Chapter 10 The Greedy Method

Civil War Knapsack. U.S. government image. Vicksburg National Military Park. Public domain.

Contents

Suppose you are designing a new online auction website that is intended to process bids for ***multi-lot*** auctions. That is, this website should be able to handle a single auction for 100 units of the same digital camera or 500 units of the same smartphone, where bids could be of the form, "x units for \$$y$," meaning that the bidder wants a quantity of x of the items being sold and is willing to pay \$$y$ for all x of them. The challenge for your website is that it must allow for a large number of bidders to place such multi-lot bids and it must decide which bidders to choose as the winners.

Naturally, in order to maximize sales commissions, the website managers are interested in you designing the website so that it always chooses a set of winning bids that maximizes the total amount of money paid for the items being auctioned. So how do you decide which bidders to choose as the winners?

One way is to use the technique we discuss in this chapter—the ***greedy method***. This algorithm design paradigm involves repeatedly making choices that optimize some objective function, like repeatedly accepting the bid that maximizes the price-per-unit. The trick in applying this technique is guaranteeing that such a local "greedy" choice can always lead to an optimal solution.

Proving that the greedy method can indeed lead to an optimal solution can often require deeper understanding of the problem being solved, though, so we may need to make additional assumptions in order for it to work. For instance, for the problem of deciding which bidders to accept, this greedy strategy can work only if you are allowed to partially satisfy bids. That is, the greedy strategy works if you can satisfy a bid to buy x units for \$$y$ by selling $k < x$ units for \$$yk/x$. In fact, this problem is equivalent to a problem we study in more detail in this chapter—the fractional knapsack problem.

In the ***knapsack*** problem, we are given a set of n items, each having a weight and a benefit, and we are interested in choosing the set of items that maximize our total benefit while not going over the weight capacity of the knapsack. So, in this case, each bid is an item, with its "weight" being the number of units being requested and its benefit being the amount of money being offered. In this particular instance, where bids can be satisfied with a partial fulfillment, then it is an instance of the ***fractional knapsack*** problem, for which the greedy method works to find an optimal solution. Interestingly, for the "0-1" version of the problem, where fractional choices are not allowed, then the greedy method may not work. In fact, solving this problem for all possible inputs is quite difficult—it is discussed in Section 17.5 in the context of ***NP***-completeness.

Still, there are several other problems, which we discuss in this chapter, for which the greedy methods works to find an optimal solution. These include problems for task scheduling and text compression. Incidentally, this technique is also used in Chapter 14, to derive efficient algorithms for finding shortest paths in weighted graphs, Chapter 15, to construct minimum spanning trees, and Section 18.2.2, to find approximate solutions to the SET-COVER problem.

The Greedy Method

The ***greedy method*** is applied to optimization problems—that is, problems that involve searching through a set of ***configurations*** to find one that minimizes or maximizes an ***objective function*** defined on these configurations. The general formula of the greedy method could not be simpler—in order to solve a given optimization problem, we proceed by a sequence of choices. The sequence of choices starts from some well-understood starting configuration, and then iteratively makes the decision that is best from all of those that are currently possible, in terms of improving the objective function. (See Figure 10.1.)

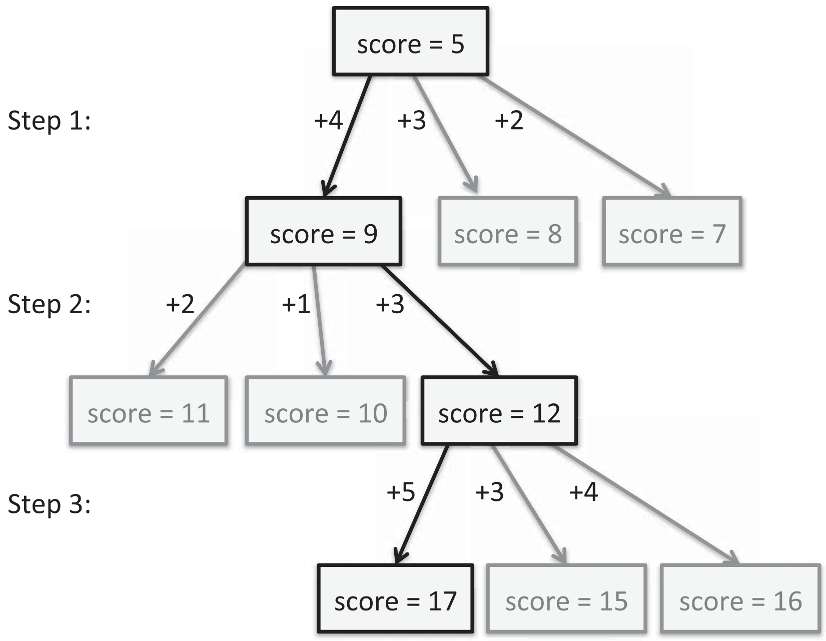

Figure 10.1: An example of the greedy method. Each rectangle represents a configuration whose score is the objective function. A configuration has three choices, each of which provides a different incremental improvement to the score. The bold choices are the ones that are picked in each step according to the greedy strategy.

This greedy approach does not always lead to an optimal solution, but there are several problems that it does work optimally for, and such problems are said to possess the ***greedy-choice*** property. This is the property that a global optimal configuration can be reached by a series of locally optimal choices (that is, choices that are the best from among the possibilities available at the time), starting from a well-defined configuration.

