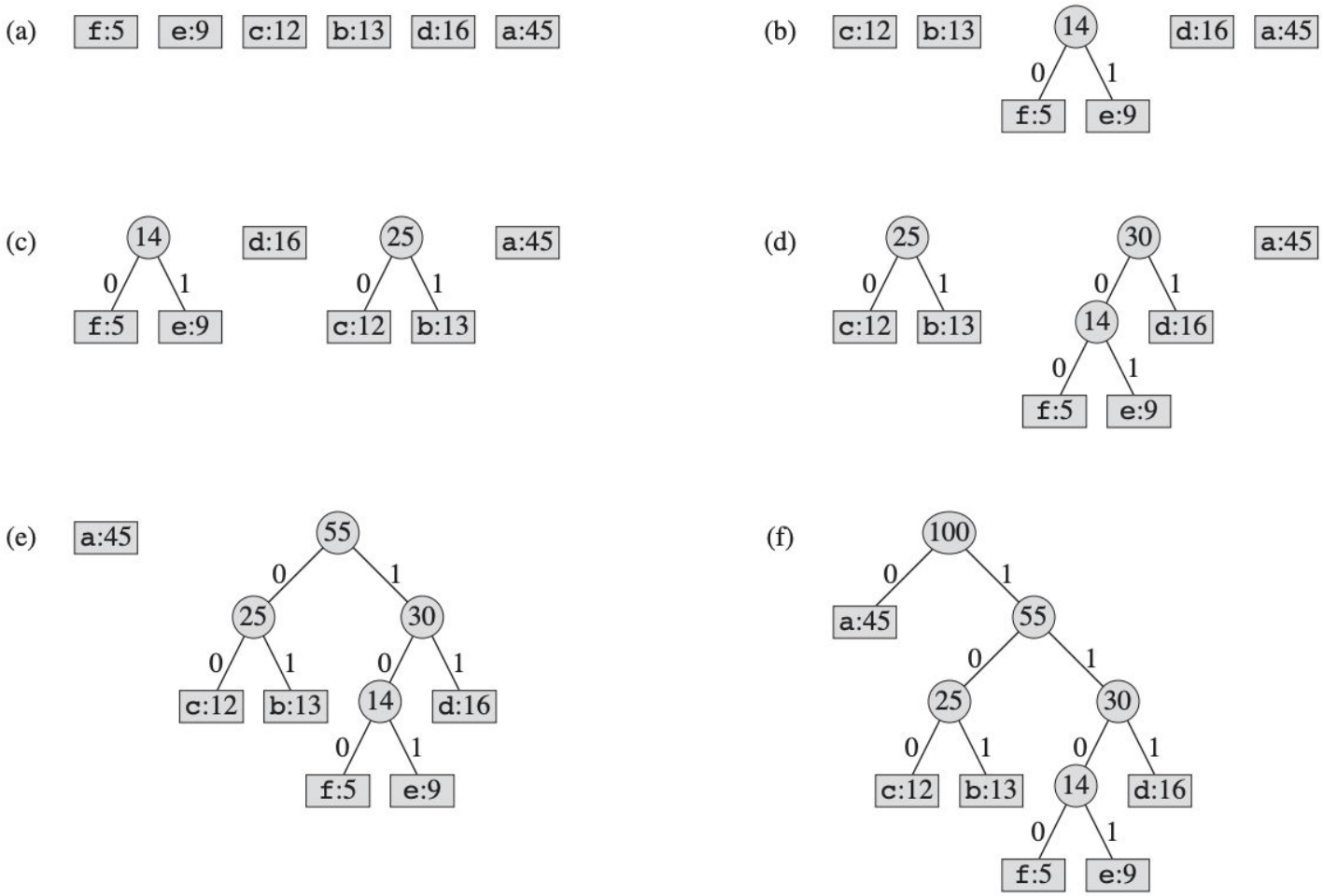
- Huffman coding is an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes". https://en.wikipedia.org/wiki/Huffman_coding
- It is a greedy algorithm that constructs an optimal prefix code called **Huffman code.**
- In the procedure, C is a set of n characters and each character $c \in C$ associated with an attribute c.freq.

Constructing Huffman Code

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

- The procedure Extract-Min(Q) removes and returns the element with minimum frequency from Q.
- Q is a min-priority queue.

Constructing Huffman Code



Huffman Code Analysis

- This procedure uses a bottom-up method.
- It begins with two least frequent characters as leaves and merge them to a node with the frequency the sum of these two leaves.
- The node is then put back to the pool. The for loop runs n times. We use a min-priority queue Q (minimum-heap: the first element is the minimum element), then the running time for the procedure will be O(n log n).

Huffman Code Analysis

Next we need to prove that the procedure really created an optimal code.

Lemma 1

If an optimal code for a file is represented by a binary tree, then the tree is full binary, that is, every non-leaf node has two children.

Proof.

Assume that there is an internal node A which has only one child B. Then we can remove the node A and the edge between A and B, and move B to the position of A. The resulting binary tree also represents the same file, but uses fewer bits. This is a contradiction. ■

Huffman Code Analysis

Lemma 2

Let C be an alphabet in which each character c ∈ C has frequency c.freq. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Huffman Code Analysis

Proof.

Let tree T be an optimal prefix code for the alphabet. Let a and b be two characters that are sibling leaves of maximum depth in T (Lemma 1 guarantees the existence of a and b). We may assume that a.freq ≤ b.freq and x.freq ≤ y.freq. We have x.freq ≤ a.freq and y.freq ≤ b.freq.

If x.freq = b.freq, then we have x.freq = b.freq = y.freq = a.freq, so the lemma is true.

So we assume that x.freq ≠ b.freq. Now we construct a tree T′ from T by exchanging the positions of a and x. Then exchange the positions of b and y to obtain a tree T″.

Huffman Code Analysis

Proof. (continue)

Since x ≠ b, x and y are sibling leaves in T″. By equation (1) the difference in cost between T and T′, D = B(T) − B(T′) is

$$\begin{aligned} D &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \\ &\geq 0. \end{aligned}$$

Similarly, we have B(T′) − B(T″) ≥ 0. Therefore B(T) ≥ B(T″). Since T is optimal, we must have B(T) = B(T″). So T″ is also optimal. ∎

Huffman Code Analysis

Next we consider the optimal substructure property for the optimal prefix codes.

Let C be an alphabet with frequency c.freq for each $c \in C$. Let x and y be two characters in C with minimum frequency. Let z be a new character with z.freq = x.freq + y.freq and $C' = (C \setminus \{x, y\}) \cup \{z\}$.

Lemma 3 Let T′ be any tree representing an optimal prefix code for alphabet C′. Then the tree T, obtained from T′ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Huffman Code Analysis

Proof. For each character c ∈ C\{x, y}, we have $d_T(c) = d_{T'}(c)$.

Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq), \end{aligned}$$

From which we have

$$B(T) = B(T') + x.freq + y.freq.$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T″ such that B(T″) < B(T).

Huffman Code Analysis

By Lemma 2, we may assume that T″ has x and y as siblings. Let T‴ be the tree T″ with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

yielding a contradiction to the assumption that T′ represents an optimal prefix code for C′. ■

From the above Lemmas, we obtain the following theorem.

Theorem: Procedure Huffman produces an optimal prefix code.

After Class

- After class: Part IV 16.3