Optimal Prefix Free Codes with Partial Sorting

(Extended abstract*)

Jérémy Barbay

Departamento de Ciencias de la Computación (DCC),
Universidad de Chile,
Santiago, Chile
jeremy@barbay.cl

Abstract

We describe an algorithm computing an optimal prefix free code for n unsorted positive weights in less time than required to sort them on many large classes of instances, identified by a new measure of difficulty for this problem, the alternation α. This asymptotical complexity is within a constant factor of the optimal in the algebraic decision tree computational model, in the worst case over all instances of fixed size n and alternation α. Such results refine the state of the art complexity in the worst case over instances of size n in the same computational model, a landmark in compression and coding since 1952, by the mere combination of van Leeuwen's algorithm to compute optimal prefix free codes from sorted weights (known since 1976), with Deferred Data Structures to partially sort multisets (known since 1988).

1998 ACM Subject Classification F.2.2 ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY/Nonnumerical Algorithms and Problems (Sorting and Searching), E.4 CODING AND INFORMATION THEORY (Data compaction and compression)

Keywords and phrases Deferred Data Structure, Huffman, Median, Optimal Prefix Free Codes, van Leeuwen.

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Given n positive weights $W[1..n]$ coding for the frequencies $\left\{W[i]/\sum_{j=1}^{n} W[j]\right\}_{i\in[1..n]}$ of n messages, and a number D of output symbols, an Optimal Prefix Free Code [11] is a set of n code strings on alphabet $[1..D]$, of variable lengths $L[1..n]$ and such that no string is prefix of another, and the average length of a code is minimized (i.e. $\sum_{i=1}^{n} L[i]W[i]$ is minimal).

Any prefix free code can be computed in linear time from a set of code lengths satisfying the Kraft inequality $\sum_{i=1}^{n} D^{-L[i]} \leq 1$. The original description of the code by Huffman [11] yields a heap-based algorithm performing $O(n \log n)$ algebraic operations, using the bijection between D-ary prefix free codes and D-ary cardinal trees [8]. This complexity is asymptotically optimal for any constant value of D in the algebraic decision tree computational model[1], in the worst case over instances composed of n positive weights, as computing the optimal binary prefix free code for the weights $W[0,\dots,Dn] =$

* See the full version [1] on http://arxiv.org/abs/1602.03934 for complete proofs and comments.

[1] The algebraic decision tree computational model is composed of algorithms which can be modelled as a decision tree where the decision made in each node is based only on algebraic operations on the input.

$\{D^{x_1}, \ldots, D^{x_1}, D^{x_2}, \ldots, D^{x_2}, \ldots, D^{x_n}, \ldots, D^{x_n}\}$ is equivalent to sorting the positive integers $\{x_1, \ldots, x_n\}$. We consider here only the binary case, where $D = 2$.

Yet, not all instances require the same amount of work to compute an optimal code:

- When the weights are given in sorted order, van Leeuwen [14] showed that an optimal code can be computed using within $O(n)$ algebraic operations.
- When the weights consist of $r \in [1..n]$ distinct values and are given in a sorted, compressed form, Moffat and Turpin [17] showed how to compute an optimal code using within $O(r(1 + \log(n/r)))$ algebraic operations, which is often sublinear in n.
- In the case where the weights are given unsorted, Belal *et al.* [5, 6] described several families of instances for which an optimal prefix free code can be computed in linear time, along with an algorithm claimed to perform $O(kn)$ algebraic operations, in the worst case over instances formed by n weights such that there is an optimal binary prefix free code with k distinct code lengths[2]. This complexity was later downgraded to $O(16^k n)$ in an extended version[4] of their article. Both results are better than the state of the art when k is finite, but worse when k is larger than $\log n$.