10.1 The Fractional Knapsack Problem

Consider the ***fractional knapsack*** problem, where we are given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, and we wish to find the maximum-benefit subset that does not exceed a given weight W. If we are restricted to entirely accepting or rejecting each item, then we would have the 0-1 version of this problem (for which we give a dynamic programming solution in Section 12.6). Let us now allow ourselves to take arbitrary fractions of some elements, however. The motivation for this fractional knapsack problem is that we are going on a trip and we have a single knapsack that can carry items that together have weight at most W. In addition, we are allowed to break items into fractions arbitrarily. That is, we can take an amount x_i of each item i such that

$$0 \leq x_i \leq w_i \text{ for each } i \in S \quad \text{and} \quad \sum_{i \in S} x_i \leq W.$$

The total benefit of the items taken is determined by the objective function

$$\sum_{i \in S} b_i(x_i/w_i).$$

(See Figure 10.2.)

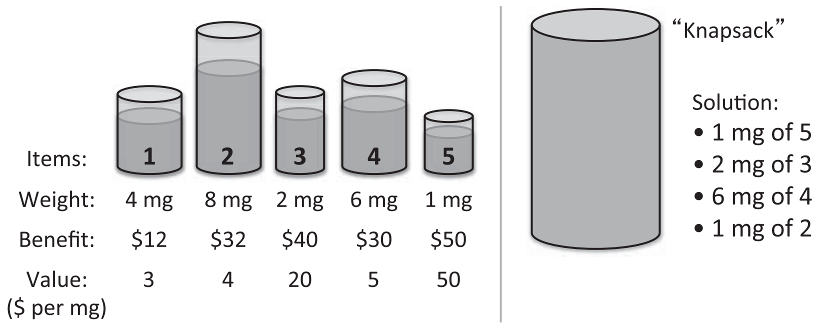

Figure 10.2: The fractional knapsack problem.

Consider, for example, a student who is going to an outdoor sporting event and must fill a knapsack full of foodstuffs to take along. As long as each candidate foodstuff is something that can be easily divided into fractions, such as drinks, potato chips, popcorn, and pizza, then this would be an instance of the fractional knapsack problem.

Using the Greedy Method to Solve the Fractional Knapsack Problem

When considering the fractional knapsack problem, this is one place where the greedy method can be applied successfully, for we can solve the fractional knapsack problem using the greedy approach.

In applying the greedy method, the most important decision is to determine the objective function that we wish to optimize. For example, we could choose to include items into our knapsack based on their weights, say, taking items in order by increasing weights. The intuition behind this approach is that the lower-weight items consume the least amount of the weight resource of the knapsack. Unfortunately, this first idea doesn't work. It is easy to construct examples where choosing items in order by their weights leads to suboptimal solutions. For example, even with just two items, with weight-benefit pairs of $(1, 10)$ and $(10, 200)$, this weight-based greedy approach, with a knapsack of weight 10, chooses 1 unit of item 1 and 9 units of item 2, for a total benefit of $10 + 180 = 190$, whereas just taking item 2 would get a benefit of 200.

Another possibility is to choose items in order based on their benefits, but again it is easy to construct examples where this results in a suboptimal strategy (see Exercise C-10.1). A much better approach is to rank the items by their ***value***, which we define to be the ratio of their benefits and weights. The intuition behind this is that benefit-per-weight-unit is a natural measurement of the inherent value that each item possesses. Indeed, this approach leads to an efficient algorithm that finds an optimal solution to the fractional knapsack problem. We describe the details of this approach in Algorithm 10.3.

```
Algorithm FractionalKnapsack($S, W$):
   Input: Set $S$ of items, such that each item $i \in S$ has a positive benefit $b_i$ and a
      positive weight $w_i$; positive maximum total weight $W$
   Output: Amount $x_i$ of each item $i \in S$ that maximizes the total benefit while
      not exceeding the maximum total weight $W$
  for each item $i \in S$ do
      $x_i \leftarrow 0$
      $v_i \leftarrow b_i/w_i$        // value index of item $i$
  $w \leftarrow 0$        // total weight
  while $w < W$ and $S \neq \emptyset$ do
      remove from $S$ an item $i$ with highest value index        // greedy choice
      $a \leftarrow \min\{w_i, W - w\}$        // more than $W - w$ causes a weight overflow
      $x_i \leftarrow a$
      $w \leftarrow w + a$
```

Algorithm 10.3: A greedy algorithm for the fractional knapsack problem.

Analyzing the Greedy Algorithm for the Fractional Knapsack Problem

The above FractionalKnapsack algorithm can be implemented in $O(n \log n)$ time, where n is the number of items in S. Specifically, we can use a heap-based priority queue (Section 5.3) to store the items of S, where the key of each item is its value index. With this data structure, each greedy choice, which removes the item with greatest value index, takes $O(\log n)$ time.

Alternatively, we could even sort the items by their benefit-to-weight values, and then process them in this order. This would require $O(n \log n)$ time to sort the items and then $O(n)$ time to process them in the while-loop of Algorithm 10.3. Either way, we have that the greedy algorithm for solving the fractional knapsack problem can be implemented in $O(n \log n)$ time. The following theorem summarizes this fact and also shows that this algorithm is correct.

