UTEP Computer Science Technical Report UTEP-CS-09-12 Experiments in teaching an engaging and demystifying introduction to algorithms

Installment 1: Huffman Codes

Alan Siegel
New York University
siegel@cs.nyu.edu

Eric Freudenthal
University of Texas at El Paso
efreudenthal@utep.edu

Abstract

As is well known, Huffman's algorithm is a remarkably simple, and wonderfully illustrative example of how to use the greedy method to design algorithms. However, the Huffman coding problem, which is to find an optimal binary character code (or an optimal binary tree with weighted leaves) is intrinsically technical, and its specification is ill-suited for students with modest mathematical sophistication.

This difficulty is circumvented by introducing an alternative *precursor* problem that is very easy to understand, and where this understanding enables students to use the greedy method to find the solution themselves. The problem is how to merge k sorted lists of varying lengths as efficiently as possible. Once students have solved it, they are better prepared to understand the Huffman coding problem, and why it reduces trivially to the list merging problem they just solved. Even the correctness argument is simplified by this approach.

1 Introduction

Introducing abstract problems with unfamiliar mathematical underpinnings to students with modest mathematics backgrounds can be challenging for all participants involved in such an exercise. For example, Huffman's algorithm for finding an optimal binary encoding of a set of symbols with differing frequency counts is commonly taught to computer science students with little prior exposure to coding theory, greedy algorithms, probability, and optimization. Students with modest backgrounds may be at risk to miss fundamental content from traditional presentations of this topic not because they are unprepared for the concepts, but because they are ill prepared for the application.

When practical, we adopt teaching strategies that gently prepare students to solve abstract problems by introducing key concepts and problem-solving techniques in the context of concrete precursor problems selected from familiar domains. This TR illustrates this approach with a detailed example to show how these strategies were used to teach Hoffman's coding algorithm in classes at NYU and UTEP. To improve student-readiness for Huffman codes, we begin with a precursor list merging problem that requires basic familiarity with binary trees, the merging of sorted lists, and little else.

Section 2 presents *optimal list merging* to concretize the more subtle Huffman coding problems, and to separate its complications from the underlying algorithm design problem. Section 3 presents an optimality proof for the list merging algorithm, and introduces – en passant – some of the necessary content that is required for an understanding of Huffman codes. Section 4 introduces Huffman codes, explains additional underlying subtleties, and shows why these coding problems are, mutatis-mutandis, completely solved by the list merging algorithm of Section 2 and the correctness argument of Section 3. Finally, Section 5 offers commentary about our emerging pedagogical approach to low-overhead instruction in algorithms. Subsequent TRs will present additional examples of this pedagogy, as well as further developments of the underlying techniques.

2 Optimal merging as a precursor to Huffman Coding

In this section, we examine optimal merging as an easily understood precursor to the problem of designing optimal Huffman Codes.

Merging two sorted lists.

Let L and M be sorted lists of numbers with respective lengths ℓ and m. It is easy to merge them into a single sorted list Z with an operation count that is proportional to $\ell + m$. For specificity, suppose that each list is sorted to have increasing values. To merge them, one simply compares the first element in each list (if neither is empty), removes the smaller of the two values, and appends that element to be the last element in the growing list Z. The process is repeated until L and M are both empty.

The exact operation count is not at all important for the two-list merging problem, but for specificity, let's define an explicit cost function for merging two such sorted lists to be the sum of the element counts in each list. This is not an exact operation count, but has the advantage of being a precise cost that we can use to define the following optimization problem.

Merging k sorted lists.

Suppose that we have $k > 2$ lists of sorted numbers to merge together, and that they can only be merged pair-wise. Thus any two lists can be selected from the k lists, and merged to form a single list, which will leave $k - 1$ lists that remain to be merged. It follows that $k - 1$ pairwise mergings must take place, and the problem is to determine the sequence of pairwise merges where the sum of the costs for these $k - 1$ merges is as small as possible.

In more formal terms, the optimal merging problem is as follows.

Given: The k sorted lists $L_1, L_2, \ldots, L_k$ with respective lengths $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_k$.

Given: The cost to merge two sorted lists of ℓ and m elements is exactly $\ell + m$.

Question: What is the least cost way to use pairwise mergings of these lists to form one sorted list of $\ell_1 + \ell_2 + \cdots + \ell_k$ numbers?

For example, if $\ell_k = n$ and each of the other lengths is one, then the worst way to merge the lists is to combine ℓ_k with ℓ_1 and then to merge each singleton list one-by-one with the long list for a total cost of $(n+1) + (n+2) + (n+3) + \cdots + (n+k)$. If $n \geq k-1$, then the short lists should be combined first in some way, and then merged with L_k, so that the long L_k would be used in just one merging operation.

