Greedy Algorithms

1. Overall Approach

Consider a problem that, like a dynamic programming problem, is an optimization problem. You want a solution with some property maximized (or minimized). The **greedy algorithm** approach is to start small and act, at each step, in a fashion that seems like the best thing to do based on the limited information you have at the time. In other words, you create a series of "local" optimized solutions by pursuing your own [selfish] interests based on the local situation. At the end, a successful greedy algorithm will result in a "global" optimized solution.

Example: A plate of cookies is passed around and around the table. If you always grab the biggest cookie left on the plate, at the end have you eaten the most? Is your locally greedy strategy going to serve your overall maximization goal?

The problem is that not all greedy approaches lead to overall optimal solutions, so when you pursue a greedy algorithm strategy, you also have to prove that it actually works. In order for a greedy algorithm to lead to an optimal solution, it must have the **greedy-choice property**, namely that each greedy choice is part of some one optimal solution. A greedy algorithm never reconsiders past choices. We'll look at a couple of examples.

2. Dijkstra's Algorithm

The Problem: The first example is one you should be familiar with from CSCI 36200, namely Dijkstra's algorithm for shortest paths in a weighted, connected graph with nonnegative weights. Some review is in order.

A **graph** is a collection of nodes and arcs, where an arc connects two endpoints and can be represented by its endpoints {i, j}; i and j are **adjacent nodes**. A **connected graph** is one where there is a path (a walk along arcs and through nodes) from every node to every other node. A **weighted graph** is one where there is a weight (distance, time, cost, etc.) associated with each arc.

The **single-source shortest path problem** is to find the path from a source node x to any other node that minimizes the weight of the path. [Note that shortest path means the one with minimal weight, which could be time, cost, or some other criterion – it's not necessarily distance.] Dijkstra's algorithm solves this problem. The shortest-path problem has many practical applications, such as how to route messages through a computer network, how to route trucks from a distribution point along highways, etc. In a computer network, each node would use Dijkstra's algorithm to find the shortest path from itself to any other node in the network, and from this construct its own routing table, that is, which adjacent node to which to forward a message based on the destination node of the message. This requires that each node have complete knowledge of the network. As conditions in the network change (for example a node goes down, or traffic clogs one of the links), each node must run Dijkstra's algorithm again to compute the new routing table.

Before getting started, it is useful to review the data structures available for representing graphs. The two common structures described below each assume some arbitrary numerical ordering has been imposed on the graph nodes.

a) an **adjacency matrix**

In an n-element graph, this is an n × n matrix where entry [i, j] is the weight of an arc between node i and node j if such an arc exists, and some "infinity" value if no such arc exists.

b) an **adjacency list**

In an n-element graph, this is a 1× n array of lists, where list[i] includes all nodes adjacent to node i.

The Algorithm: To describe Dijkstra's algorithm, we'll use an adjacency matrix representation. Dijkstra's algorithm assumes that all weights are nonnegative, so the adjacency matrix entries are either nonnegative values or "infinity" to represent missing arcs. The algorithm maintains a set IN of nodes whose shortest distance from source node x, using only nodes already in IN, is known. For every node z outside IN, we keep track of the shortest distance d[z] from x to that node, using a path whose only non-IN node is z. We also keep track of the node adjacent to z on this path, s[z]. Initially, IN = {x}, and the distance of x from x is, of course, 0.

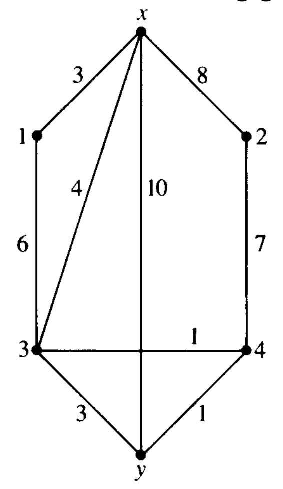
How do we let IN grow; that is, which node should be moved into IN next? Because we are greedy, we pick the non-IN node with the smallest distance d[1]. Once we add that node, call it p, to IN, then we have to recompute d for all the remaining non-IN nodes, because there may be a shorter path from x going through p than there was before p belonged to IN. If there is a shorter path, we must also update s[z] so that p is now shown to be the node adjacent to z on the current shortest path. When all nodes are in IN, the distance array *d* supposedly contains the shortest distance from x for each node using only nodes in IN, but that's all the nodes of the graph, so it's the overall shortest distance. The value *d*[*y*] is the distance for the shortest path from x to node y, and its nodes are found by looking at y, s[y], s[s[y]], and so forth, until we have traced the path back to x.

