The Greedy Method

February 26, 2018

Contents

1 Introduction 2

2 The Basics of the Greedy Method 2

3 Greedy Method Vs. Dynamic Programming 2

4 The Problems 5

4.1 The Activity Selection Problem 5

4.1.1 The Optimal Substructure 5

4.1.2 The Greedy Choice 5

4.2 Huffman Codes 6

4.2.1 Prefix Codes 6

4.2.2 Properties of the Binary Tree Representation 7

4.2.3 Constructing the Huffman Code 8

4.2.4 Correctness of Huffman's algorithm 9

1 Introduction

All the optimization problems are solved through a series of steps and decisions. There is a class of optimization problems where the multiple decision problem can be reduced to only one, the greedy option. This basically is telling us that the greedy method will be faster than dynamic programming. Examples where the greedy strategy can be applied are:

1. Minimum Spanning Tree problems.
2. Shortest paths from a single source.
3. Set Covering (Chvátal's Heuristic).

2 The Basics of the Greedy Method

Then, the basic steps of the greedy method are:

1. Determine the optimal substructure
2. Develop a recursive solution
3. Prove that the greedy choice is possible, then only one subproblem needs to be solved.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive greedy algorithm.
6. Convert to an iterative version

It is clear that it looks quite similar to the dynamic programming.

3 Greedy Method Vs. Dynamic Programming

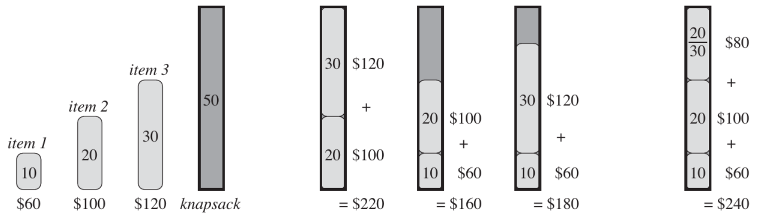
Therefore, you could mistakenly think that the it is possible to solve dynamic programming problems using greedy strategies. It can be dangerous to think that way because the although greedy problems can be solved by dynamic programming (This being an overkill). Dynamic programming where there is not a greedy property cannot be solved by greedy methods. Look at this two problems to understand this

1. The 0-1 knapsack problem. You have a thief trying to robber a store. He uses a knapsack to carry the items, where each item i is worth v_i and has a weight w_i. The thief can carry an item or not to carry it (0-1 choice). It needs to decide which ones are worth their time, knowing that the knapsack can only withstand a W total weight.
2. The fractional knapsack has the same setup, but the thief can carry a fraction of each item.

Both problems show optimal-substructure:

1. If W is the most optimal valuable load that can be carry in the 0-1 problem. If we remove j from this load, the remaining load is the most valuable load for a weight $W - w_j$ knapsack when considering $n-1$ items (Excluding j) by the cut-and paste strategy.
2. In the fractional problem, now if we remove a weight w, the remaining load must be the most valuable load weighing at most $W - w$ from the $n - 1$ original item plus the $w_j - w$ remaining weight from which the w weight was subtracted.

The first one cannot be solved by greedy choice by simply showing the following counterexample:



With respect to the fractional knapsack, we have the following:

- The Greedy Choice: sort the items by using the values $\frac{v_i}{w_i}$. Then use this sorting to put fractional stuff in the knapsack beginning at the maximal element in the sorting.

Theorem

The greedy choice, which always selects the object with better ratio value/weight, always finds an optimal solution to the Fractional Knapsack problem.

Proof:

Let $X = (x_1, ..., x_n)$ be the solution computed by the greedy algorithm.

Case 1

If $x_i = 1$ for all i, the solution is optimal.

Case 2

Otherwise, let j be the smallest value for which $x_j < 1$. It is more, according with the algorithm we have:

- If $i < j$ then $x_i = 1$,
- If $i = j$ then $x_i \leq 1$
- if $i > j$ then $x_i = 0$.

Note: We need to try to prove that the value is optimal under greedy choice.