In the context described above, various questions are left unanswered, from the confirmation of the existence of an algorithm running in time $O(16^k n)$ or $O(kn)$, to the existence of an algorithm taking advantage of small values of both n and k, less trivial than running two algorithms in parallel and stopping both whenever one computes the answer. Given n positive integer weights, *can we compute an optimal binary prefix free code in time better than $O(\min\{kn, n \log n\})$ in the algebraic decision tree computational model?* We answer in the affirmative for many classes of instances, identified by the alternation measure α defined in Section 3.1:

▶ **Theorem 1.** *Given n positive weights of alternation $\alpha \in [1..n-1]$, there is an algorithm which computes an optimal binary prefix free code using within $O(n(1+\log\alpha)) \subseteq O(n \lg n)$ algebraic instructions, and this complexity is asymptotically optimal among all algorithms in the algebraic decision tree computational model in the worst case over instances of size n and alternation α.*

Proof. We show in Lemma 12 that any algorithm A in the algebraic decision tree computational model performs within $\Omega(n \lg \alpha)$ algebraic operations in the worst case over instances of size n and alternation α. We show in Lemma 9 that the `GDM` algorithm, a variant of the van Leeuwen's algorithm [14], modified to use the deferred data structure from Lemma 5, performs $q \in O(\alpha(1+\lg \frac{n-1}{\alpha}))$ such queries, which yields in Corollary 10 a complexity within $O(n(1+\log\alpha) + \alpha(\lg n)(\lg \frac{n}{\alpha}))$, all within the algebraic decision tree computational model. As $\alpha \in [1..n-1]$ and $O(\alpha(\lg n)(\lg \frac{n}{\alpha})) \subseteq O(n(1+\log\alpha))$ for this range (Lemma 11), the optimality ensues. ◀

We discuss our solution in Section 2 in three parts: the intuition behind the general strategy in Section 2.1, the deferred data structure which maintains a partially sorted list of weights while supporting `rank`, `select` and `partialSum` queries in Section 2.2, and the algorithm which uses those operators to compute an optimal prefix free code in Section 2.3. Our main contribution consists in the analysis of the running time of this solution, described in Section 3: the formal definition of the parameter of the analysis in Section 3.1, the upper bound in Section 3.2 and the matching lower bound in Section 3.3. We conclude with a comparison of our results with those from Belal *et al.* [5] in Section 4.

[2] Note that k is not uniquely defined, as for a given set of weights there can exist several optimal prefix free codes varying in the number of distinct code lengths used.

2 Solution

The solution that we describe is a combination of two results: some results about deferred data structures for multisets, which support queries in a "lazy" way; and some results about the relation between the computational cost of sorting and that of computing an optimal prefix free code. We describe the general intuition of our solution in Section 2.1, the deferred data structure in Section 2.2, and the algorithm in Section 2.3.

2.1 General Intuition

The algorithm suggested by Huffman [11] starts with a heap of external nodes, selects the two nodes of minimal weight, pairs them into a new node which it adds to the heap, and iterates untill only one node is left. Whereas the type of the nodes selected, external or internal, does not matter in the analysis of the complexity of Huffman's algorithm, we claim that the computational cost of optimal prefix free codes can be greatly reduced on instances where many external nodes are selected consecutively. We define the "`EI` signature" of an instance as the first step toward the characterization of such instances:

▶ **Definition 2.** Given an instance of the optimal prefix free code problem formed by n positive weights $W[1..n]$, its ***EI** signature* $\mathcal{S}(W) \in \{E, I\}^{2n-1}$ is a string of length $2n-1$ over the alphabet $\{E, I\}$ (where E stands for "External" and I for "Internal") marking, at each step of the algorithm suggested by Huffman [11], whether an external or internal node is chosen as the minimum (including the last node returned by the algorithm, for simplicity).