Theorem 10.1: *Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time.*

Proof: To see that our algorithm (10.3) for solving the fractional knapsack problem is correct, suppose, for the sake of contradiction, that there is an optimal solution better than the one chosen by this greedy algorithm. Then there must be two items i and j such that

$$x_i < w_i, \quad x_j > 0, \quad \text{and} \quad v_i > v_j.$$

Let

$$y = \min\{w_i - x_i, x_j\}.$$

But then we could then replace an amount y of item j with an equal amount of item i, thus increasing the total benefit without changing the total weight, which contradicts the assumption that this non-greedy solution is optimal. Therefore, we can correctly compute optimal amounts for the items by greedily choosing items by increasing benefit-to-weight values. ■

The proof of this theorem uses an ***exchange argument*** to show that the greedy method works to solve this problem optimally. The general structure of such an argument is a proof by contradiction, where we assume, for the sake of reaching a contradiction, that there is a better solution than one found by the greedy algorithm. We then argue that there is an exchange that we could make among the components of this solution that would lead to a better solution. In this case, this approach shows that the greedy method can effectively be used to solve the fractional knapsack problem. Incidentally, the all-or-nothing, or “0-1” version of the knapsack problem does not have an efficient greedy solution, however, and solving this version of the problem is much harder, as we explore in Sections 12.6 and 17.5.

10.2 Task Scheduling

Suppose we are given a set T of n ***tasks*** such that each task i has a ***start time***, s_i, and a ***finish time***, f_i (where $s_i < f_i$). Task i must start at time s_i and must finish by time f_i. Each task has to be performed on a ***machine*** and each machine can execute only one task at a time. Two tasks i and j are said to be ***nonconflicting*** if they do not overlap in time, i.e., $f_i \leq s_j$ or $f_j \leq s_i$. Clearly, two tasks can be scheduled to be executed on the same machine only if they are nonconflicting.

The ***task scheduling*** problem we consider here is to schedule all the tasks in T on the fewest machines possible in a nonconflicting way. Alternatively, we can think of the tasks as meetings that we must schedule in as few conference rooms as possible. (See Figure 10.4.)

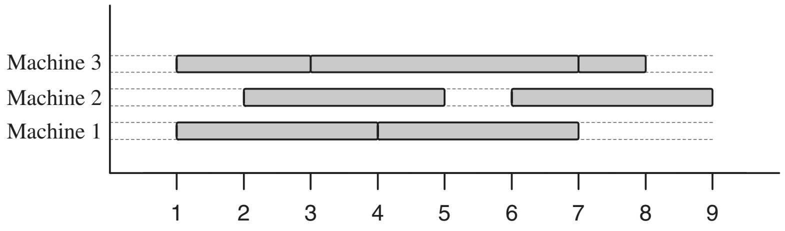

Figure 10.4: An illustration of a solution to the task scheduling problem, for tasks whose collection of pairs of start times and finish times is $\{(1,3),\ (1,4),\ (2,5),\ (3,7),\ (4,7),\ (6,9),\ (7,8)\}$.

There are several ways we might wish to solve this problem using the greedy method. As with any greedy strategy, the challenge is to find the right objective function. For example, we might consider the tasks from longest to shortest, assigning them to machines based on the first one available, since the longest tasks seem like they would be the hardest to schedule. Unfortunately, this approach does not necessarily result in an optimal solution, as is shown in Figure 10.5.

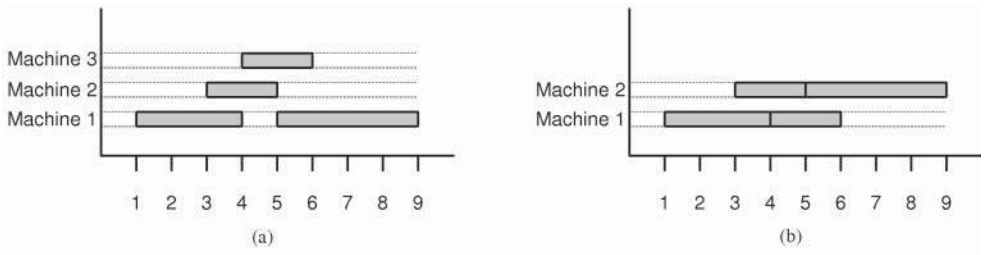

Figure 10.5: Why the longest-first strategy doesn't work, for the pairs of start and finish times in the set $\{(1,4),(5,9),(3,5),(4,6)\}$; (a) the solution chosen by the longest-first strategy; (b) the optimal solution. Note that the longest-first strategy uses three machines, whereas the optimal strategy uses only two.

A Better Greedy Approach to Task Scheduling

Another greedy approach is to consider the tasks ordered by increasing start times. In this case, we would consider each task by the order of its start time, and assign it to the first machine that is available at that time. If there are no available machines, however, then we would need to allocate a new machine and schedule this task on that machine. The intuition behind this approach is that, by processing tasks by their start times, when we process a task for a given start time we will have already processed all the other tasks that would conflict with this starting time. We give the details of this greedy algorithm in Algorithm 10.6.

```
Algorithm TaskSchedule($T$):
  Input: A set $T$ of tasks, such that each task has a start time $s_i$ and a finish
    time $f_i$
  Output: A nonconflicting schedule of the tasks in $T$ using a minimum number
    of machines
  $m \leftarrow 0$        // optimal number of machines
  while $T \neq \emptyset$ do
      remove from $T$ the task $i$ with smallest start time $s_i$
      if there is a machine $j$ with no task conflicting with task $i$ then
          schedule task $i$ on machine $j$
      else
          $m \leftarrow m + 1$        // add a new machine
          schedule task $i$ on machine $m$
```

Algorithm 10.6: A greedy algorithm for the task scheduling problem.