In a classroom setting, it is worthwhile emphasizing that there will be $k-1$ mergings no matter how the lists are selected. Students can also be asked which two lists would a lazy person select to merge first. It is not difficult to see that the two shortest lists should be merged first.

Virtually all students can be expected to see that the least-cost first step is to merge the two shortest lists first, and most will realize that the idea can be repeated.

This list merging problem illustrates a fundamental property of greedy algorithm design that may not be apparent to beginners: greedy algorithms are based on program invariants that repeatedly drive (and simplifies) the algorithm until it is done.

Some students might make a conceptual error by suggesting that the lists $L_1, L_2, \ldots$ should be merged one-by one with a growing list of consecutive merges, so that at iteration h, L_{h+1} would be merged with the merge of L_1 through L_h. It is easy to see that this approach does not follow the heuristic of always selecting the two shortest lists to merge together at each iteration. (Just set all of the list lengths to one. The rule will be broken at iteration 2).

Others might decide that the best solution is to merge L_1 with L_2, and then L_3 with L_4. It can be pointed out that many algorithms use dynamic decisions based on the evolution of the data, and this problem illustrates the point very well. If $\ell_1 = 1$, $\ell_2 = 1$, and $\ell_3 = 3$, then the second merging step must merge L_3 with the merge of L_1 and L_2.

Of course, most students will see the overall strategy. In pseudo-code, a solution might read:

```
function ListMerge(k, L_1, ..., L_k);
  Create an empty set S;
  for i ← 1 to k do
    Insert L_i into S
  endfor;
  while S has more than one list do
    A ← remove shortest list from S;
    B ← remove the shortest list from S;
    C ← Merge(A, B);
    Insert C into S
  endwhile;
  A ← remove list in S;
  return(A)
end_ListMerge;
```

As reasonable and complete as this program logic is, two key questions remain.

- Is this greedy (no look-ahead) approach really the best possible? (We need to prove this.)

- What is the best structure for managing the deletes?

The second question has an answer that students can readily understand, but are not so likely to discover. A naive approach would be to use a priority queue, which might require $\theta(k \log_2 k)$ operations. But the initial data is sorted, so the minimum size of the as yet unmerged lists can always be determined in constant time, since the initial set of k lists are sorted by size. As for the merged lists, their lengths are non decreasing; at each iteration of the **while** loop, the newly formed list will be at least as long as any of the previous merging results. So these new lists can be maintained in a queue, which will be guaranteed to be sorted by the algorithmic processing of increasingly longer lists. Thus the shortest list to delete at each step can be found by comparing the shortest list among the original as yet unprocessed lists, and the shortest of the (remaining) lists in the queue of merged lists.

We now address the correctness problem.

3 Is the greedy $ListMerge$ really optimal?

To figure out if this solution is indeed the most efficient way to merge the k lists, we need to know how to measure the number of operations that would be used to merge the lists in the way prescribed by $ListMerge$.

So the next question is how to model or formalize the $k-1$ mergings executed by $ListMerge$.

Curiously, very few students will realize that they know enough to answer the question. But it is still a good idea to ask a class to reflect on how the specific merge steps might be modeled or represented as an abstract process.

Students will need help in recognizing that the total cost to perform the merges can be formulated by following the bottom-up "data flow" of the lists in the merge tree. In our experience, a particularly effective prompt is to point at a particular leaf and ask how many different merging steps will use the contents of that list.

Let T be any merge tree such as (but not necessarily) the one generated by $ListMerge$. Let leaf L_i have depth $depth_i$ in T, for $i = 1, 2, \ldots, k$, and let $\mathcal{C}$ be the total cost for executing the $k-1$ pairwise merges prescribed by T.

Then $\mathcal{C}$ is the sum of list (leaf) lengths multiplied by the depth of the respective leaves. In more formal terms:

$$\mathcal{C} = \ell_1 \times depth_1 + \ell_2 \times depth_2 + \cdots + \ell_k \times depth_k,$$

where $depth_i$ is the depth of leaf L_i in the merge tree.

The instructor may wish to explain this relationship or might challenge the class to derive it via guided collaborative discussion.

And now that we have a precise way to compute the operation count for any merge tree of sorted lists, we can prove that the greedy algorithm $ListMerge$ builds a merge tree with the least cost.

We proceed with an implicitly inductive proof-of-correctness. It is obvious that the algorithm is the best possible when $k = 2$ because there is just one way to do the merging. So let k be the smallest index where this algorithm fails to find the most efficient solution, and let L_1 through L_k be the merging problem with explicit list lengths ℓ_1 through ℓ_k where there is a better solution.