Furthermore, $\sum_{i=1}^{n} x_i w_i = W$. Now, let $Y = (y_1, ..., y_n)$ be any feasible solution, we have that:

$$\sum_{i=1}^{n} y_i w_i \leq W = \sum_{i=1}^{n} x_i w_i$$

Therefore

$$\sum_{i=1}^{n} x_i w_i - \sum_{i=1}^{n} y_i w_i = \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0.$$

Let $V(Y)$ denote the total value of a feasible solution, thus we have

$$V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}.$$

If $i < j$, $x_i = 1$ and $y_i \geq 1$, then $x_i - y_i \geq 0$. Additionally, we have that

$$\frac{v_i}{w_i} \geq \frac{v_j}{w_j} \tag{1}$$

because $i < j$. Thus, we have

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j} \tag{2}$$

if $i = j$, then $x_i < 1$ and $y_i \leq x_i \Rightarrow x_i - y_i \geq 0$ (If it was no like that we will not have the $\sum_{i=1}^{n} x_i w_i - \sum_{i=1}^{n} y_i w_i \geq 0$.), thus

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j} \tag{3}$$

If $i > j$, $x_i = 0$ and y_i =0 or $y_i \neq$0, then

$$x_i - y_i \leq 0, \ \frac{v_i}{w_i} \leq \frac{v_j}{w_j} \tag{4}$$

Therefore, we have

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j} \tag{5}$$

Finally,

$$V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i} \geq \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_j}{w_j} = \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \tag{6}$$

Now,

$$\frac{v_j}{w_j} \geq 0$$

$$\sum_{i=1}^{n} (x_i - y_i)\, w_i \geq 0$$

Thus, $V(X) \geq V(Y)$ for all Y. Therefore X is an optimal solution. Q.E.D.

4 The Problems

4.1 The Activity Selection Problem

In the activity selection problem, we have a set of possible activities $S = \{a_1, a_2, ..., a_n\}$ where each activity is such that $a_i = [s_i, f_i)$. We say that two activities a_i, a_j are overlapping, if $[s_i, f_i)$ and $[s_j, f_j)$ are not overlapping. In the activity selection problem, we want the biggest subset of activities that can use certain resource without overlapping. For this, we will assume that

$$f_1 \leq f_2 \leq ... \leq f_n.$$

4.1.1 The Optimal Substructure

Lets to denote S_{ij} the set of activities that start after activity a_i finishes and that finishes before activity a_j start. Now suppose that the set A_{ij} denotes the maximum set of compatible activities for S_{ij}. In addition assume that A_{ij} includes some activity a_k. Then imagine that $a_k$belong to some optimal solution. Then, we need to find the optimal solutions for S_{ik} and S_{kj}.

Proof

For the sake of consistency then we have that $A_{ik} = S_{ik} \cap A_{ij}$ and similarly $A_{kj} = S_{kj} \cap A_{ij}$. Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ or $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$. Then, we can use the cut-and-paste arguments to prove that there is an optimal sub-structure:

- Assume that exist A'_{ik} such that $\left|A'_{ik}\right| > |A_{ik}|$. Then $\left|A'_{ik}\right| + |A_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$, which is a contradiction.

Therefore, we have the optimal-substructure.

4.1.2 The Greedy Choice

Now, the greedy choice is the following one:

- Choose activities for the solution with the earliest finishing time.

This can be proved using:

Theorem 16.1

Consider any nonempty subproblem $S_k = \{a_i \in S | s_i > f_k\}$, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k , and let a j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A_k^{'} = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j . The activities in $A_k^{'}$ are disjoint, which follows because kthe activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since$|A_k^{'}| = |A_k|$, we conclude that $A_k^{'}$ is a maximum-size subset of mutually compatible activities of S_k , and it includes a_m .

