EECS 3101A : Design and Analysis of Algorithms

Suprakash Datta Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/3101A`
Also on Moodle

Greedy Algorithms (Ch. 16)

Basic idea:

- In order to get an optimal solution, just keep grabbing what looks best.
- No backtracking (reversing earlier choices) allowed
- Local algorithm; often produces globally optimal solutions
- Typically the algorithm is simple. The proof that a greedy algorithm produces an optimal solution may be harder.

"Every two year old knows the greedy algorithm"

Greedy Algorithms - outline

In a loop:

- grab the next best object
- if it conflicts with committed objects, or fulfills no new requirements: Reject this object
- else: Commit to it.

Relationship with Dynamic Programming

- In DP, we do not know a priori what the best choice is
- For greedy algorithms we believe we know a best choice
- The proof of optimality is really a proof of the above claim
- This is why the book covers greedy after DP
- Optimal substructure and greedy choice are properties of the problem and a particular formulation .... if these properties hold, we know that DP and greedy algorithms (respectively) will be optimal for a problem

Example

- Making change problem: Find the minimum number of coins (i.e., quarters, dimes, nickels, and pennies) that total to a given amount.
- Greedy Algorithm: Keep grabbing the largest coin that keeps the solution cost less than or equal to the given amount.
- E.g.: Make change for 71 cents
  Solution: A subset of the coins that total 71 cents (25, 25, 10, 10, 1)
  Cost of Solution: The number of objects in solution or the sum of the costs of objects (5)

The greedy algorithm does not always work

- Problem: Find the minimum number of 4, 3, and 1 cent coins to make up 6 cents.
- Greedy solution: (4, 1, 1) cost 3
  Optimal Solution: (3,3) cost 2
- Lessons
  - Not all problems admit greedy algorithms.
  - For those that do, all greedy algorithms do not work.
  - The proof that a greedy algorithm works is subtle but essential.

Proving optimality of greedy algorithms

- Loop Invariant: There is at least one optimal solution consistent with the choices made so far
- Initially no choices have been made and hence all optimal solutions are consistent with these choices.
- It is often easier to carry out the proof by contradiction.
- For denominations 1,5,10,25 cents, prove optimality for amount C cents.

Making change:the greedy algorithm is optimal

- Consider solutions from greedy algorithm $Sol(G)$ and that from optimal $Sol(O)$. Sort both in decreasing order.
- Look at first place (k) where they differ. $Sol(G)$ MUST contain a coin of higher denomination
    - Case 1: $Sol(G)$ has a 5 c coin, $Sol(O)$ does not. $Sol(O)$ must make 5 c with 1 c; cannot be optimal.
    - Case 2: $Sol(G)$ has a 10 c coin, $Sol(O)$ does not. Must make 10 c with 5 c and 1 c. $Sol(O)$ cannot be optimal.
    - Case 3: $Sol(G)$ has a 25 c coin, $Sol(O)$ does not. If $Sol(G)$ has 2 or more 25 c coins, $Sol(O)$ must make 50 cents with 10c, 5c, 1c; cannot be optimal. Else $Sol(O)$ must use 1 or 2 or 3 or more 10c; in each case, $Sol(O)$ must be suboptimal.

Making change:the greedy algorithm is optimal - 2

Q: How is this consistent with "LI: There is at least one optimal solution consistent with the choices made so far."

- A: Take a different view of what we have done
- We proved that the next coin of $Sol(G)$ agrees with that of some $Sol(O)$
- More precisely, we proved that if no solution in $Sol(O)$ agrees with $Sol(G)$, then $Sol(O)$ cannot be optimal.

Another example: (Continuous) Knapsack problem

Problem: Given n commodities, with total values v_i dollars and weight w_i kg, and a knapsack that can carry maximum weight K, to put in the knapsack a set of items that maximize total value. You can take arbitrary fractions of any item.
Greedy algorithm:

- Sort in decreasing order of v_i/w_i
- Fill knapsack greedily
- Correctness: Compare $Sol(G)$ with $Sol(O)$, with both solutions sorted in decreasing order of v_i/w_i. If they differ, then prove that by replacing the object in $Sol(O)$ with the object in $Sol(G)$, we violate the optimality of $Sol(O)$.