The analysis described in Section 3 is based on the number $|S|_{EI}$ of blocks formed only of E in the `EI` signature of the instance S. We can already show some basic properties of this measure:

▶ **Lemma 3.** *Given the `EI` signature S of n unsorted positive weights $W[1..n]$, $|S|_E = n$; $|S|_I = n-1$; $|S| = 2n-1$; S starts with two E; S finishes with one I; $|S|_{EI} = |S|_{IE}+1$; $|S|_{EI} \in [1..n-1]$.*

Proof. The three first properties are simple consequences of basic properties on binary trees. S starts with two E as the first two nodes paired are always external. S finishes with one I as the last node returned is always (for $n > 1$) an internal node. The two last properties are simple consequences of the fact that S is a binary string starting with an E and finishing with an I. ◀

For example, the text T = "ABBCCCDDDDEEEEEFFFFFGGGGGGHHHHHHH" has frequencies $W = \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{5}\boxed{6}\boxed{7}$. It corresponds to an instance of size $n = 8$, of ***EI** signature* $\mathcal{S}(W)$ = `EEEIEEEEIEIIIII` of length 15, which starts with EE, finishes with I, and contains only $\alpha = 3$ occurrences of EI, corresponding to a decomposition into $\alpha = 3$ maximal blocks of consecutive Es.

Instances such as this, with very few blocks of E, are easier to solve than instances with many such blocks. For example, an instance W of length n such that its `EI` signature $\mathcal{S}(W)$ is composed of a single run of n Es followed by a single run of $n-1$ Is (such as the one described in Figure 1) can be solved in linear time, and in particular without sorting the weights: it is enough to assign the codelength $l = \lfloor \log_2 n \rfloor$ to the $n - 2^l$ largest weights and the codelength $l+1$ to the 2^l smallest weights. Separating those weights is a simple `select` operation, supported by the data structures described in the following section.

We describe two extreme examples. First, consider the text T = "`ba_bb_caca_ba_cc`". Each of the four symbols of its alphabet $\{a, b, c, \_\}$ occurs exactly 4 times, so that an optimal

a	4
b	4
c	4
_	4

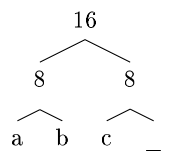

Figure 1 Frequencies and code tree for the text T = "ba_bb_caca_ba_cc", minimizing the number of occurrences of "EI" in its `EI` signature $\mathcal{S}(T)$ = "EEEEIII".

a	8
b	1
c	2
_	4

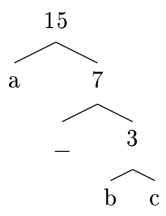

Figure 2 Frequencies and code tree for the text T = "aaaaaaaaabcc____", maximizing the number of occurrences of "EI" in its `EI` signature $\mathcal{S}(T)$ = "EEIEIEI".

prefix free code assigns a uniform codelength of 2 bits to all symbols (see Figure 1). There is no need to sort the symbols by frequency (and the prefix free code does not yield any information about the order in which the symbols would be sorted by monotone frequencies), and accordingly the `EI` signature of this text, $\mathcal{S}(T)$ = "EEEEIII", has a single block of Es, indicating a very easy instance. The same holds if the text is such that the frequencies of the symbols are all within a factor of two of each other. On the other hand, consider the text T = "aaaaaaaaabcc____", where the frequencies of its symbols follow an exponential distribution, so that an optimal prefix free code assigns different codelengths to almost all symbols (see Figure 2). The prefix free code does yield a lot of information about the order in which the symbols would be sorted by monotone frequencies, and accordingly the `EI` signature of this text, $\mathcal{S}(T)$ = "EEIEIEI", has three blocks of Es, indicating a more difficult instance. The same holds with more general distribution, as long as no two pairs of symbol frequencies are within a factor of two of each other.

2.2 Partial Sum Deferred Data Structure

Given a Multiset $W[1..n]$ on alphabet $[1..\sigma]$ of size n, Karp *et al.* [13] defined the first deferred data structure supporting for all $x \in [1..\sigma]$ and $r \in [1..n]$ queries such as `rank`(x), the number of elements which are strictly smaller than x in W; and `select`(r), the value of the r-th smallest value (counted with multiplicity) in W. Their data structure supports q queries in time within $O(n(1 + \lg q) + q \lg n)$, all in the comparison model.

Karp *et al.*'s data structure [13] supports only `rank` and `select` queries in the comparison model, whereas the computation of optimal prefix free codes requires to sum pairs of weights from the input, and the algorithm that we propose in Section 2.3 requires to sum weights from a range in the input. Such a requirement can be reduced to `partialSum` queries. Whereas `Partial Sum` queries have been defined in the literature based on the positions in the input array, we define such queries here in a way that depends only on the content of the Multiset (as opposed to a definition depending on the order in which it is given), so that it can be generalized to deferred data structures.