4.2 Huffman Codes

This is an effective way of compressing text information. Imagine having 1,000,000-character data file, and we want to store it compactly. In addition, we have the following distribution of character in the hundred of thousands (Table 1). Because we only have six different characters, we could decide to use a fix-length character code after all we can represent 2^3 different characters with a fixed one (Third row in the table 1). When seeing how many bits are required to store all the characters:

$$1,000,000 \times 3 = 3,000,000 \text{ bits} .$$

	a	b	c	d	e	f
Frequency	450,000	130,000	120,000	160,000	90,000	50,000
Fixed-Leng cw	000	001	010	011	100	101
Vari Leng cw	0	101	100	111	1101	1100

Table 1: Distribution of characters in the text and their codewords.

Then, it is clear that using a fix-length code is not a good idea. Therefore, we require a variable one. An example of that one is in (Fourth row table 1):

$$(45 \times 1 + 13 \times 3C + 12 \times 3 + 16 \times 3 + 94 + 5 \times 4) \times 10,000 = 2,240,000 \text{ bits} .$$

4.2.1 Prefix Codes

It has been show that prefix codes:

- Codes in which no codeword is also a prefix of some other codeword.

using prefix code can always achieve the optimal data compression among any character code. Therefore, we will restrict our study to the generation of prefix codes. In addition, they have the following nice properties

- Easy to decode.
- They are unambiguous. For example in our example the string 001011101 transform as $0 \circ 0 \circ 101 \circ 1101 = aabe$.

We require a nice representation for picking up the prefix code. Binary trees are perfect in that regard because once a branch is picked, we never go back until we have the decoded character. Example of this is in (Fig. 1).

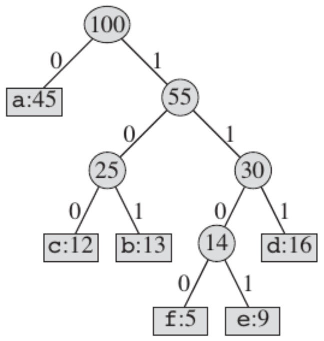


Figure 1: Binary tree for the variable prefix code in table 1

4.2.2 Properties of the Binary Tree Representation

As we can prove (Exercise 16.3-2) an optimal code for a text is always represented by a full binary tree (Each non-leaf node has two children). An example of a non optimal representation is in (Fig.).

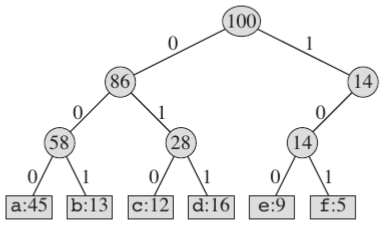


Figure 2: No optimal tree in our problem.

Now, given that we will concentrate our attention for the prefix codes to full binary tree, it is possible to say the following: Given that C is the alphabet for the text file

1. The tree for the optimal prefix code has $|C|$ leaves.
2. The number of internal leaves is $|C| - 1$.
3. Each character x at the leaves has a depth $d_T(x)$ which is the length of the codeword.

Knowing the frequency of each character and the tree T representing the optimal prefix encoding, we can define the number of bits necessary to encode the text file:

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

Which can be defined as the cost of the tree.

4.2.3 Constructing the Huffman Code

Basically, before proving the correctness of the greedy choice, we will comment on it and talk about the greedy algorithm generated using it. The greedy choice is as follow:

- You start with an alphabet C with an associated frequency for each element in it.
- Use the frequencies to build a min priority queue.
- Subtract the two least frequent elements (Greedy Choice)
- Build a three using as children the two nodes of the subtrees extracted from the min priority queue. The new root holds the sum of frequencies of the two subtrees.
- Put it back into the Priority Queue.

The final algorithm looks like

Algorithm 1 The Huffman code algorithm

```
Huffman(C)
        n=|C|
        Q = C
        for i = 1 to n-1
                allocate new node z
                z.left = x = Extract-Min(Q)
                z.right = y = Extract-Min(Q)
                z.freq = x.freq+y.freq
                Insert(Q,z)
        return Extract-Min(Q) // return root of the Huffman Tree
```

In the following example (Fig.), we can see the construction of a Huffman tree.