A detailed pseudocode form of the algorithm is given below. The input is the adjacency matrix for a connected graph G with positive weights and source node x; the algorithm writes out the shortest path between x and any node and the distance for that path.

[1] Note: we don't yet know that this greedy process will result in the overall final best outcome.

```
Dijkstra's Algorithm (n × n matrix A; source node x)

Local variables:
set of nodes IN       //set of nodes whose shortest path from x is known
nodes z, p, y         //temporary nodes
array of integers d   //for each node, the distance from x using nodes in IN
array of nodes s      //for each node, the previous node in the shortest path
integer OldDistance   // distance to compare against

   //initialize set IN and arrays d and s
   IN = {x}
   d[x]  = 0
   for all nodes z not in IN do
      d[z] = A[x, z] //arc from x to z
      s[z] = x
   end for

   //process nodes into IN
   while nodes outside IN remain do
      //add minimum-distance node not in IN
      p  = node z not in IN with minimum d[z]
      IN = IN  ∪ {p}

      //recompute d for non-IN nodes, adjust s if necessary
      for all nodes z not in IN  do
         OldDistance = d[z]
         d[z] = min(d[z], d[p] + A[p, z]) //may be shorter path through p
         if d[z] ≠ OldDistance then
            s[z] = p
         end if
      end for
   end while

   //get user's choice of destination node
   write("Destination node?")
   read y
   write("In reverse order, the path is")
   write (y)
   z = y
   repeat
      write (s[z])
      z = s[z]
   until z = x
   write("The path distance is", d[y])
```

EXAMPLE 1 Consider the following graph and its adjacency matrix.



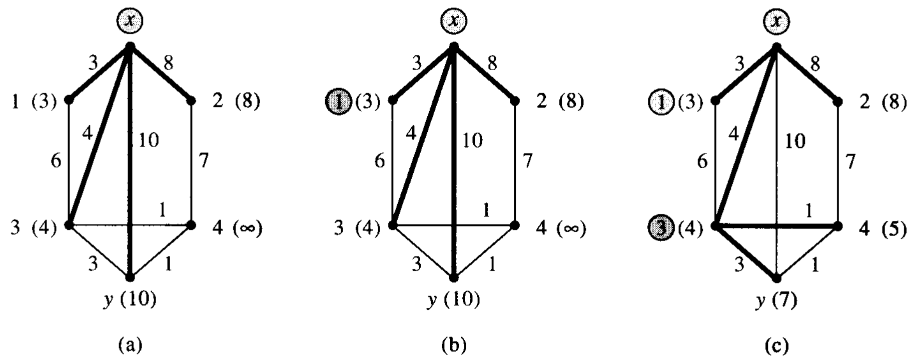
$$\begin{array}{c|cccccc} & x & 1 & 2 & 3 & 4 & y \\ \hline x & \infty & 3 & 8 & 4 & \infty & 10 \\ 1 & 3 & \infty & \infty & 6 & \infty & \infty \\ 2 & 8 & \infty & \infty & \infty & 7 & \infty \\ 3 & 4 & 6 & \infty & \infty & 1 & 3 \\ 4 & \infty & \infty & 7 & 1 & \infty & 1 \\ y & 10 & \infty & \infty & 3 & 1 & \infty \end{array}$$

At the end of the initialization phase, *IN* contains only x, d contains all the direct distances from x to other nodes, and the s value for all nodes is x because x is the "predecessor" in each case.

IN = {x}

	x	1	2	3	4	*y*
d	0	3	8	4	∞	10
s	–	*x*	*x*	*x*	*x*	*x*

In Figure 1, circled nodes are those in set *IN*, heavy lines show the current shortest paths, and the d-value for each node is written along with the node label. Figure 1a is the picture after initialization.