▶ **Definition 4.** Given n unsorted positive weights $W[1..n]$, a `Partial Sum` data structure supports the following queries: `rank`(x), the number of elements which are strictly smaller than x in W; `select`(r), the value of the r-th smallest value (counted with multiplicity) in W; `partialSum`(r), the sum of the r smallest elements (counted with multiplicity) in W.

For example, given the array $A = \boxed{5 \mid 3 \mid 1 \mid 5 \mid 2 \mid 4 \mid 6 \mid 7}$, this definition of the operators yields `rank`$(5) = 4$, `select`$(6) = 5$, and `partialSum`$(2) = 3$.

We describe below how to extend Karp *et al.*'s deferred data structure [13], which supports `rank` and `select` queries on MULTISETS, in order to add the support for `partialSum` queries, with an amortized running time within a constant factor of the original asymptotic time. Note that the operations performed by the data structure are not any more within the comparison model, but rather in the algebraic decision tree computational model, as they introduce algebraic operations (additions) on the elements of the MULTISET. The result is a direct extension of Karp *et al.* [13], adding a sub-task taking linear time (updating partial sums in an interval of positions) to a sub-task which was already taken linear time (partitioning this same interval by a pivot):

▶ **Lemma 5.** *Given n unsorted positive weights $W[1..n]$, there is a `PartialSum` Deferred Data Structure which supports q operations of type `rank`, `select` and `partialSum` in time within $O(n(1+\lg q)+q(1+\log n))$, all within the algebraic decision tree computational model.*

Proof. Karp *et al.* [13] described a deferred data structure which supports the `rank` and `select` queries (but not `partialSum` queries). It is based on median computations and $(2,3)$-trees, and performs q queries on n values in time within $O(n(1+\lg q)+q(1+\log n))$, all within the algebraic decision tree computational model. We describe below how to modify their data structure in a simple way to support `partialSum` queries with asymptotically negligible additional cost. At the initialization of the data structure, compute the n partial sums corresponding to the n positions of the unsorted array. After each median computation and partitioning in a `rank` or `select` query, recompute the partial sums on the range of values newly partitioned, adding only a constant factor to the cost of the query. When answering a `partialSum` query, perform a `select` query and then return the value of the partial sum corresponding to the value by the `select` query: the asymptotic complexity is within a constant factor of the one described by Karp *et al.* [13]. ◀

In the next section we describe an algorithm that uses the deferred data structure described above to batch the operations on the external nodes, and to defer the computation of the weights of some internal nodes for later, so that for many instances the input is not completely sorted at the end of the execution, which reduces the execution cost.

2.3 Algorithm "`Group-Dock-Mix`" (`GDM`)

There are five main phases in the `GDM` algorithm: the *Initialization*, three phases (*Grouping*, *Docking* and *Mixing*, giving it the name "`GDM`" to the algorithm) inside a loop running until only internal nodes are left to process, and the *Conclusion*:

- In the *Initialization* phase, initialize the `Partial Sum` deferred data structure with the input, and the first internal node by pairing the two smallest weights of the input.
- In the *Grouping* phase, group the weights smaller than the smallest internal node: this corresponds to a run of consecutive E in the `EI` signature of the instance.
- In the *Docking* phase, pair the consecutive *positions* of those weights (as opposed to the weights themselves, which can be reordered by future operations) into internal nodes, and pair those internal nodes until the weight of at least one such internal node becomes equal or larger than the smallest remaining weight: this corresponds to a run of consecutive I in the `EI` signature of the instance.
- In the *Mixing* phase, rank the smallest unpaired weight among the weights of the available internal nodes: this corresponds to an occurrence of IE in the `EI` signature of the instance. This is the most complicated (and most costly) phase of the algorithm.