We show an example solution produced by this algorithm in Figure 10.7, using the same set of tasks used in Figure 10.4. Note that even though this is an optimal solution, it is not the same as the optimal solution shown in Figure 10.4.

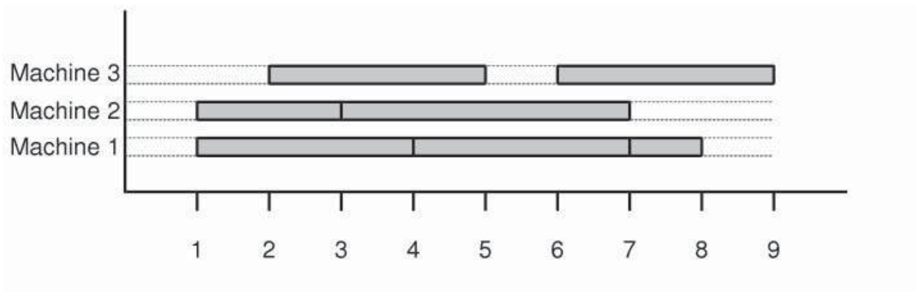

Figure 10.7: An example solution produced by the greedy algorithm based on considering tasks by increasing start times.

Analysis of the Greedy Task Scheduling Algorithm

So, in words, in the above algorithm, TaskSchedule, we begin with no machines and we consider the tasks in a greedy fashion, ordered by their start times. For each task i, if we have a machine that can handle task i, then we schedule i on that machine, say, choosing the first such available machine. Otherwise, we allocate a new machine, schedule i on it, and repeat this greedy selection process until we have considered all the tasks in T.

The following theorem states that greedy method TaskSchedule (Algorithm 10.6) produces an optimal solution, because we are always considering a task starting at a given time after we have already processed all the tasks that might conflict with this start time.

Theorem 10.2: *Given a set of n tasks specified by their start and finish times, the Algorithm* TaskSchedule *produces, in $O(n \log n)$ time, a schedule for the tasks using a minimum number of machines.*

Proof: Let k be the last machine allocated by algorithm TaskSchedule, and let i be the first task scheduled on k. When we scheduled i, each of the machines 1 through $k - 1$ contained tasks that conflict with i. Since they conflict with i and because we consider tasks ordered by their start times, all the tasks currently conflicting with task i must have start times less than or equal to s_i, the start time of i, and have finish times after s_i. In other words, these tasks not only conflict with task i, they all conflict with each other. But this means we have k tasks in our set T that conflict with each other, which implies it is impossible for us to schedule all the tasks in T using only $k - 1$ machines. Therefore, k is the minimum number of machines needed to schedule all the tasks in T.

We leave as a simple exercise (R-10.2) the job of showing how to implement the Algorithm TaskSchedule in $O(n \log n)$ time. ■

Note that the proof of this theorem does not use an exchange argument to prove the correctness of the greedy algorithm we used in this case. Instead, the above proof uses a ***lower-bound*** argument, which is another technique for proving greedy algorithms are correct. In using this technique, we argue that any solution to our problem will require a cost of at least some given parameter and we then show that the greedy algorithm achieves this lower bound as an upper bound. In the above proof, we used the parameter k for this purpose.

10.3 Text Compression and Huffman Coding

In this section, we consider another application of the greedy method—to ***text compression***. In this problem, we are given a string X defined over some alphabet, such as the ASCII or Unicode character sets, and we want to efficiently encode X into a small binary string Y (using only the characters 0 and 1). Text compression is useful in any situation where we are communicating over a low-bandwidth channel, such as a slow wireless or satellite connection, and we wish to minimize the time needed to transmit our text. Likewise, text compression is also useful for storing collections of large documents more efficiently, to allow a computational device with a small amount of storage to contain as many documents as possible.

Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters, with 7 bits in the ASCII system and 16 in the Unicode system. So, for example, an English document whose length is 100 million characters would require at least 7 megabits to represent in ASCII and 16 megabits to represent in Unicode. This is a waste of bits, however, since there are some characters that are hardly ever used and others, like the letters “e” and “t,” that are used so often that it is shame to be using the same number of bits for them as the seldomly used characters.

An alternative to such fixed-length encoding schemes, then, is a ***variable-length encoding scheme***, where the codes for various characters are allowed to have different lengths. Ideally, we would like the most-frequently used characters to use the fewest number of bits, and the least-frequently used characters to use the most. To encode a string X, we would then represent each character in X with its associated variable-length code word, and we concatenate all these code words in order to produce a binary representation, Y, for X.

In order to avoid ambiguities in this approach, we insist that our encoding scheme be a ***prefix code***, that is, we insist that no code word in our scheme is a prefix of any other code word in our scheme. The advantage of using such a prefix code is that decoding can be accomplished by using the greedy strategy of processing the bits of Y in order, repeatedly matching bits to the first code word they represent. Moreover, the savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

The challenge, of course, is that to get the maximum compression possible with this approach we want to guarantee that high-frequency characters are assigned to short code words and low-frequency characters are assigned to longer code words. In other words, to get the maximum compression for X based on this approach, our code words must be chosen based on the frequencies of how characters appear in X. So, let us assume that we have, for each character, c, in X, a count, $f(c)$, of the number of times c appears in the string X.

