CS624 - Analysis of Algorithms

Greedy Algorithms

October 21, 2019

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the greedy-choice property.
- When we have a choice to make, make the one that looks best right now.
- Make a locally optimal choice in hope of getting a globally optimal solution.

Greedy Strategy

- **The choice that seems best at the moment is the one we go with.**
- Prove that when there is a choice to make, one of the optimal choices is the greedy choice.
- Therefore, its always safe to make the greedy choice.
- Show that all but one of the subproblems resulting from the greedy choice are empty.

Example – Character Encoding

- A way to compress a text message.
- Example: 100,000 characters, with only the letters $\{a, b, c, d, e, f\}$.
- Fixed length coding:

character	code
a	000
b	001
c	010
d	011
e	100
f	101

We need three bits for each character, so the entire message will take 300,000 bits to encode. Can we do better?

Variable Length Code

- Using codes of variable lengths to encode characters.
- The length is proportional to the frequency of the character.
- Suppose the frequencies of the characters are as follows
- We could do better if a had a shorter code than f,

character	times used
a	45,000
b	13,000
c	12,000
d	16,000
e	9,000
f	5,000

Prefix Codes

- A set of codes such that no code is the prefix of another
- This is the only way we know when one code ends and another one begins. For example:

character	Frequency	code
a	.45	0
b	.13	101
c	.12	100
d	.16	111
e	.9	1101
f	.5	1100

The total size of the encoded message is now

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000 \text{ bits}$$

which is a significant improvement, even though some of the code words are actually longer this time.

Prefix Codes

- If we treat the frequency as the relative number of times a character appears in the code, then we can re-write the former equation as:

$$1(.45) + 3(.13) + 3(.12) + 3(.16) + 4(.09) + 4(.05) = 2.24$$

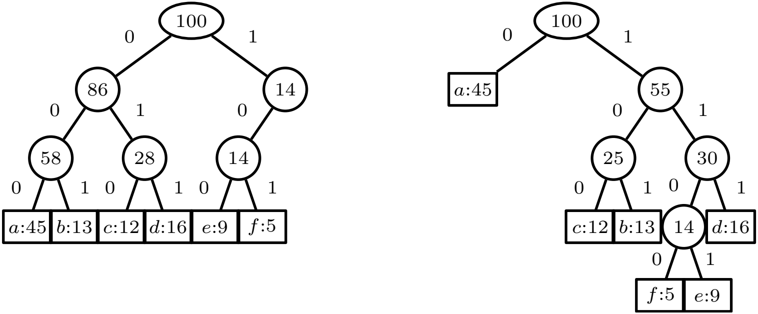
- This is the expected number (or "average" number) of bits per character – as opposed to 3 bits per character in our fixed-length encoding.

Prefix Codes

- We can measure the efficiency of a code by the expected number of bits per character.
- Let C be the set of characters.
- x is a variable that runs over the set of characters in C, and if $f(x)$ is the frequency of the character x, and if $length(x)$ is the length of the code word corresponding to x, then the average number of bits per character will be: $\sum_{x \in C} f(x) \cdot length(x)$
- Also – $\sum_{x \in C} f(x) = 1$
- Just think of the values of the function f as weights.
- Our problem is – given the set C and the frequency function f, find a prefix encoding that minimizes this value.

Decoding

- Retrieval of original text.
- The codes can be represented by binary trees (left: fixed code. Right: variable code).



Decoding

- The depth of a leaf in the tree is just the length of the code word for that character.
- Let $d_T(x)$ be the depth of a leaf node corresponding to the character x in the tree T.
- The average cost AC per character in the encoding scheme defined by the tree T is

$$AC(T) = \sum_{x \in C} f(x) d_T(x)$$

Finding the Optimal Encoding

Exhaustive search:

- Enumerate all possible prefix trees and find the one with the smallest average cost per character.
- Without performing an exact analysis, the cost of this algorithm would be exponential in the number of characters, and therefore completely useless.

Finding the Optimal Encoding

Lemma

If T is the tree corresponding to an optimal prefix encoding, and if T_L and T_R are its left and right subtrees, respectively, then T_L and T_R are also trees corresponding to optimal prefix encodings.

Proof.

- Let us say that C_L is the set of characters that are leaf nodes in T_L and similarly for C_R and T_R.
- If $x \in C_L$, then certainly $d_{T_L}(x) = d_T(x) - 1$, and the same is true for C_R and T_R.

$\square$

Finding the Optimal Encoding

Proof (cont.)

- Therefore we can see from our basic cost formula that

$$
\begin{aligned}
AC(T) &= \sum_{x \in C} f(x) d_T(x) \\
&= \sum_{x \in C_L} f(x)\big(d_{T_L}(x) + 1\big) + \sum_{x \in C_R} f(x)\big(d_{T_R}(x) + 1\big) \\
&= \sum_{x \in C_L} f(x) d_{T_L}(x) + \sum_{x \in C_R} f(x) d_{T_R}(x) + \sum_{x \in C} f(x)
\end{aligned}
$$

- If T_R were not an optimal encoding tree, then we could replace it by a more efficient one (with the same leaves and the same frequencies), and this would show in turn that T could not have been optimal, a contradiction.

$\square$

Finding the Optimal Encoding

Corollary

If T is the tree corresponding to an optimal prefix encoding, then every subtree of T also corresponds to an optimal prefix encoding.

Proof.

This follows immediately by induction. □

- This lemma expresses the fact that the problem of finding an optimal prefix code has the *optimal substructure property*.
- This means that we could write a recursive algorithm for it.