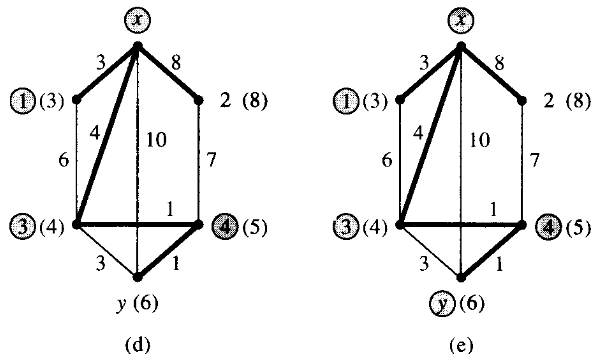


Figure 1

We now enter the **while** loop and search through the d-values for the node of minimum distance that is not in *IN*; this turns out to be node 1, with $d[1] = 3$. We throw node 1 into *IN*, and in the **for** loop we recompute all the *d*-values for the remaining nodes, 2, 3, 4, and y.

$p = 1$
$IN = \{x, 1\}$
$d[2] = \min(8, 3 + \mathbf{A}[1, 2]) = \min(8, \infty) = 8$
$d[3] = \min(4, 3 + \mathbf{A}[1, 3]) = \min(4, 9) = 4$
$d[4] = \min(\infty, 3 + \mathbf{A}[1, 4]) = \min(\infty, \infty) = \infty$
$d[y] = \min(10, 3 + \mathbf{A}[1, y]) = \min(10, \infty) = 10$

There were no changes in the *d*-values, so there were no changes in the *s*-values (there were no shorter paths from *x* by going through node 1 than by going directly from *x*). Figure 1b shows that 1 is now in *IN*.

The second pass through the **while** loop produces the following:

$p = 3$ (3 has the smallest d-value, namely 4, of 2, 3, 4, or y)
$IN = \{x, 1, 3\}$
$d[2] = \min(8, 4 + A[3, 2]) = \min(8, 4 + \infty) = 8$
$d[4] = \min(\infty, 4 + A[3, 4]) = \min(\infty, 4 + 1) = 5$ (a change, so update $s[4]$ to 3)
$d[y] = \min(10, 4 + A[3, y]) = \min(10, 4 + 3) = 7$ (a change, so update $s[y]$ to 3)

	x	1	2	3	4	*y*
d	0	3	8	4	5	7
s	–	*x*	*x*	*x*	3	3

Shorter paths from *x* to the two nodes 4 and *y* were found by going through 3. Figure 1c reflects this.

On the next pass,

$p = 4$ (d-value = 5)
$IN = \{x, 1, 3, 4\}$
$d[2] = \min(8, 5 + 7) = 8$
$d[y] = \min(7, 5 + 1) = 6$ (a change, update $s[y]$)

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	–	x	x	x	3	4

See Figure 1d.

Processing the **while** loop again, we get

$p = y$
$IN = \{x, 1, 3, 4, y\}$
$d[2] = \min(8, 6 + \infty) = 8$

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	–	x	x	x	3	4

See Figure 1e. Finally node 2 is brought into IN and the **while** loop terminates.

If y is the destination node, the distance for the shortest path from x to y is $d[y] = 6$. The path goes through y, $s[y] = 4$, $s[4] = 3$, and $s[3] = x$. Thus the path uses nodes x, 3, 4, and y. (The algorithm gives us these nodes in reverse order.). By looking at the graph and checking all the possibilities, we can see that this is the shortest path from x to y.

End of Example 1

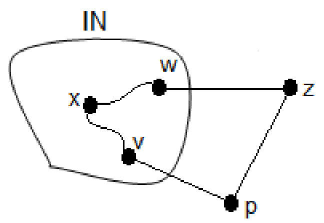
Correctness: Now, what about the correctness of this algorithm, the issue raised by the footnote earlier? Once we bring the next node p into the set IN, d[p] is never changed again. How do we know that adding some node z to IN later doesn't produce a shorter path for p? In other words, how do we prove that this algorithm has the greedy-choice property?

