Greedy Algorithms

CS 624 — Analysis of Algorithms

April 2, 2024

Greedy Algorithms

Greedy algorithms, like dynamic programming, are used to solve optimization problems.

- Problems exhibit optimal substructure, as in DP.
- Problems also exhibit the greedy-choice property:

  Instead of having to *search* over results of sub-problems, we have a criterion (a locally optimal choice) that lets us *predict* the choice that leads to a globally optimal solution.

Character Encoding

Goal: Encode a text message as a bit string.

The message is 100,000 characters, with only the letters $\{a, b, c, d, e, f\}$.
The frequency of each character is given by the following table:

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

Fixed-Length Encoding

An example fixed-length encoding:

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

Idea: use a *variable-length* encoding, where *more frequent characters* are given *shorter codes*.

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

For example "a" should have a shorter code than "f".

Prefix Codes

Definition (Prefix Code)

A **prefix code** (aka **prefix-free code**) is a mapping from an alphabet to codes (typically, bit strings), such that no code is a prefix of another code.

This property allows *variable-length* codes to be uniquely parsed.

Prefix Codes

For example:

character	frequency	code
a	.45	0
b	.13	101
c	.12	100
d	.16	111
e	.09	1101
f	.05	1100

The total size of the encoded message is now

$$(1(.45) + 3(.13) + 3(.12) + 3(.16) + 4(.09) + 4(.05)) \cdot 100,000 \text{ bits}$$
$$= 224,000 \text{ bits}$$

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

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

The **efficiency** of a code is the expected number of bits per character (given a distribution of characters).

- Let C be the set of characters.
- Let $f(x)$ be the frequency of the character $x \in C$.
  Assume that $\sum_{x \in C} f(x) = 1$.
- Let $length(x)$ be the length of the code word for $x \in C$.

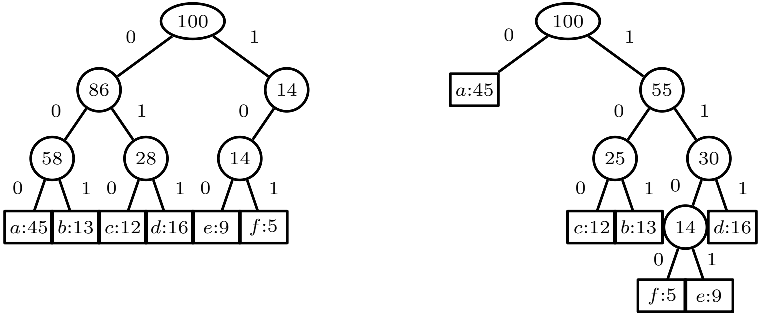
Then the average number of bits per character for this encoding is

$$\sum_{x \in C} f(x) \cdot length(x)$$

Our problem is this: Given the set C and the frequency function f, find a prefix code that minimizes this value.

Codes as Binary Trees

Codes can be represented by binary trees.



Left: fixed code, right: variable code.

Codes as Binary Trees

- 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

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

Exhaustive Search

Strategy #1: 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.

Optimal Substructure

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 for the alphabets they cover.

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 $d_T(x) = d_{T_L}(x) + 1$, and likewise if $x \in C_R$, then $d_T(x) = d_{T_R}(x) + 1$.

Optimal Substructure

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. □

Optimal Substructure

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. □

Since this problem has the optimal substructure property, we could use dynamic programming to solve it recursively.

Recursive (Top-Down) Algorithm

Strategy #2: Recursive algorithm

- For a given alphabet of characters C where $|C| > 1$, choose a partition of C into two non-empty sets C_L and C_R.
- Solve the subproblems corresponding to C_L and C_R recursively, and form a binary tree from the results.
- Minimize over $AC(T)$ for every candidate T.
- There are overlapping subproblems when we hit the same subset of C along different paths.

Analysis:

- A subproblem is identified by a non-empty subset of C.
- If $|C| = n$, then there are $2^n - 1$ subproblems.

Top-Down vs Bottom-Up

Strategy #2′: Bottom-up algorithm

- Build the tree from the leaves up.
- This corresponds to filling in the memo table in increasing order by subproblem cardinality.
- Initialize the table for each single leaf (cardinality 1).
- Next fill in the table for all pairs of leafs (cardinality 2).
- And so on, until we get to cardinality n, which has the original C mapped to the solution for the original problem.

Analysis:

- Memo table still has $2^n - 1$ entries, must fill all of them.

Greedy Choice Property

Lemma (Greedy Choice Property)

Let x and y be two characters in C having the lowest frequencies. 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.

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:

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

Finding the Optimal Encoding

Proof (cont.)