Huffman Coding

An interesting greedy approach to solving the text compression problem using this approach is ***Huffman coding***. This method produces a variable-length prefix code for X based on the construction of a proper binary tree T that represents the code. Each edge in T represents a bit in a code word, with each edge to a left child representing a "0" and each edge to a right child representing a "1." Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v.

Since characters are associated only with external nodes, and no internal node is associated with any code word, such a scheme produces a prefix code. Put another way, if we start matching bits in X based on the path that they trace out in T, then the external node that we reach will correspond to the character represented by the code word this string of bits is equal to. (See Figure 10.8.)

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

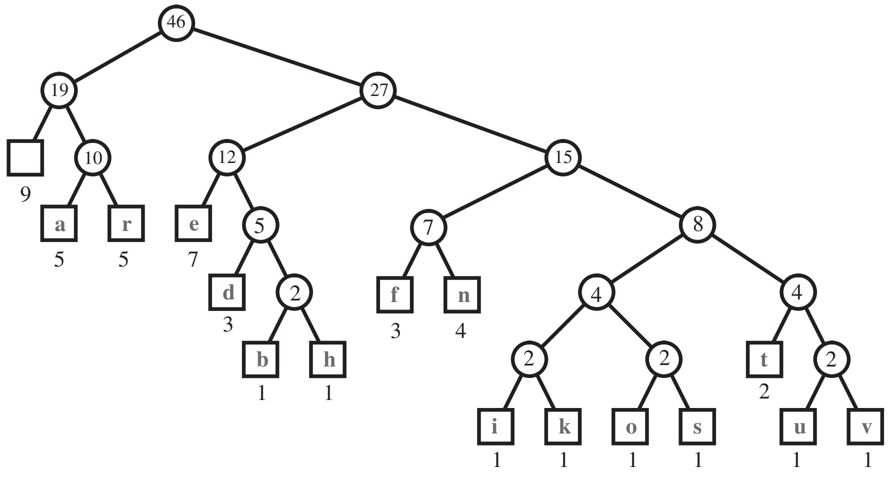

Figure 10.8: An illustration of an example Huffman code for the input string $X =$ `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

The Huffman Coding Algorithm

The critical part of the Huffman coding algorithm is to construct the tree, T, so that it represents a good prefix code. We begin with a collection, C, of characters, with each character c in C having a numeric weight, $f(c)$, which we think of as its frequency.

Each external node v in T is associated with a character and has a ***frequency***, $f(v)$, which is the frequency in X of the character associated with v. For each internal node, v, in T, we associate a total frequency, $f(v)$, which is the sum of the frequencies of all the external nodes in the subtree rooted at v. The remaining optimization problem involves choosing how T is structured so as to determine an optimal prefix code.

Let T be a binary tree, defined as above, with a numeric weight, $f(v)$, assigned to each external node, v in T, and a weight, $f(v)$, assigned to each internal node, v, that is the sum of the weights of its external-node descendants. Define the ***total path weight***, $p(T)$, of T to be the sum of all the weights in T. That is, define $p(T)$ as follows:

$$p(T) = \sum_{v \in T} f(v).$$

Note that this is equivalent to us summing, over all the external nodes of T, the product of each external node's weight and its depth. That is, given a set C of characters, where each c in C is given a (frequency) weight, $f(c)$, we can also characterize the total path weight of T as

$$p(T) = \sum_{c \in C} f(c) \cdot d(v_c),$$

where v_c is the external node associated with the character c in C and $d(v_c)$ is the depth of v_c in T.

The goal is for us to construct T so that it has minimum total path weight over all binary trees having external nodes associated with the characters in C, using the frequency of each character in C as the weight of its associated external node. (This property was true, for instance, in the tree shown in Figure 10.8b.) Thus, the problem of constructing an optimal prefix code can be reduced to the problem of constructing a binary tree with minimum total path weight.

The Huffman coding algorithm begins with a set, C, of characters that are the d distinct characters from the string X, such that each such character is associated with the root node of a single-node binary tree. The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest weight at their respective roots and merges them into a single binary tree, giving the root of the new tree the sum of the roots of the two merged trees. The algorithm then repeats this process until only one tree is left. (See Algorithm 10.9.)

```
Algorithm Huffman($C$):
   Input: A set, $C$, of $d$ characters, each with a given weight, $f(c)$
   Output: A coding tree, $T$, for $C$, with minimum total path weight
    Initialize a priority queue $Q$.
    for each character $c$ in $C$ do
        Create a single-node binary tree $T$ storing $c$.
        Insert $T$ into $Q$ with key $f(c)$.
    while $Q$.size() $> 1$ do
        $f_1 \leftarrow Q$.minKey()
        $T_1 \leftarrow Q$.removeMin()
        $f_2 \leftarrow Q$.minKey()
        $T_2 \leftarrow Q$.removeMin()
        Create a new binary tree $T$ with left subtree $T_1$ and right subtree $T_2$.
        Insert $T$ into $Q$ with key $f_1 + f_2$.
    return tree $Q$.removeMin()
```

Algorithm 10.9: Huffman coding algorithm.

Analysis of the Huffman Coding Algorithm

Each iteration of the while-loop in the Huffman coding algorithm can be implemented in $O(\log d)$ time using a priority queue represented with a heap. In addition, each iteration takes two binary trees out of Q and adds one in, all of which can be done in $O(\log d)$ time. This process is repeated $d - 1$ times before exactly one node is left in Q. Thus, this algorithm runs in $O(d \log d)$ time, assuming we are given the set, C, of d distinct characters in the string X as input. In addition, we can construct C from X in $O(n)$ time, including the calculation, for each c in C, of the frequency, $f(c)$, of how many times the character c appears in X, where n is the length of X; see Exercise C-10.3.