What we want to prove is that for every node y in IN, d[y] is actually the shortest path to y from x. We can prove this by induction on the size of IN.

Base case: |IN| = 1, that is, IN = {x}. We set d[x] = 0, which is clearly the length of the shortest path from x to x.

Assume the hypothesis is true for |IN| = k

Consider the situation when node p is added to IN to make |IN| = k + 1, with v being the node in IN that is s[p] . We can picture the situation as shown here:



Then d[p] = |x-v-p|, and we need to prove that x-v-p is the shortest path from x to p. Suppose that is not true, and that the path x-w-z-p is a shorter path. We know that |x-w-z| ≥ |x-v-p| because p, not z, was chosen as the next node to be brought into IN. Because the weight of the z-p arc is nonnegative, |x-w-z-p| ≥ |x-w-z|, so that |x-w-z-p| ≥ |x-v-p| and therefore x-w-z-p is not a shorter path from x to p.

Analysis: How efficient is the shortest-path algorithm? Most of the work seems to take place within the **for** loop that modifies the *d* and *s* arrays. Here the algorithm checks all n nodes to determine which nodes *z* are not in *IN* and recomputes *d*[*z*] for those nodes, possibly also changing *s*[*z*]. The necessary quantities *d*[*z*], *d*[p], and *A*[*p*, *z*] for a given *z* are directly available. Therefore the **for** loop requires $\Theta(n)$ operations. In addition, determining the node *p* to add to *IN* can also be done in $\Theta(n)$ operations by checking all n nodes. With the additional small amount of work to add p to *IN*, each execution of the **while** loop takes $\Theta(n)$ operations. The **while** loop will be executed n - 1 times. Therefore the total number of operations involved in the **while** loop is $\Theta(n(n - 1)) = \Theta(n^2)$. Initialization and writing the output together take $\Theta(n)$ operations, so the algorithm requires $\Theta(n + n^2) = \Theta(n^2)$ operations in the worst case.

What if we keep *IN* (or rather the complement of *IN*) as some sort of linked list, so that all the nodes of the graph do not have to be examined to see which are not in *IN*? Surely this would make the algorithm more efficient. Note that the number of nodes not in *IN* is initially n - 1, and that number decreases by 1 for each pass through the **while** loop. Within the **while** loop the algorithm thus has to perform on the order of n - 1 operations on the first pass, then n - 2, then n - 3, and so on. But, as proof by induction will show,

$$(n - 1) + (n - 2) + \ldots + 1 = (n - 1)n / 2 = \Theta(n^2)$$

Thus the algorithm still requires $\Theta(n^2)$ operations.

Dijkstra's algorithm also works for directed graphs, assuming the adjacency matrix reflects the direction of the arc.

3. Huffman Encoding[2]

Problem and Trial Solution: Character data consist of letters of the alphabet (both uppercase and lowercase), punctuation symbols, the blank space, and other keyboard symbols such as @ and %. Computers store character data in binary form, as a sequence of 0s and 1s. The usual approach is to fix some length n so that 2^n is as large as the number of distinct characters and to encode each distinct character as a particular sequence of n bits. Each character must be encoded into its fixed binary sequence, and then the binary sequence must be decoded when the character is to be displayed. The most common encoding scheme is ASCII (American Standard Code for Information Interchange), which uses n = 8, so that each character requires 8 bits to store. A version of the Unicode encoding scheme uses n = 16, so that each character requires 16 bits to store. But whatever value is chosen for n, each character requires the same amount of storage space.

Suppose a collection of character data to be stored in a file in binary form is large enough that the amount of storage required is a consideration. Suppose also that the file is archival in nature, and its contents will not often be changed. Then it may be worthwhile to invest some extra effort in the encoding process if the amount of storage space required for the file could be reduced.

Rather than using a fixed number of bits per character, an encoding scheme could use a variable number of bits and store frequently occurring characters as sequences with fewer bits. In order to store all the distinct characters, some sequences will still have to be long, but if the longer sequences are used for characters that occur less frequently, the overall storage required should be reduced. This approach requires knowledge of the particular file contents, which is why it is best suited for a file whose contents will not be frequently changed. We will study such a **data compression** or **data compaction** scheme next.