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

so $AC(T') \leq AC(T)$. Since T was assumed to be optimal, it must be that $AC(T') = AC(T)$ and so T' is optimal and has x, y in the positions described by the lemma. □

Strategy #3: Iterative pairing

- Start with a set of leaf nodes, one for each character in C.
- Select the nodes with the least frequencies. Remove them from the set, pair them to create a new tree, and add the new tree.
- Repeat the process: select the two *trees* with least *total frequencies*, remove them, pair them, and add the new tree.
- Stop when there is a single tree left. That is the solution.

Worry: The greedy choice lemma guaranteed that doing this for the least-frequency *characters* would work, but it said nothing about repeating the process on intermediate trees.

Optimal Substructure, Revisited

Lemma (Optimal Substructure, v2)

Let C be an alphabet and $f : C \to \mathbb{R}^+$ be frequencies. Let $x, y \in C$ be the characters with least frequencies.

Let $C' = (C - \{x, y\}) \cup \{z\}$, where $z \notin C$, and set $f(z) = f(x) + f(y)$.

Suppose that T' is a tree representing an optimal prefix code for C'. Define T by replacing z in T' with a node pairing x and y.

Then T represents an optimal prefix code for C.

The proof is in the textbook (p435, Lemma 16.3).

Note: this is a very different optimal substructure property than the first one we showed. It is specialized to the single subproblem generated by the greedy choice.

Huffman's Algorithm

Algorithm 1 Huffman(C)

$n \leftarrow |C|$
$Q \leftarrow \mathrm{BuildMinHeap}(C)$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    $z \leftarrow$ allocate new node
    $\mathrm{left}[z] \leftarrow \mathrm{ExtractMin}(Q)$
    $\mathrm{right}[z] \leftarrow \mathrm{ExtractMin}(Q)$
    $f[z] \leftarrow f[x] + f[y]$
    $\mathrm{Insert}(Q, z)$
end for
return $\mathrm{ExtractMin}(Q)$

Analysis:

- $O(n)$ for $\mathrm{BuildMinHeap}$
- $n - 1$ loop iterations
  - $O(\log n)$ for $\mathrm{ExtractMin} \times 2$
  - $O(\log n)$ for Insert
- total: $O(n \log n)$

Huffman's Algorithm

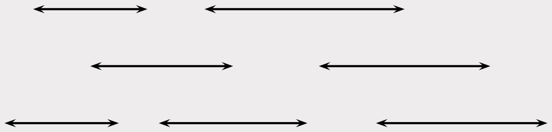
- This algorithm works much more efficiently than a dynamic programming algorithm.
  - It avoids *searching*. We know at each step what to do.
  - It does not need to memoize intermediate results.
- This is called a "greedy" algorithm because we chose the locally best solution at each step.
- What is is the best at each step is guaranteed (in this case) to turn to out to be the best overall.

Activity Selection

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

Example

Overlapping lines represent incompatible activities:



Optimal Substructure

Optimal Substructure for Activity Selection

Assume activities are sorted by finishing times: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Suppose A is an optimal solution for activities $S = \{a_1, \ldots, a_n\}$, and suppose $a_k \in A$.

This generates two subproblems:

- Let $S_L \subseteq \{a_1, \ldots, a_{k-1}\}$ be the set of activities ending before a_k starts.
- Let $S_R \subseteq \{a_{k+1}, \ldots, a_n\}$ be the set of activties starting after a_k ends.

Then $A_L = A \cap S_L$ is an optimal solution for S_L, and $A_R = A \cap S_R$ is an optimal solution for S_R.

So $A = A_L \cup \{a_k\} \cup A_R$.

Optimal Substructure

Let S_{ij} be the subset of activities in S that start after a_i finishes and finish before a_j starts.

Let $c[i,j]$ be the size of maximum-size subset of mutually compatible activities in S_{ij}.

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

Can we do better?

Greedy Choice Property

This problem also exhibits the greedy choice property.

Greedy Choice Property for Activity Selection

There is an optimal solution to the subproblem S_{ij} that includes the activity with the *earliest finish time* in the set S_{ij}.

Proof.

(why?) □

Greedy Choice Property

Thus:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ c[k,j] + 1 & \text{where } k = \min\{k \mid a_k \in S_{ij}\} \end{cases}$$

(Recall that we are assuming that activities are sorted by finish time.)

That is, for a subproblem S_{ij}:

- Make the greedy choice *without* solving subproblems first and evaluating them. (No search!)
- Solve the (single) 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 $\mathrm{SelectActivities}(i,j)$

$m \leftarrow i + 1$
while $m < j$ **and** $s_m < f_i$ **do**
    $m \leftarrow m + 1$
end while
if $m < j$ **then**
    return $\{a_m\} \cup \mathrm{SelectActivities}(m,j)$
else
    return $\emptyset$
end if

What is the running time?

Treat s, f, a as global.

Initial call:

$$\mathrm{SelectActivities}(0, n+1)$$

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 yield an optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.

  For example: Sorting activities by finish time.