Finding the Optimal Encoding – Recursive Algorithm

- Start with a worklist consisting of n trees, each tree consisting of exactly 1 character.
- From these trees construct other trees bottom-up and add them to the worklist.
- As each new tree is constructed, check the worklist to see if a tree with the same leaves is in it.
- Keep the tree with the smallest cost in the worklist and remove any others with the same set of leaves.
- At the end of this process there will be one tree in the worklist that contains all the characters in C as leaves, and that tree represents an optimal encoding.
- This algorithm will definitely give the correct answer, but is still inefficient, although it is better than exhaustive search.

- The optimal substructure property should remind us of dynamic programming.
- If there were also an *overlapping subproblems* property of this problem, we could try such a solution.
- Actually we have something even better: We don't actually have to form all possible trees on the way up and check them all.
- We actually can tell at each step exactly which tree to form.

Finding the Optimal Encoding

Lemma

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.

Proof.

- Suppose that the tree T represents an optimal prefix code for our problem.
- If x and y are sibling nodes of greatest depth, then we are done.
- Otherwise, suppose that p and q are sibling nodes of greatest depth.
- We will exchange x and p, and we will also exchange y and q.

$\square$

Finding the Optimal Encoding

Proof (cont.)

- We know that

$$
\begin{aligned}
d_T(x) &\leq d_T(p) \\
d_T(y) &\leq d_T(q) \\
f(x) &\leq f(p) \\
f(y) &\leq f(q)
\end{aligned}
$$

- Suppose the tree T, after these two switches, is turned into the tree T'. Then we have:

$$
\begin{aligned}
d_{T'}(x) &= d_T(p) \\
d_{T'}(p) &= d_T(x) \\
d_{T'}(y) &= d_T(q) \\
d_{T'}(q) &= d_T(y)
\end{aligned}
$$

$\square$

Finding the Optimal Encoding

Proof (cont.)

$$
\begin{aligned}
AC(T') - AC(T) &= \sum_{z \in C} f(z)\big(d_{T'}(z) - d_T(z)\big) \\
&= f(p)\big(d_{T'}(p) - d_T(p)\big) + f(x)\big(d_{T'}(x) - d_T(x)\big) \\
&+ f(q)\big(d_{T'}(q) - d_T(q)\big) + f(y)\big(d_{T'}(y) - d_T(y)\big) \\
&= f(p)\big(d_T(x) - d_T(p)\big) + f(x)\big(d_T(p) - d_T(x)\big) \\
&+ f(q)\big(d_T(y) - d_T(q)\big) + f(y)\big(d_T(q) - d_T(y)\big) \\
&= \big(f(p) - f(x)\big)\big(d_T(x) - d_T(p)\big) \\
&+ \big(f(q) - f(y)\big)\big(d_T(y) - d_T(q)\big) \\
&\leq 0
\end{aligned}
$$

so $AC(T') \leq AC(T)$, which shows that T was not an optimal tree to begin with, and this is a contradiction. □

Finding the Optimal Encoding – Huffman's Algorithm

- We can start out with our initial worklist, and we can take two nodes of smallest frequency and build a tree from them (in which they are the two leaves).
- Then we delete those two nodes from the worklist, because we know that they will definitely be part of the little tree we have just constructed – we will never have to look at them again.
- By exactly the same argument, we can take the two elements of the worklist that are now of smallest cost, and build a little tree from them, and then throw them away.
- When we are done, we have the tree we are looking for.
- The algorithm: We keep a minimum-priority queue Q of subtrees. Q initially consists of the n characters. The priority of any element in Q will be the cost of that subtree.

Finding the Optimal Encoding – Huffman's Algorithm

Algorithm 1 Huffman(C)

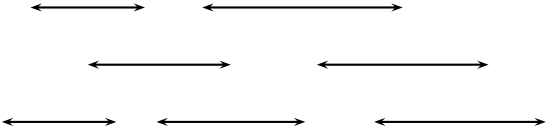
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1 \ldots n-1$ **do**
    allocate a new node z
    $left[z] \leftarrow ExtractMin(Q)$
    $right[z] \leftarrow ExtractMin(Q)$
    $f[z] \leftarrow f[x] + f[y]$
    $Insert(Q.z)$
end for
return $ExtractMin(Q)$ //Return the root of the tree.

- This algorithm works even better than a dynamic programming algorithm: we don't have to memoize intermediate results for later use.
- We know exactly at each step what we need to do.
- This is called a "greedy" algorithm because we chose the locally best solution at each step.
- In effect, we act as if we were "greedy".
- What is is the best at each step is guaranteed (in this case) to turn to out to be the best overall.

Another Example – Activity Selection

- **Input:** Set S of n activities – $\{a_1, a_2, \ldots, a_n\}$.
- s_i = start time of activity i.
- f_i = finish time of activity i.
- **Output:** Subset A of maximum number of compatible activities.
- Two activities are compatible, if their intervals do not overlap.

Example (activities in each line are compatible):



Optimal Substructure

- Assume activities are sorted by finishing times – $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
- This generates two subproblems:
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
- The solutions to the two subproblems must be optimal.
- Prove using the cut-and-paste approach.

Optimal Substructure

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let c[i,j] = size of maximum-size subset of mutually compatible activities in S_{ij}.
- The recursive solution is:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

- The problem also exhibits the greedy-choice property.
- There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
- It can be proved easily (how?).
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Recursive Solution

Algorithm 2 Recursive-Activity-Selector (s, f, i, j)

$m \leftarrow i + 1$
while $m < j$ and $s_m < f_i$ **do**
    $m \leftarrow m + 1$
end while
if $m < j$ **then**
    return $a_m \cup Recursive - Activity - Selector(s, f, m, j)$
else
    return $\emptyset$
end if

- Top level call: $Recursive - Activity - Selector(s, f, 0, n + 1)$
- Complexity??
- See text for iterative version

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.
- Example: Sorting activities by finish time.