Let the merge tree T_{LM} be the structure that results from the $ListMerge$ algorithm, and let T_{bp} be the tree that gives a best possible answer, which by assumption is less than the solution for T_{LM}.

We will show that T_{bp} is not better. So the assumption that there is a data set where the algorithm $ListMerge$ does not give the best possible tree is wrong.

Well, we know that T_{LM} merges the two shortest lists together as a first step. There could be several candidate pairs of equally shortest lists; it does not matter. Some two shortest pair of lists are the first to be merged by $ListMerge$.

What about T_{bp}?

T_{bp} has some deepest pair of leaves in the structure. (Notice that the deepest leaf must have a sibling leaf since it should be one level higher if its sibling is `Nil`, and its sibling cannot have any descendants since the leaf is the deepest vertex in the tree.) There might be just two deepest leaves, but there could be some larger (even number) of deepest leaves in T_{bp}. So where are those two shortest lists that are merged together by the first step for T_{LM}? If they are not at a deepest level in T_{bp}, then there must be some other pair of equally short lists at the deepest level: otherwise swapping a short list with a longer list at a deeper level of the tree will decrease the total work.

So we can conclude that by swapping equally sized lists, if necessary, the two shortest lists first processed by $ListMerge$ will be at the deepest level of (T_{bp} or an equally efficient variant of) T_{bp}. Now, there can be more than one pair of leaf-lists at this level, but the total work does not change if we rearrange these lowest level leaves, because the merging work to precess these lists completely is just the product of their lengths and their depth. So we can group those two lists to be siblings at the bottom level of (a possibly new but equally efficient) T_{bp}. So let us merge these two lists in the two trees.

The work to merge them is the same for both structures, and the two structures now solve the exact same merging problem for $k-1$ lists. But by hypothesis, we know that for $k-1$ lists, the reduced tree produced by $ListMerge$ is the best possible. That is, after one merging step, $ListMerge$ solves the problem of merging $k-1$ lists together, and by definition of k, must do so with a best possible solution. So the work to merge the $k-1$ lists in the reduced T_{bp} cannot be any less than that for T_{LM}. It follows that T_{LM} and T_{bp} must have the same merging cost, which shows that there is no k where the cost for the best possible tree T_{bp} is less than the cost computed by $ListMerge$.

Thus, we conclude $ListMerge$ finds an optimal ordering for merging k sorted lists for any k and any set of list lengths.

4 Reducing Huffman Codes to $ListMerge$

Our objective in this portion of the lesson is to help students understand that the metrics and algorithm for merging lists generalizes to solve the Huffman coding problems. Thus, the immediate objective is – as is standard – to teach the Huffman algorithm and correctness argument. At the same time, it is hoped that this approach reinforces a recurrent theme that any algorithm might be suitable for adaptation as a programming schema to solve not only similar – but even seemingly unrelated – problems of interest. The course curriculum includes the notion of problem equivalences (or reductions) where problems are transformed as opposed to algorithms, and the Huffman coding problem offers – en passant – an engaging and gentle introduction to this concept.

As a cost-minimizing algorithm, we have seen that $Listmerge$ builds the k-leaf binary tree with the smallest sum

$$\ell_1 \times depth_1 + \ell_2 \times depth_2 + \cdots + \ell_k \times depth_k,$$

where $depth_i$ is the depth of leaf-record L_i in the tree, and ℓ_i is the length of list L_i.

Students should be asked of the algorithms will work if the values ℓ_i are positive real numbers.

With some thought, it can be seen that nothing in the algorithm or correctness argument required the weights to be integers.[1]

At this point, the problem of designing an optimal Huffman tree can be introduced, and likewise the problem of designing optimal prefix-free character codes for a block of data with (integer) frequency counts ℓ_1, ℓ_2, ..., ℓ_k for a k-character alphabet. Similarly, the encoding of the characters by the zero-one interpretations of the paths from the tree root to its character leaves is readily introduced.

Below, we provide the necessary definitions and show that the solution algorithm $ListMerge$ and correctness argument are applicable solve these more complex problems with no modification whatsoever.

4.1 A Review of the Huffman coding problem

We begin by reducing the Huffman tree problem (HT) to the optimal list-merge problem, and then, as is standard, reduce Huffman coding (HC) to the Huffman tree problem. The simplest interpretation of the Huffman tree problem is to design an optimal binary tree T in the case where the records are stored as leaves, have prespecified probabilities of access, and have no constraints on their ordering in T:

> **Inputs**: The k pointers L_1, L_2, ..., L_k, and associated access probabilities p_1, p_2, ..., p_k.
>
> **Problem HT**: Build a k-leaf binary tree T where each leaf is a different one of the k pointers, and the expected tree depth of the leaves is as small as possible.