The correctness argument for the Huffman coding algorithm begins with the following lemma.

Lemma 10.3: *If T is a binary tree, T, with minimum total path weight for a set, C, of characters, with each c in C having a positive weight, $f(c)$, then T is proper, that is, each internal node in T has two children.*

Proof: Suppose, for the sake of contradiction, that T has an internal node, v, with only one child, w. Then we can replace v with w and T will have the same set of external nodes as before, but each node in the subtree rooted at T will have its depth reduced by 1. Since the weights for the characters in C are positive, this implies that the total path weight for the new tree is less than T, which is impossible, since T is a binary tree with minimum total path weight. ■

In addition, the following lemma is also important.

Lemma 10.4: *Given a set, C, of characters, with a positive weight, $f(c)$, defined for each c in C, two characters, b and c, with the smallest two weights, are associated with nodes that have the maximum depth and are siblings in a binary tree, T, with minimum total path weight for C.*

Proof: Let T be a binary tree with minimum total path weight for C. Suppose that the node for one of b or c does not have maximum depth in T, and let v be this node and, without loss of generality, let c be the associated character. Then there must be another external node, w, associated with a character, e, such that $f(e) \geq f(c)$ and $d(w) > d(v)$. We can swap the characters for v and w, which will cause a change in the total path weight of T by the amount

$$f(c)(d(w) - d(v)) \; - \; f(e)(d(w) - d(v)),$$

which cannot be positive. Thus, c can be associated with a maximum-depth node without increasing the total weight of T; hence, there is an optimal tree having a maximum-depth node associated with c.

To see that b and c can be siblings, note that, since we have already shown that b and c are at maximum depth in an optimal tree, T, and an optimal tree is proper, by Lemma 10.3, we can swap a sibling of either b or c to make these two be siblings without changing the total path weight of T. ■

We use this lemma to prove the following.

Theorem 10.5: *The Huffman coding algorithm constructs an optimal prefix code for a string of length n with d distinct characters in $O(n + d \log d)$ time.*

Proof: We have already established the time bound for the Huffman coding algorithm, as well as the fact that an optimal prefix code can be derived from a binary tree, T, with minimum total path weight, $p(T)$, for a set of characters and weights associated with its external nodes.

We can prove the correctness of the Huffman coding algorithm by induction on d, the number of characters in C. For the basis, note that for $d = 1$, the optimal tree is a single (root) node, and that this is the tree constructed by the Huffman coding algorithm.

Let T be a tree constructed by the Huffman coding algorithm for a set, C, of $d > 1$ characters. After merging two lowest-frequency characters, b and c, in C into a single tree having a root, r, which has the two associated nodes as its children, note that the iterative structure of the algorithm is the same as if we had started with $d - 1$ characters, including a character for r with frequency $f(r)$. Thus, we can assume inductively that the tree, T', constructed for this set of characters is optimal. In addition, since b and c are associated with the children of r,

$$p(T) = p(T') + f(b) + f(c).$$

Now, let U be an optimal tree for the set C, where, by Lemma 10.4, the nodes for b and c are siblings. Let U' be the tree derived from U by removing the nodes for b and c, but keeping their parent node, which has weight $f(b) + f(c)$. That is, U' is defined on the same set of characters and frequencies as T', and

$$p(U) = p(U') + f(b) + f(c).$$

In addition, since T' is optimal,

$$p(T') \leq p(U').$$

Therefore,

$$\begin{aligned} p(T) &= p(T') + f(b) + f(c) \\ &\leq p(U') + f(b) + f(c) \\ &= p(U). \end{aligned}$$

In other words, T has weight less than or equal to U. But, since U is an optimal tree, this means that T must be an optimal tree. ■

How the Huffman Coding Algorithm Uses the Greedy Method

The Huffman coding algorithm for building an optimal prefix code is another application of the ***greedy method***. As mentioned above, this technique is applied to optimization problems, where we are trying to construct some structure while minimizing or maximizing some property of that structure.

Indeed, the Huffman coding algorithm closely follows the general formula for the greedy method. Namely, in order to solve the given optimization code problem using the greedy method, we proceed by a sequence of choices. The sequence starts from a well-understood starting condition, and computes the cost for that initial condition. Finally, we iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible. This approach does not always lead to an optimal solution, but it does indeed find the optimal prefix code when used according to the approach of the Huffman coding algorithm.

This global optimality for the Huffman coding algorithm is due to the fact that the optimal prefix coding problem possesses the ***greedy-choice*** property. This is the property that a global optimal condition can be reached by a series of locally optimal choices (that is, choices that are each the current best from among the possibilities available at the time), starting from a well-defined starting condition. In this case, the general condition is defined by a set of disjoint binary trees, each with a given weight for its root, and the greedy choice is to combine two lowest-weight trees into a single tree.

10.4 Exercises

Reinforcement

R-10.1 Let $S = \{a, b, c, d, e, f, g\}$ be a collection of objects with benefit-weight values, a: $(12, 4)$, b: $(10, 6)$, c: $(8, 5)$, d: $(11, 7)$, e: $(14, 3)$, f: $(7, 1)$, g: $(9, 6)$. What is an optimal solution to the fractional knapsack problem for S assuming we have a sack that can hold objects with total weight 18? Show your work.