EXAMPLE 2 As a trivial example, suppose that a collection of data contains 50,000 instances of the six characters a, c, g, k, p, and ?, which occur with the following percent frequencies:

Character	a	c	g	k	p	?
Frequency	48	9	12	4	17	10

Because six distinct characters must be stored, the fixed-length scheme would require at a minimum three bits for each character ($2^3 = 8 \geq 6$). The total storage required would then be 50,000 * 3 = 150,000 bits. Suppose instead that the following encoding scheme is used:

Character	a	c	g	k	p	?
Encoding scheme	0	1101	101	1100	111	100

Then the storage requirement (number of bits) is

$$50{,}000(0.48 * 1 + 0.09 * 4 + 0.12 * 3 + 0.04 * 4 + 0.17 * 3 + 0.10 * 3) = 108{,}500$$

[2] Much of the material in this section comes from Mathematical Structures for Computer Science, 7th Edition, Judith L. Gersting, copyright W. H. Freeman and Company, 2014.

which is roughly two-thirds of the previous requirement.

End of Example 2

In the fixed-length storage scheme with n bits for each character, the long string of bits within the encoded file can be broken up into the code for successive characters by simply looking at n bits at a time. This makes it easy to decode the file. In the variable-length code, there must be a way to tell when the sequence for one character ends and the sequence for another character begins.

EXAMPLE 3 Using the variable-length code of Example 2, each of the following strings can be decoded in only one way.

a. 11111111010100:	ppca?
b. 1101010101100:	cagak
c. 100110001101100;	?kac?

As each new digit is considered, the possibilities are narrowed as to which character is being represented until the character is uniquely identified by the end of that character's representation. There is never any need to guess at what the character might be and then backtrack if our guess proves wrong. This ability to decode uniquely without false starts and backtracking comes about because the code is an example of a **prefix code**. In a prefix code, the code for any character is never the prefix of the code for any other character. (A prefix code is therefore an "antiprefix" code!)

Consider the code

Character	a	b	c
Encoding scheme	01	101	011

which is not a prefix code. Given the string 01101, it could represent either ab (01-101) or ca (011-01). Furthermore, in processing the string 011011 digit by digit as a computer would do, the decoding could begin with ab (01-101) and only encounter a mismatch at the last digit. Then the process would have to go all the way back to the first digit in order to recognize cc (011 - 011).
End of Example 3

As an aside, Morse code is a variable length code. Morse's encoding scheme for telegraphic communication, invented in 1838, uses strings of dots and dashes to represent letters of the alphabet. The most frequently-occurring letter of the alphabet in English text is the letter "e", which is assigned the shortest code, a single dot. The Morse code for "hello world"

is

".... . .-.. .-.. --- .-- --- .-. .-.. -.."

Here you can see that "e" is a single dot, that "l" is dot-dash-dot-dot, and that "r" is dot-dash-dot. Morse code is not a prefix code, however; note that the code for "r" is the first part of the code for "l". In order to avoid the ambiguity of the above example, Morse code inserts a pause between the code for each letter. To decode, you wait for the pause and at that point you have the code for exactly one letter.

In our approach to prefix codes, we will build binary trees with the characters as leaves. Once the tree is built, a binary code can be assigned to each character by simply tracing the path from the root to that leaf, using 0 for a left branch and 1 for a right branch. Because no leaf precedes any other leaf on some path from the root, the code will be a prefix code. The binary tree for the code of Example 2 is shown in Figure 2 below.

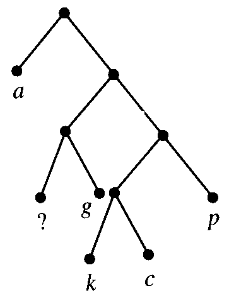


Figure 2

Suppose a code tree T exists, with leaves representing characters. For any leaf i, its depth d(i) in T equals the number of bits in the code for the corresponding character. Let f(i) denote the percentage frequency of that character in the data to be stored, and let S be the total number of characters to be stored. Then, just as in Example 2, the total bits required is given by the expression

$$S * \left[\sum_{\text{all leaves } i} (d(i)f(i)) \right]$$

We seek to build an optimal tree T, one for which the expression

$$E(T) = \sum_{\text{all leaves } i} (d(i)f(i))$$

is a minimum and hence the file size is a minimum.