For this problem, the expected depth of the leaves is $\sum_{i=1}^{k} d_i \times p_i$, where d_i is the depth of pointer L_i in T, and there are no restrictions on the ordering of the pointers as leaves in T. For specificity, the depth of the root of T can be taken to be zero.

Of course, this problem is just a scaled version of the list merging problem as discussed in the previous subsection. So there is nothing new (apart from the irrelevant (and pragmatically unlikely) possibility that the probabilities might not be rational.

One of the standard applications of this problem is to use this tree for data compression. In somewhat informal terms, the data compression problem is as follows.

Suppose you have a fixed block of text, and want a fixed binary encoding for each letter, digit, punctuation mark, tab and space. What encoding uses the least number of bits?

Such an encoding is just a sequence of zeros and ones; there are no separators to partition the stream of zeros and ones into the encodings for the individual letters. One solution is to use a fixed number of bits (such as ASCII) for each character. But it is wasteful to use as many bits for a space (which appears frequently) as for the (infrequent) 'x.' It will be more efficient to use shorter encodings for the characters with a high frequency of use, and to use longer strings to encode the less common characters.

To ensure that there is no ambiguity in the separation of the stream into individual character encodings, the code should be prefix-unique, which means that no character has an encoding that is an initial prefix of some longer code for some other character.

[1] If students have difficulty understanding this, it may be desirable to *concretize* this subproblem by supposing that the ℓ_i are rational, and then rescaling the data by multiplying these weights by a common denominator.

For example, Morse code is not prefix-unique. It encodes the letter *j* as ·---, and the digit *1* as ·----. The code avoids ambiguity by requiring a pause between encoded characters, which permits the human decoder to separate the code for one character from the next. Thus, there is no confusion between the code for *jt* (·--- -) and the code for 1.

It is probably a very good idea to give humans that brief rest and marker between characters. But for computers, the pause is just a waste of time. With prefix-unique codes, a character is recognized as soon as its code has been transmitted, and the next bit is understood to begin the code for the next character. So with prefix-unique encodings, the characters can be extracted by a greedy bit-by-bit processing to extract the characters one-by one.[2]

The Huffman coding problem is: given a block of text, find a prefix-unique binary encoding for the characters in the text that results in an encoding that is as short as possible.

A more formal description of the coding problem is:

> **Input**: The k frequency counts n_1, n_2, ..., n_k for a block of text with $n_1 + n_2 + \cdots + n_k$ consecutive characters from an alphabet of k characters.
>
> **Problem HC**: Find k prefix-unique binary codes b_1, b_2, ..., b_k with bit lengths ℓ_1, ℓ_2, ..., ℓ_k where $\sum_{i=1}^{k} n_i \times \ell_i$ is as small as possible.

As is well known, the solution to problem HC is just to build the least cost weighted binary tree with k leave that have the leaf weightings ℓ_1, ℓ_2, ..., ℓ_k in some order, and the tree cost $\sum_{i=1}^{k} d_1 \times \ell_i$ is as small as possible. The codes are read off the tree by encoding the path from the root to a leaf as a sequence of zeros and ones where a zero corresponds to the selection of the left descendant as the next step on the path, and a one represents the selection of the right descendant. Each code has a bit count that is exactly depth of the record that it represents in the Hoffman tree. Since each record is a leaf, the records are prefix-unique, and likewise, every prefix-unique encoding can be represented by such a binary tree.

Thus, a best solution to the HT problem with weights ℓ_1, ℓ_2, ..., ℓ_k gives a code for problem HC. Is it the best possible? Well, suppose that there is a better prefix-unique code for HC. Its zero-one bits define a binary tree T_{HC} where each character is a leaf because it is prefix free. And if T_{HC} were better for the HC problem, then it would also be better for the HT problem.

In short, these two problems are equivalent.

For completeness, it is worth noting that such a coding will not give the best possible data compression. The problem statement is worded to prohibit codes that adapt as the frequencies of character usage changes in the text, or that builds codes for sequences of letters. For example, in English, some extra efficiency would result by providing a special code for the word "the." A small gain would occur by providing a code for the 2-character sequence **qu**, and allocating a longer code for the very rare q that is not followed by a u. But the Huffman coding problem is restricted to the encoding of individual characters, punctuation, spaces, etc. Nevertheless, this algorithms does provide an nice introduction to information theory and data compression – not to mention the use of the greedy method to design algorithms.