More examples

- Huffman codes
- greedy scheduling

Data Compression

- When can you compress data?
- Key question: Do you allow information to be lost?
- Answer: depends on the application:
  Music/movies: small loss ok
  Text/data file transmission/storage: no loss permitted
- Lossy compression: uses signal processing techniques
  - Used in computer vision, image and speech processing
  - Utilizes the fact that some part of the data (signal) can be discarded without perceptible quality loss
- Lossless compression: Must ve able to reconstruct the exact data

Lossless Data Compression

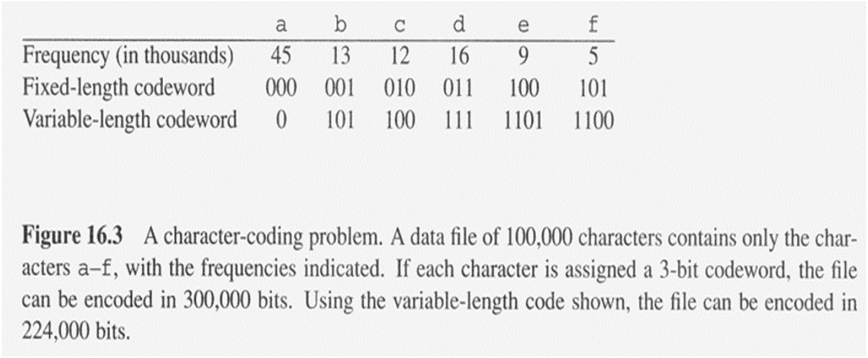
- Q: if you cannot throw away any “‘data”, how can you reduce its size?
- Answer: by removing redundancy in the data
- E.g.: My daughter sends me an sms “where are you?”
  I could answer “I am at York”, “at York”, “York”
- is this lossless compression?
- Aside: What about the obvious redundancy in language?
  (utilized by sms-language, e.g. I lv u, wt 4 me . . .)
  Why/when is redundancy useful?

Lossless Data Compression - 2

Assume: message is given, and cannot be altered

- Q: How can you reduce the size?
- Answer: variable length encodings
- If there are k characters in the alphabet, each character could be encoded using $\lceil \log k \rceil$ bits (fixed length encoding),
  or some characters could use 1 bit, some 2 bits, etc.
- Idea: the more frequent the letter, the shorter its encoding.
- Tradeoff: ease of parsing

Fixed and variable length codes



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

Fixed and variable length codes

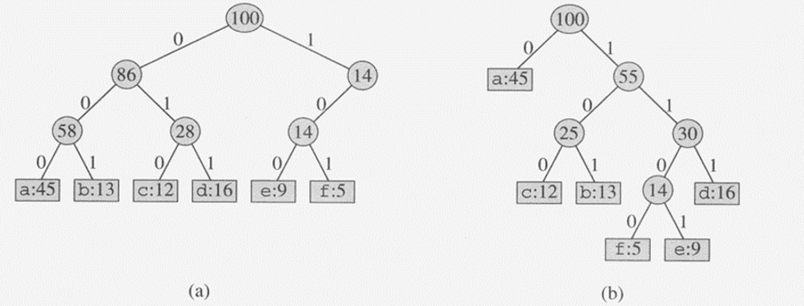
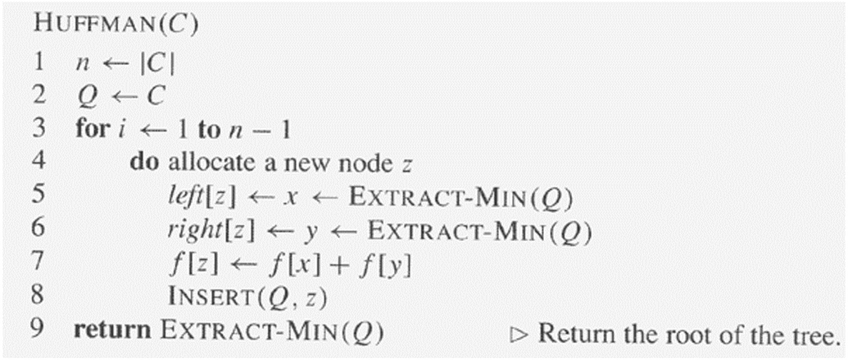


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . . . , f = 101. **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.