This process could be done by trial and error, because there is only a finite number of characters and thus only a finite number of ways to construct a tree and assign characters to its leaves. However, the finite number quickly becomes very large! Instead we will use the algorithm known as **Huffman encoding**

Huffman Encoding Algorithm:

Suppose, then, that we have m characters in a file and we know the percentage frequency of each character. The algorithm to build the tree works by maintaining a list L of nodes that are roots of binary trees. Initially L will contain m roots, each labeled with the frequency of one of the characters; the roots will be ordered according to increasing frequency, and each will have no children.

A pseudocode description of the algorithm follows.

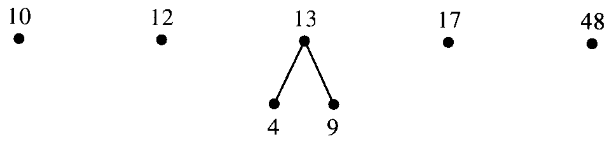
```
for (i = 1 to m - 1) do
   create new node z
   let x, y be the first two nodes in L   //minimum frequency nodes
   f(z)  = f(x) + f(y)
   insert z in order into L
   left child of z  = node x
   right child of z = node y       //x and y are no longer in L
end for
```

When this algorithm terminates, L consists of just one node, which is the root of the final binary tree. Codes can then be assigned to each leaf of the tree by tracing the path from the root to the leaf and accumulating 0s for left branches and 1s for right branches. By the way the tree is constructed, every internal node will have exactly two children.

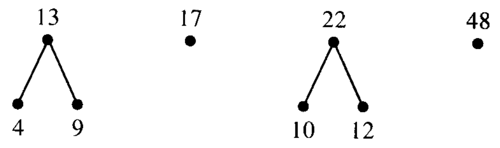
EXAMPLE 4 We'll use the Huffman encoding algorithm to build the tree of Figure 2, which is based on the data of Example 2. L initially contains the six nodes, ordered by percent frequency:

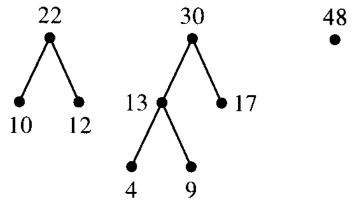
4
9
10
12
17
48

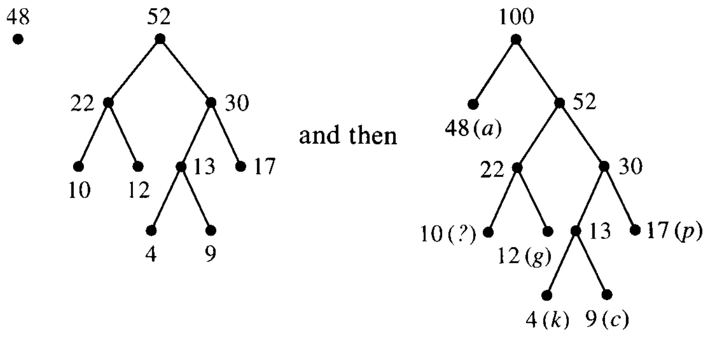
Following the algorithm, we enter the **for** loop for the first time. The x and y nodes are those with frequencies 4 and 9, respectively. A new node z with frequency 4 + 9 = 13 is created and inserted in order into L, with the x node as its left child and the y node as its right child. The new L looks like the following



This process is repeated four more times. The resulting L at each stage follows:







At this point the tree is complete and the codes can be assigned. The code for c, for example, is 1101 (right branch, right branch, left branch, right branch).

End of Example 4

In order to use Huffman encoding/decoding for data compression, you first perform a frequency analysis on the cleartext file. Using the frequency information (which works just as well as percent information), you apply the Huffman algorithm to build the binary tree and determine the code for each character; store this code table in a separate file. To encode the cleartext file, read one character at a time from the cleartext file and, using the code table, encode it in binary form as its Huffman code. This binary file is the data-compressed version of the cleartext file and presumably requires less storage space. However, the code table file must also be stored in order to be able to decode the coded file later on. To subsequently decode the coded file, read the binary coded file one bit at a time until the bit string matches a string in the code table, then write the corresponding character to a "decoded" file and begin another bit string. In the end, the "decoded" file should match the original cleartext file.