[2] Actually, it is worth noting that Morse code is really a prefix-unique ternary encoding scheme that uses the three symbols ·, -, and ' ' (a space). It is prefix-unique because each character code ends with a space, and has no spaces anywhere else within the code sequence. However, this limited use of the space character is wasteful, since a good ternary code would make full use of all three characters to provide better data compression: there would be far more short words to use as character codes.

5 The underlying pedagogy

Subjects that require mathematical sophistication present difficulties for students with modest mathematics skills. Combinatorial algorithms is a subject that concerns the modeling and solving of a rich and widely varying set of problems. As such, it is steeped in, and enriched by a substantial mathematical heritage. Yet the subject does not require extensive mathematical knowledge, although some degree of mathematical reasoning is absolutely necessary. The overall objective behind this article and the project that has produced it is to design a high quality algorithms course that is accessible to as wide an audience of students as possible, and to do so effectively. The educational, economic, and sociological reasons for the democratization of technical knowledge and access to computer science in particular are manifest. At issue is how to present the full content of a standard curriculum (and more) without leaving a significant percentage of a class lost due to modest mathematics skills.

In the process of experimenting with many different approaches, the authors of this report have noticed that certain techniques have consistently emerged as the most successful. This report presents two of these pedagogical perspectives, and illustrates (with additional examples to follow) their application in the teaching of a standard algorithms problem: designing Huffman codes.

Principle 1. Build chains of reasoning that are rigorous but are as short and background-free as possible.
Mathematical arguments, by way of contrast, often build long chains of definitions, lemmas, and other logical constructs that are readily absorbed by those with significant training, but which soon lose all others as the abstractions and dependence on newly internalized content build.

When teaching algorithms, the intermediate definitions and complex arguments may seem difficult to circumvent, and some of these intermediate reasonings will even have conceptual value in their own right. These difficulties are precisely why improvements in teaching algorithms is intrinsically evolutionary as opposed to revolutionary, and why the approach is worthy of investigation in its own right. This TR series is intended to support the contention that there are a rich set of improved presentations that await those willing to search for them. Thus, this TR is written because Principle 1, though easy to state, is difficult to implement: Algorithm presentations should follow the shortest, simplest chain of rigorous reasoning that leads to algorithmic understanding. The unanswered question is how. In this TR, we endeavored to offer an evolving answer for Huffman codes.

This first principle includes an implicit mandate that both the problem and its solution be as easy to state and implement as possible. However, if an idea that has been eliminated from the simplified presentation has educational value, then it should be introduced *after* the algorithm has been taught, and used to reinforce or refine student understanding as opposed to using it to insert yet another link in a chain of reasonings that lead to an uncertain (cognitive) end. With this reversal in sequencing, students may well be better prepared and better motivated to internalize the idea because it is built on top of a fundamental technique (the algorithm).

Principle 2. Restructure abstract ideas to present them in as concrete a manner as possible.

Of course, such a prescription is open ended, and easy to misinterpret. Moreover, the use of abstraction has been a matter of contention between two very different schools of thought about best practices in teaching. In the majority of k-12 mathematics education studies and reform textbooks, there has been an unmistakable movement to avoid abstraction because students find it difficult to absorb. In the cognitive psychology literature, the prevailing view is that abstractions must be explicitly taught because naive learners are unable to extract fundamental principles and systematic methods from examples alone.

In view of this controversy, it would seem appropriate to say what kinds of concretized

abstraction seem to work and what does not. Based on years of teaching certain content that is difficult to formalize, we have accumulated informal — but convincing — evidence to support the notion that examples of clear ideas are inadequate for most students, whereas the same content is absorbed far more successfully when taught in conjunction with abstract formulations. Thus, our approach to concretize abstractions is not to eliminate the teaching of abstract principles, but rather to teach them in as concrete a formulation as possible, and to merge – as much as possible – the use of examples and the application of abstractions into a unified presentation where each part supports the other.

In this TR, the real problem of interest had nothing to do with the merging of sorted lists. The emphasis on list merging was a very concrete ruse to enable the greedy method to be applied before introducing the structural analysis needed to quantify the total cost of a merge tree, and to eliminate any confusions that might be caused by replacing a subtree of weighted leaves by a leaf whose weight is the sum of the leaf-weights in the subtree.

References

[**1**] Anderson, J.R., Reder, L.M., & Simon, H.A. Applications and Misapplications of Cognitive Psychology to Mathematics Education. Texas Educational Review, (2000, Summer).

[**2**] Schwartz, D.L. and Bransford, J. D. A Time for Telling. Cognition and Instruction, (1998).