R-10.2 Describe how to implement the TaskSchedule method to run in $O(n \log n)$ time.

R-10.3 Suppose we are given a set of tasks specified by pairs of the start times and finish times as $T = \{(1,2), (1,3), (1,4), (2,5), (3,7), (4,9), (5,6), (6,8), (7,9)\}$. Solve the task scheduling problem for this set of tasks.

R-10.4 Draw the frequency table and Huffman tree for the following string:

```
"dogs do not spot hot pots or cats".
```

R-10.5 Give an example set of 10 characters and their associated frequencies so that, in the Huffman tree for this set, every internal node has an external-node child.

R-10.6 Give an example set of 8 characters and their associated frequencies so that the Huffman tree for this set is a complete binary tree.

R-10.7 Fred says that he ran the Huffman coding algorithm for the four characters, `A`, `C`, `G`, and `T`, and it gave him the code words, 0, 10, 111, 110, respectively. Give examples of four frequencies for these characters that could have resulted in these code words or argue why these code words could not possibly have been output by the Huffman coding algorithm.

R-10.8 Repeat the previous exercise for the code words, 0, 10, 101, 111.

R-10.9 Repeat the previous exercise for the code words, 00, 100, 101, 11.

R-10.10 Indicate for each of the lemmas used in the proof of correctness for the Huffman coding algorithm whether the proof of that lemma uses an exchange argument or a lower-bound argument?

Creativity

C-10.1 Provide an example instance of the fractional knapsack problem where a greedy strategy based on repeatedly choosing as much of the highest-benefit item as possible results in a suboptimal solution.

C-10.2 Suppose you are given an instance of the fractional knapsack problem in which all the items have the same weight. Show that you can solve the fractional knapsack problem in this case in $O(n)$ time.

C-10.3 Given a character string X of length n, describe an $O(n)$-time algorithm to construct the set, C, of distinct characters that appear in C, along with a count, $f(c)$, for each c in C, of how many times the character c appears in X. You may assume that the characters in X are encoded using a standard character indexing scheme, like the ASCII system.

C-10.4 A native Australian named Anatjari wishes to cross a desert carrying only a single water bottle. He has a map that marks all the watering holes along the way. Assuming he can walk k miles on one bottle of water, design an efficient algorithm for determining where Anatjari should refill his bottle in order to make as few stops as possible. Argue why your algorithm is correct.

C-10.5 Describe an efficient greedy algorithm for making change for a specified value using a minimum number of coins, assuming there are four denominations of coins (called quarters, dimes, nickels, and pennies), with values 25, 10, 5, and 1, respectively. Argue why your algorithm is correct.

C-10.6 Give an example set of denominations of coins so that a greedy change making algorithm will not use the minimum number of coins.

C-10.7 Suppose T is a Huffman tree for a set of characters having frequencies equal to the first n nonzero Fibonacci numbers, $\{1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots\}$, where $f_0 = 1$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$. Prove that every internal node in T has an external-node child.

C-10.8 Suppose you've been sent back in time and have arrived at the scene of an ancient Roman battle. Moreover, suppose you have just learned that it is your job to assign n spears to n Roman soldiers, so that each man has a spear. You observe that the men and spears are of various heights, and you have been told (in Latin) that the army is at its best if you can minimize the total difference in heights between each man and his spear. That is, if the ith man has height m_i and his spear has height s_i, then you want to minimize the sum,

$$\sum_{i=1}^{n} |m_i - s_i|.$$

Consider a greedy strategy of repeatedly matching the man and spear that minimizes the difference in heights between these two. Prove or disprove that this greedy strategy results in the optimal assignment of spears to men.

C-10.9 Consider again the time-travel problem of the previous exercise, but now consider a greedy algorithm that sorts the men by increasing heights and sorts the spears by increasing heights, and then assigns the ith spear in the ordered list of spears to the ith man in the ordered list of Roman soldiers. Prove or disprove that this greedy strategy results in the optimal assignment of spears to men.

C-10.10 Suppose you are organizing a party for a large group of your friends. Your friends are pretty opinionated, though, and you don't want to invite two friends if they don't like each other. So you have asked each of your friends to give you an "enemies" list, which identifies all the other people among your friends that they dislike and for whom they know the feeling is mutual. Your goal is to invite the largest set of friends possible such that no pair of invited friends dislike each other. To solve this problem quickly, one of your relatives (who is not one of your friends) has offered a simple greedy strategy, where you would repeatedly invite the person with the fewest number of enemies from among your friends who is not an enemy of someone you have already invited, until there is no one left who can be invited. Show that your relative's greedy algorithm may not always result in the maximum number of friends being invited to your party.

Applications

A-10.1 In the ***art gallery guarding*** problem we are given a line L that represents a long hallway in an art gallery. We are also given a set $X = \{x_0, x_1, \ldots, x_{n-1}\}$ of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

A-10.2 Consider a single ***machine scheduling*** problem, where we are given a set, T, of tasks specified by their start times and finish times, as in the task scheduling problem, except now we have only one machine and we wish to maximize the number of tasks that this single machine performs. Design a greedy algorithm for this single machine scheduling problem and show that it is correct. What is the running time of this algorithm?