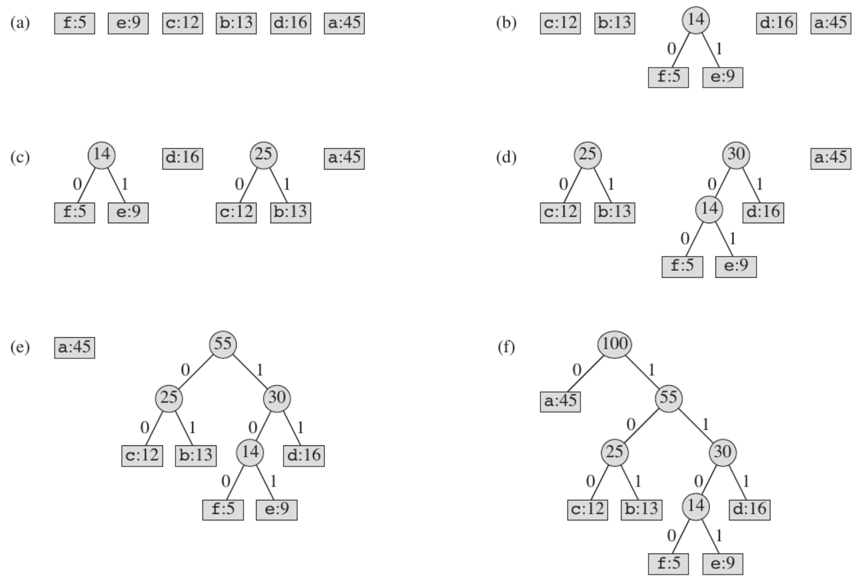


Figure 3: Example Huffman Tree

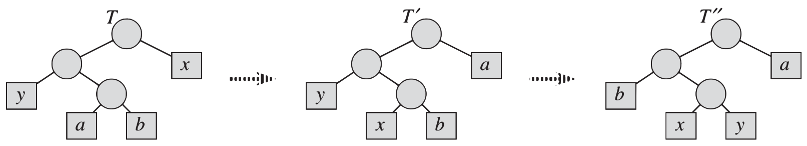
4.2.4 Correctness of Huffman's algorithm

From the following lemmas, we can prove the correctness of the algorithm.

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ 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.

Proof We want something like this:



Given a and b characters at sibling leaves of maximum depth in an optimal prefix code tree T. Without loss of generality, we can assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. Then, $x.freq \leq a.freq$ and $y.freq \leq b.freq$. Thus, we have two case

Case I

if $x.freq = b.freq$, then $a.freq = b.freq = x.freq = y.freq$ the lemma would be trivially true.

Case II

if $x.freq \neq b.freq$ (i.e. $x \neq b$). Now, we exchange nodes a and x to produce a new tree T'. Then, we use the equation of cost to measure the difference between T and T':

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

In a similar way we can create T'' from T' by exchanging y and b. Thus $B(T') - B(T'') \geq 0$. Following these inequalities, we have that $B(T'') \leq B(T') \leq B(T)$ and $B(T) \leq B(T'')$, this means that $B(T'') = B(T)$.

Now, given this lemma, we can prove that the merging of nodes into trees using the least frequent trees has a sub-optimal structure.

Lemma 16.3

Let C be a given alphabet with frequency c:freq defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the 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.

Proof: For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c) \Rightarrow c.freq \times d_T(c) = c.freq \times d_{T'}(c)$, but $d_T(x) = d_T(y) = d_{T'}(z) + 1$, thus

x.freq

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

Using this we can conclude $B(T') = B(T) - (x.freq + y.freq)$. Now use a contradiction to prove the lemma:

1. First assume that T does not represent an optimal prefix code for C.
2. Then, there is a tree T'' such that $B(T'') \leq B(T)$, where (By Lemma 16.2) x and y are siblings.
3. Now, we build T''' by substituting x and y by z assigning the sum of frequencies at x and y to it.
4. Finally, we have that

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

A contradiction because T''' and T' represents trees for C', in addition that T' is optimal. Thus, T represents an optimal prefix code for C.

Theorem 16.4

Procedure HUFFMAN produce an optimal prefix code.

Proof: Directly from the previous lemmas.