- In the *Conclusion* phase, with i internal nodes left to process, assign codelength $l = \lfloor \log_2 i \rfloor$ to the $i - 2^l$ largest ones and codelength $l{+}1$ to the 2^l smallest ones: this corresponds to the last run of consecutive I in the `EI` signature of the instance.

The algorithm and its complexity analysis distinguish two types of internal nodes: *pure* nodes, which descendants were all paired during the same *Grouping* phase; and *mixed* nodes, each of which either is the ancestor of such a *mixed* node, or pairs a *pure* internal node with an external node, or pairs two *pure* internal nodes produced at distinct phases of the `GDM` algorithm. The distinction is important as the algorithm computes the weight of any *mixed* node at its creation (potentially generating several data structure operations), whereas it defers the computation of the weight of some *pure* nodes for later, and does not compute the weight of some pure nodes.

Before describing each phase more in detail, it is important to observe the following invariant of the algorithm:

▶ **Lemma 6.** *Given an instance of the optimal prefix free code problem formed by $n > 1$ positive weights $W[1..n]$, between each phase of the algorithm, all unpaired internal nodes have weight within a constant factor of two (i.e. the maximal weight of an unpaired internal node is strictly smaller than twice the minimal weight of an unpaired internal node).*

We now proceed to describe each phase in more details:

- **Initialization:** Initialize the deferred data structure `Partial Sum` with the input; compute the weight `currentMinInternal` of the first internal node through the operation `partialSum`(2) (the sum of the two smallest weights); create this internal node, of weight `currentMinInternal` and children 1 and 2 (the positions of the first and second weights, in any order); compute the weight `currentMinExternal` of the first unpaired weight (i.e. the first available external node) by the operation `select`(3); setup the variables `nbInternals` = 1 and `nbExternalProcessed` = 2.
- **Grouping:** Compute the position r of the first unpaired weight which is larger than the smallest unpaired internal node, through the operation `rank`(`currentMinInternal`); pair the (($r -$ `nbExternalProcessed`) modulo 2) indices to form $\lfloor \frac{r-\texttt{nbExternalProcessed}}{2} \rfloor$ *pure* internal nodes; if the number $r-$`nbExternalProcessed` of unpaired weights smaller than the first unpaired internal node is odd, select the r-th weight through the operation `select`(r), compute the weight of the first unpaired internal node, compare it with the next unpaired weight, to form one *mixed* node by combining the minimal of the two with the extraneous weight.
- **Docking:** Pair all internal nodes by batches (by Lemma 6, their weights are all within a factor of two, so all internal nodes of a generation are processed before any internal node of the next generation); after each batch, compare the weight of the largest such internal node (compute it through `partialSum` on its range if it is a *pure* node, otherwise it is already computed) with the first unpaired weight: if smaller, pair another batch, and if larger, the phase is finished.
- **Mixing:** Rank the smallest unpaired weight among the weights of the available internal nodes by a doubling search starting from the beginning of the list of internal nodes. For each comparison, if the internal node's weight is not already known, compute it through a `partialSum` operation on the corresponding range (if it is a *mixed* node, it is already known). If the number r of internal nodes of weight smaller than the unpaired weight is odd, pair all but one, compute the weight of the last one and pair it with the unpaired weight. If r is even, pair all of the r internal nodes of weight smaller than the unpaired weight, compare the weight of the next unpaired internal node with the weight of the

next unpaired external node, and pair the minimum of the two with the first unpaired weight. If there are some unpaired weights left, go back to the *Grouping* phase, otherwise continue to the *Conclusion* phase.

- **Conclusion:** There are only internal nodes left, and their weights are all within a factor of two from each other. Pair the nodes two by two in batches as in the *Docking* phase, computing the weight of an internal node only when the number of internal nodes of a batch is odd.

The combination of those phases forms the `GDM` algorithm, which computes an optimal prefix free code given an unsorted sets of positive integers. In the next section, we analyze the number q of `rank`, `select` and `partialSum` queries performed by the `GDM` algorithm, and deduce from it the complexity of the algorithm in terms of algebraic operations.