A-10.3 A ***floating-point number*** is a pair, (m, d), of integers, which represents the number $m \times b^d$, where b is either 2 or 10. In any real-world programming environment, the sizes of m and d are limited; hence, each arithmetic operation involving two floating-point numbers may have some ***roundoff error***. The traditional way to account for this error is to introduce a ***machine precision*** parameter, $\epsilon < 1$, for the given programming environment, and bound roundoff errors in terms of this parameter. For instance, in the case of floating-point addition, $fl(x + y)$, for summing two floating-point numbers, x and y, we can write

$$fl(x + y) = (x + y) \cdot (1 + \delta_{x,y}),$$

where

$$|\delta_{x,y}| \leq \epsilon.$$

Suppose we are given a set of positive floating-point numbers, $\{x_1, x_2, \ldots, x_n\}$, and we wish to sum up all these numbers. Any such summation algorithm can be modeled with a binary expression tree, T, that has each x_i associated with one of its external nodes, with each internal node, v, representing the floating-point sum of the floating point numbers computed by v's children. Given the machine-precision approach to bounding floating-point errors, and assuming that ϵ is small enough so that ϵ^2 times any floating-point number is negligibly small, then we can use a term, e_n, to estimate an upper bound for the roundoff error for summing these numbers using the tree T as

$$e_n = \epsilon \sum_{i=1}^{n} x_i d(v_{x_i}),$$

where v_{x_i} is the external node associated with x_i and $d(v_{x_i})$ is the depth of this node in T. Design an efficient algorithm for constructing the binary expression tree, T, that minimizes e_n. What is the running time of your method?

A-10.4 Whenever a word processor or web browser displays a long passage of text, it must be broken up into lines of text that are displayed one on top of the other. Determining where to make these breaks is known as the ***line breaking*** problem. The most natural place to make such breaks are between words. So, suppose you are given a sequence, W, of n words, $W = (w_1, w_2, \ldots, w_n)$, where each word, w_i in W, has a given length, l_i. Also, for the sake of simplicity, let us ignore any spaces or punctuation that might come before or after any of these words. Suppose further that you are given a line length, L, that is an upper bound on the sum of the lengths of words that can fit on a single line. We specify how to break W into lines, by a sequence of indices, $(i_1, i_2, \ldots, i_k)$, where $i_1 = 1$, $i_k = n$, to indicate that we should break W into the lines, $w_{i_j} \ldots w_{i_{j+1}-1})$, for $j = 1, 2, \ldots, k-1$, subject to the constraints that $i_j < i_{j+1}$ and

$$\sum_{r=i_j}^{i_{j+1}-1} l_r \le L.$$

In addition, define the ***penalty*** for such a set of line breaks to be

$$\sum_{j=1}^{k} \left| L - \sum_{r=i_j}^{i_{j+1}-1} l_r \right| .$$

Describe an efficient greedy algorithm for breaking a sequence of words, W, into lines and prove that your method minimizes this penalty. What is the running time of your method?

A-10.5 Consider the line breaking problem from the previous exercise, but now consider changing the penalty for line breaks, so that it is now

$$\sum_{j=1}^{k} \left(L - \sum_{r=i_j}^{i_{j+1}-1} l_r \right)^2 .$$

Show that a greedy strategy of scanning W from beginning to end and making the choice that minimizes the term,

$$\left(L - \sum_{r=i_j}^{i_{j+1}-1} l_r \right)^2 ,$$

based on previous choices, while maintaining each line to be of length at most L, does not necessarily result in an optimal set of line breaks.

A-10.6 In the 2003 California gubernatorial recall election, the ballot contained 135 candidates, including people with various listings for their current job, including "actor," "comedian," and even "adult film actress." The winner was the actor-businessman Arnold Schwarzenegger, who got over 48% of the vote. Suppose we have the election results from such an election, with a large number, n, of candidates, and the only tool we can use to determine the winner is to encode the names of all the candidates using the Huffman coding algorithm, based on the number of votes each candidate got in this election. Suppose further that a friend of yours is guessing that if the winning candidate gets more than 40% of the votes, then his or her name will be encoded with a single bit. Prove that this conjecture is true and analyze the running time of this election-counting algorithm.

A-10.7 When data is transmitted across a noisy channel, information can be lost during the transmission. For example, a message that is sent through a noisy channel as

"WHO PARKED ON HARRY POTTER'S SPOT?"

could be received as the message,

"HOP ON POP"

That is, some characters could be lost during the transmission, so that only a selected portion of the original message is received. We can model this phenomenon using character strings, where, given a string $X = x_1x_2\ldots x_n$, we say that a string $Y = y_1y_2\ldots y_m$ is a ***subsequence*** of X if there are a set of indices $\{i_1, i_2, \ldots, i_k\}$, such that $y_1 = x_{i_1}$, $y_2 = x_{i_2}$, ..., $y_k = x_{i_k}$, and $i_j < i_{j+1}$, for $j = 1, 2, \ldots, k-1$. In a case of transmission along a noisy channel, it could be useful to know if our transcription of a received message is indeed a subsequence of the message sent. Therefore, describe an $O(n+m)$-time method for determining if a given string, Y, of length m is a subsequence of a given string, X, of length n.

Chapter Notes

The term "greedy algorithm" was coined by Edmonds [63] in 1971, although the concept existed before then. For more information about the greedy method and the theory that supports it, which is known as matroid theory, please see the book by Papadimitriou and Steiglitz [169]. The application of the greedy method we gave to the coding problem comes from Huffman [108].