Correctness: Why is this a greedy algorithm? Our goal is to minimize the expression

$$E(T) = \sum_{\text{all leaves } i} (d(i)f(i))$$

and the value of E(T) depends at least in part on the frequencies. In Huffman's algorithm, at each step we combine the two subtrees with the lowest frequencies. Fortunately, with this strategy the overall lowest frequencies end up toward the bottom of the final tree, where the depths are greatest and, likewise, the highest frequencies end up toward the top of the final tree where the depths are smallest. So it seems that this greedy approach will indeed give an optimal solution, but we can justify this with a formal proof.

We want to show that the binary tree built by Huffman's algorithm gives the minimum possible value for E(T).

First, if we have an optimal tree T for m characters, the nodes with the lowest frequencies can always be assumed to be siblings at the lowest level of the tree. To prove this, label the two nodes with the lowest frequencies x and y. Find two siblings p and q at the lowest level of the tree, and assume that x and y are not at that level (see Figure 3a). Because f(x) is one of the two smallest values, we know that f(x) ≤ f(p). If f(x) < f(p), then interchanging x and p in the tree would result in a new tree T' with E(T') < E(T) (Figure 3b: the larger frequency is now at a lesser depth), but this would contradict the fact that T was optimal. Therefore f(x) = f(p), and x and p can be interchanged in the tree with no effect on E(T). Similarly, y and q can be interchanged, resulting in Figure 3c, in which x and y are siblings at the lowest level. If x or y are at the same level as p and q to begin with, they can certainly be interchanged with p or q without affecting E(T) (Figure 3d).