3 Analysis

The `GDM` algorithm runs in time within $O(n \lg n)$ in the worst case over instances of size n (which is optimal (if not a new result) in the algebraic decision tree computational model), but much faster on instances with few blocks of consecutive Es in the `EI` signature of the instance. We formalize this concept by defining the *alternation* α of the instance in Section 3.1. We then proceed in Section 3.2 to show upper bounds on the number of queries and operations performed by the `GDM` algorithm in the worst case over instances of fixed size n and alternation α. We finish in Section 3.3 with a matching lower bound for the number of operations performed.

3.1 Alternation $\alpha(W)$

We suggested in Section 2.1 that the number $|S|_{EI}$ of blocks of consecutive Es in the `EI` signature of an instance can be used to measure its difficulty. Indeed, some "easy" instances have few such blocks, and the instance used to prove the $\Omega(n \lg n)$ lower bound on the computational complexity of optimal prefix free codes in the algebraic decision tree computational model in the worst case over instances of size n has $n-1$ such blocks (the maximum possible in an instance of size n). We formally define this measure as the "alternation" of the instance (it measures how many times the van Leeuwen algorithm "alternates" from an external node to an internal node) and denote it by the parameter α:

▶ **Definition 7.** Given an instance of the optimal prefix free code problem formed by n positive weights $W[1..n]$, its *alternation* $\alpha(W) \in [1..n-1]$ is the number of occurrences of the substring "EI" in its `EI` signature $\mathcal{S}(W)$.

This number is of particular interest as it measures the number of iteration of the main loop in the `GDM` algorithm:

▶ **Lemma 8.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm performs α iterations of its main loop.*

In the next section, we refine this result to the number of data structure operations and algebraic operations performed by the `GDM` algorithm.

3.2 Upper Bound

In order to measure the number of queries performed by the `GDM` algorithm, we detail how many queries are performed in each phase of the algorithm.

- The *Initialization* corresponds to a constant number of data structure operations: a `select` operation to find the third smallest weight, and a simple `partialSum` operation to sum the two smallest weights of the input.
- Each *Grouping* phase corresponds to a constant number of data structure operations: a `partialSum` operation to compute the weight of the smallest internal node if needed, and a `rank` operation to identify the unpaired weights which are smaller or equal to this node.
- The number of operations performed by each *Docking* and *Mixing* phase is better analyzed together: if there are i symbols in the I-block corresponding to this phase in the `EI` signature, and if the internal nodes are grouped on h levels before generating an internal node larger than the smallest unpaired weights, the *Docking* phase corresponds to at most h `partialSum` operations, whereas the *Mixing* phase corresponds to at most $\log_2(i/2^h)$ `partialSum` operations, which develops to $\log_2(i) - h$, for a total of $h + \log_2(i) - h = \log_2 i$ data structure operations.
- The *Conclusion* phase corresponds to a number of data structure operations logarithmic in the size of the last block of Is in the `EI` signature of the instance: in the worst case, the weight of one *pure* internal node is computed for each batch, through one single `partialSum` operation each time.

Lemma 8 and the concavity of the log yields the total number of data structure operations performed by the `GDM` algorithm:

▶ **Lemma 9.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm performs within $O(\alpha(1 + \lg \frac{n-1}{\alpha}))$ data structure operations on the deferred data structure given as input.*

Proof. For $i \in [1..\alpha]$, let n_i be the number of internal nodes at the beginning of the i-th *Docking* phase. According to Lemma 8 and the analysis of the number of data structure operations performed in each phase, the `GDM` algorithm performs in total within $O(\alpha + \sum_{i=1}^{\alpha} \lg n_i)$ data structure operations. Since there are at most $n - 1$ internal nodes, by concavity of the logarithm this is within $O(\alpha + \alpha \lg \frac{n-1}{\alpha}) = O(\alpha(1 + \lg \frac{n-1}{\alpha}))$. ◀

Combining this result with the complexity of the `Partial Sum` deferred data structure from Lemma 5 directly yields the complexity of the `GDM` algorithm in algebraic operation (and running time):