Huffman codes

- Want unique parse trees (PREFIX codes)
- Start with each character being a node, and set its weight to be its frequency
- Greedy strategy:
  select the two least weight nodes and make them children of the tree.
  Replace the nodes with a new node with the sum of the weights

Huffman codes - algorithm



```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n-1$
        do allocate a new node $z$
            $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
            $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
            $f[z] \leftarrow f[x] + f[y]$
            INSERT($Q$, $z$)
return EXTRACT-MIN($Q$)                    ▷ Return the root of the tree.
```

Huffman codes algorithm - example

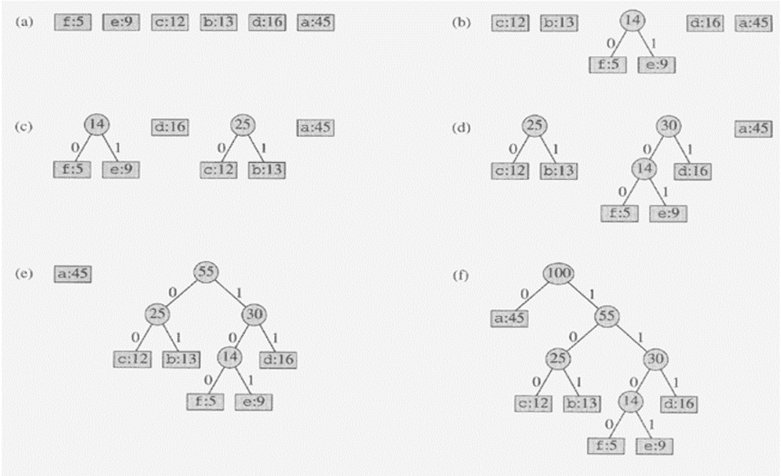


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children

Huffman codes - Optimality

- Requires a bit of work
- Can make the optimal solution more similar to greedy solution

Huffman codes - Optimality

Lemma: If x, y have the lowest frequencies, then there is an optimal prefix code in which they are sibling leaves.

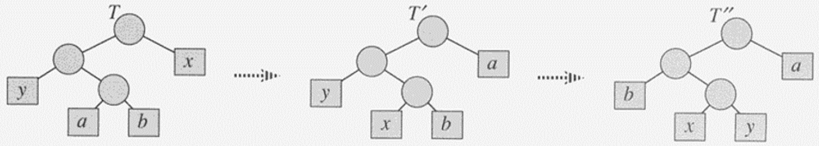
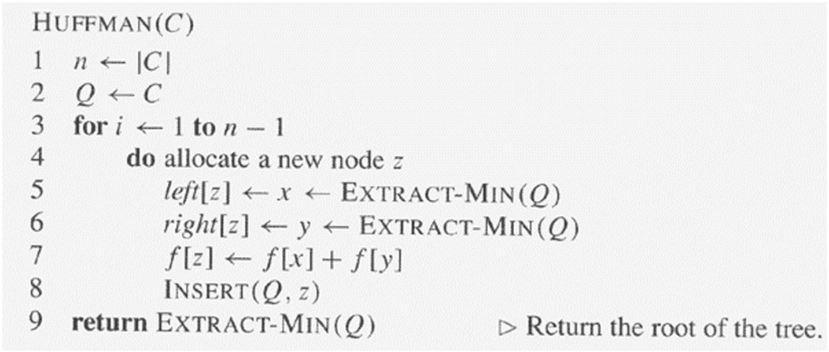


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two of the deepest leaves and are siblings. Leaves x and y are the two leaves that Huffman's algorithm merges together first; they appear in arbitrary positions in T. Leaves a and x are swapped to obtain tree T'. Then, leaves b and y are swapped to obtain tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

Huffman codes - Optimality

- If a, b have the lowest frequencies, then the greedy algorithm replaces them by another "character" c whose frequency is the sum of that of a, b.
- Inductive argument:
  - Suppose that the greedy algorithm is optimal for $k-1$ letter alphabets
  - For a k letter alphabet, it produces a tree S with x, y as children. Inductively, the tree S" obtained by fusing nodes x, y must be optimal
  - Suppose there exists a lower cost tree T. There must exist a tree T' with the same lower cost with x, y as children (previous lemma). Fuse nodes $x, y \in T'$ to get T". T" has strictly lower cost than S"; CONTRADICTION!.

Huffman codes - Running Time



```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n-1$
        do allocate a new node $z$
            $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
            $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
            $f[z] \leftarrow f[x] + f[y]$
            INSERT($Q, z$)
return EXTRACT-MIN($Q$)                ▷ Return the root of the tree.
```

- Naively, this requires $O(n^2)$ time. With priority queues implemented with heaps, Extract-Min takes logarithmic time. This gives total running time $O(n \log n)$.