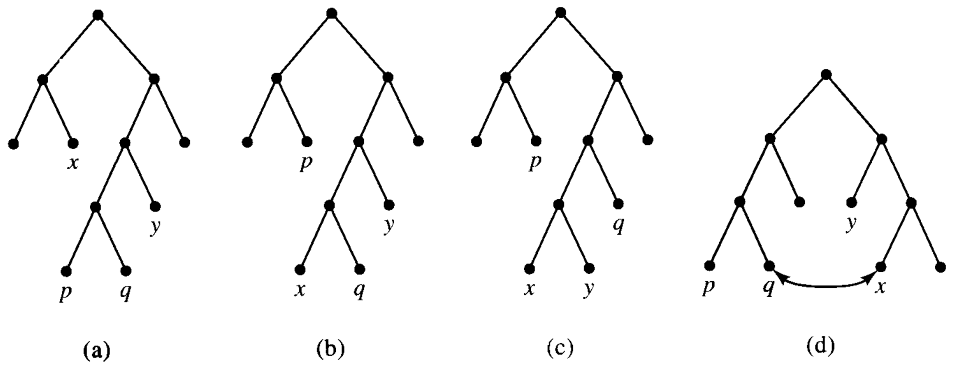


Figure 3

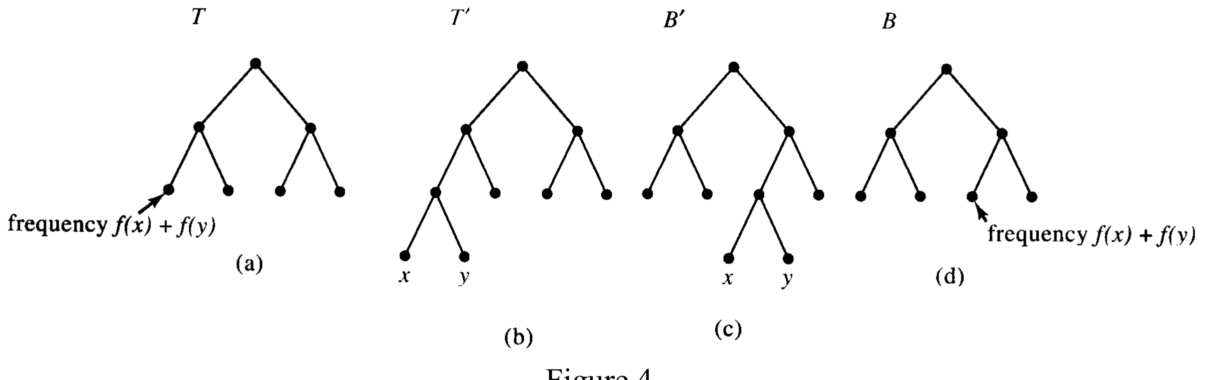


Figure 4

Now again let f(x) and f(y) be the minimum frequencies, and suppose we have a tree T that is optimal for the other frequencies together with the sum f(x) + f(y) (Figure 4a). This sum will be the frequency of a leaf node; create a tree T' that has this node as an interior node with children x and y having frequencies f(x) and f(y) (Figure 4b). T' will be optimal for frequencies f(x), f(y), and the rest. The proof of this fact begins with some optimal tree B' for frequencies f(x), f(y), and the rest. We know such an optimal tree exists (since it could be found by trial and error), and from the preceding paragraph, we can assume that x and y are siblings at the lowest level of B' (Figure 4c). Now create a tree B by stripping nodes x and y from B' and giving frequency f(x) + f(y) to their parent node, now a leaf (Figure 4d). Because T is optimal for the other frequencies together with f(x) + f(y), we have

$$E(T) \leq E(B) \tag{1}$$

But the difference between E(B) and E(B') is one arc each for x and y; that is, E(B') = E(B) + f(x) + f(y). Similarly, we have E(T') = E(T) + f(x) + f(y). Thus, if we add f(x) + f(y) to both sides of (1), we get

$$E(T') \leq E(B') \tag{2}$$

Because B' was optimal, it cannot be the case that E(T') < E(B'), so E(T') = E(B'), and T' is optimal.

Finally, a tree with a single node whose frequency is the sum of all the frequencies is trivially optimal for that sum. We can repeatedly split up this sum and drop down children in such a way that we end up with the Huffman tree. By the previous paragraph, each such tree, including the final Huffman tree, is optimal.

An Application of Huffman Codes: JPEG is a standardized image compression mechanism for photographic quality images. JPEG stands for Joint Photographic Experts Group, the name of the group that developed this international standard. The need for improved image compression was largely fueled by the desire to transmit images over the Internet. There are actually two versions of JPEG encoding, lossy and lossless, but the lossy version is by far the more common. A lossy compression scheme means that once the compressed data has been unencoded, it does not precisely match the original – some information has been "lost". In the case of lossy JPEG compression, this data loss comes from the preprocessing of the image before Huffman encoding is applied; the Huffman encoding/decoding faithfully restores the data it starts with.

JPEG compression is intended for images to be viewed by humans, and it takes advantage of the fact that the human eye is much more sensitive to gradients of light and dark than it is to small changes in color. The first step in the JPEG process is therefore to take the color image information, which is usually given as 24 bits per pixel, 8 bits for each of the red, green, and blue components, and transform each pixel into components that capture the luminance (lightness/darkness) with reduced information about the color components. Next, pixels with similar color information are grouped together and an "average" color value is used, while more accurate luminance data are maintained. The data are then transformed into frequency data (that is, the data are represented as a combination of cosine waves of varying frequencies), which in turn go through a "quantization" process (basically rounding the results of a computation) to end up in integer form. Higher-frequency variations (to which the human eye is less sensitive) are lost in this process, but again the luminance data is treated at a finer grain than the color data. Huffman encoding is applied to the result. Areas of the image whose representations occur frequently will encode to smaller bit strings.

A JPEG image file contains not only the compressed data but also the information needed to reverse the compression process (including the information to reverse the Huffman encoding). The resulting image will have lost the high-frequency changes and color variations that were eliminated in the stages before the Huffman coding was applied. Parameters in the JPEG encoding process allow tradeoffs to be made between the amount of compression to be obtained and the faithfulness of the restored image to the original. Because of the nature of the algorithms used, JPEG encoding has little or no effect on black and white line drawings where there is no data to throw away.