▶ **Lemma 10.** *Given an instance of the optimal prefix free code problem of alternation α, the `GDM` algorithm runs in time within $O(n(1+\log\alpha) + \alpha(\lg n)(1 + \lg \frac{n-1}{\alpha}))$, all within the algebraic decision tree computational model.*

Proof. Let q be the number of queries performed by the `GDM` algorithm. Lemma 9 implies that $q \in O(\alpha(1 + \lg \frac{n-1}{\alpha}))$. Plunging this into the complexity of $O(q \lg n + n \lg q)$ from Lemma 5 yields the complexity $O(n(1+\log\alpha) + \alpha(\lg n)(1 + \lg \frac{n-1}{\alpha}))$. ◀

Some simple functional analysis further simplifies the expression to our final upper bound:

▶ **Lemma 11.** *Given two positive integers $n > 0$ and $\alpha \in [1..n-1]$,*

$$O(\alpha(\lg n)(\lg \frac{n}{\alpha})) \subseteq O(n(1 + \lg \alpha))$$

Proof. Given two positive integers $n > 0$ and $\alpha \in [1..n-1]$, $\alpha < \frac{n}{\lg n}$ and $\frac{\alpha}{\lg \alpha} < n$. A simple rewriting yields $\frac{\alpha}{\lg \alpha} < \frac{n}{\lg^2 n}$ and $\alpha \lg^2 n > n \lg \alpha$. Then, $n/\alpha < n$ implies $\alpha \times \lg n \times \lg \frac{n}{\alpha} < n \lg \alpha$, which yields the result. ◀

In the next section, we show that this complexity is indeed optimal in the algebraic decision tree computational model, in the worst case over instances of fixed size n and alternation α.

3.3 Lower Bound

A complexity within $O(n(1+\lg\alpha))$ is exactly what one could expect, by analogy with the sorting of MULTISETS: there are α groups of weights, so that the order within each group does not matter much, but the order between weights from different groups matter a lot. We prove a lower bound within $\Omega(n\lg\alpha)$ by reduction to MULTISET sorting:

▶ **Lemma 12.** *Given the integers $n \leq 2$ and $\alpha \in [1..n-1]$, for any algorithm A in the algebraic decision tree computational model, there is a set $W[1..n]$ of n positive weights of alternation α such that A performs within $\Omega(n\lg\alpha)$ algebraic operations.*

Proof. For any MULTISET $A[1..n] = \{x_1,\dots,x_n\}$ of n values from an alphabet of α distinct values, define the instance $W_A = \{2^{x_1},\dots,2^{x_n}\}$ of size n, so that computing an optimal prefix free code for W, sorted by codelength, provides an ordering for A. W has alternation α: for any two distinct values x and y from A, the van Leeuwen algorithm pairs all the weights of value 2^x before pairing any weight of value 2^y, so that the `EI` signature of W_A has α blocks of consecutive Es. The lower bound then results from the classical lower bound on sorting MULTISETS in the comparison model in the worst case over MULTISETS of size n with α distinct symbols, itself based on the number α^n of such multisets. ◀

We compare our results to previous ones in the next section.

4 Discussion

We described an algorithm computing an optimal prefix free code for n unsorted positive weights in time within $O(n(1+\lg\alpha)) \subseteq O(n\lg n)$, where the alternation $\alpha \in [1..n-1]$ roughly measures the amount of sorting required by the computation by combining van Leeuwen's results about optimal prefix free codes [14], known since 1976, with Karp *et al.*'s results about Deferred Data Structures [13], known since 1988. The results described above yield many new questions, of which we discuss only a few in the following sections: how do those results relate to previous results on optimal prefix free codes (Section 4.1), or to other results on Deferred Data Structures obtained since 1988 (Section 4.2 and 4.3). We discuss the potential lack of practical applications of our results on optimal prefix free codes in Section 4.4, and the perspectives of research on this topic in Section 4.5.

4.1 Relation to previous work on optimal prefix free codes

In 2006, Belal *et al.* [5], described a variant of Milidiú *et al.*'s algorithm [16, 15] to compute optimal prefix free codes, announcing that it performs $O(kn)$ algebraic operations when the weights are not sorted, where k is the number of distinct code lengths in some optimal prefix free code. They describe an algorithm claimed to run in time $O(16^k n)$ when the weights are unsorted, and propose to improve the complexity to $O(kn)$ by partitioning the weights into smaller groups, each corresponding to disjoint intervals of weights[3]. The claimed complexity

[3] Those results were downgraded in the December 2010 update of their initial 2005 publication through `Arxiv` [4].

is asymptotically better than the one suggested by Huffman when $k \in o(\log n)$, and they raise the question of whether there exists an algorithm running in time $O(n \log k)$.

Like the `GDM` algorithm, the algorithm described by Belal *et al.* [5] for the unsorted case is based on several computations of the median of the weights within a given interval, in particular, in order to select the weights smaller than some well chosen value. The essential difference between both works is the use of a deferred data structure, which simplifies both the algorithm and the analysis of its complexity.

While an algorithm running in time within $O(n \lg k)$ would improve over the running time within $O(n(1+\lg\alpha))$ of our proposed solution, such an algorithm has not been defined yet, and for $\alpha < 2^k$ our solution is superior to the complexity within $O(nk)$ claimed by Belal and Elmasry [5] (and even more so over the complexity of $O(16^k n)$).

4.2 Applicability of dynamic results on Deferred Data Structures

Karp *et al.* [13] defined the first Deferred Data Structures, supporting `rank` and `select` on MULTISETS and other queries on CONVEX HULL. They left as an open problem the support of dynamic operators such as `insert` and `delete`. Ching *et al.* [7] quickly demonstrated how to add such support in good amortized time.

The dynamic addition and deletion of elements in a deferred data structure (added by Ching *et al.* [7] to Karp *et al.* [13]'s results) does not seem to have any application to the computation of optimal prefix free codes: even if the list of weights was dynamic, further work is required to build a deferred data structure supporting prefix free code queries.

4.3 Applicability of refined results on Deferred Data Structures

Karp *et al.*'s analysis [13] of the complexity of the deferred data structure is in function of the total number q of queries and operators, while Kaligosi *et al.* [12] analyzed the complexity of an offline version in function of the size of the gaps between the positions of the queries. Barbay *et al.*[2] combined the three results into a single deferred data structure for MULTISETS which supports the operators `rank` and `select` in amortized time proportional to the entropy of the distribution of the sizes of the gaps between the positions of the queries.

At first view, one could hope to generalize the refined entropy analysis (introduced by Kaligosi *et al.* [12] and applied by Barbay *et al.*[2] to the online version) of MULTISETS deferred data structures supporting `rank` and `select` to the computational complexity of optimal prefix free codes: a complexity proportional to the entropy of the distribution of codelengths in the output would nicely match the lower bound of $\Omega(k(1+\mathcal{H}(n_1,\ldots,n_h)))$ suggested by information theory, where the output contains n_i codes of length l_i, for some integer vector $(l_1,\ldots,l_h)$ of distinct codelengths and some integer h measuring the number of distinct codelengths. Our current analysis does not yield such a result: the gap lengths between queries in the list of weights are not as regular as $(l_1,\ldots,l_h)$.

4.4 Potential (lack of) Practical Impact of our Results

We expect the impact of our faster algorithm on the execution time of optimal prefix free code based techniques to be of little importance in most cases: compressing a sequence S of $|S|$ messages from an input alphabet of size n requires not only computing the code (in time $O(n(1+\lg\alpha))$ using our solution), but also computing the weights of the messages (in time $|S|$), and encoding the sequence S itself using the computed code (in time $O(|S|)$), which usually dominates the total cost. Improving the code computation time will improve on the compression time only in cases where the size n of the input alphabet is very large compared to the length $|S|$ of the compressed sequence. One such application is the compression of texts in natural language, where the input alphabet is composed of all the natural words [18]. Another potential application is the boosting technique from Ferragina *et al.* [9], which divides the input sequence into very short subsequence and computes a prefix free code for each subsequences on the input alphabet of the whole sequence.

Another argument for the potential lack of practical impact of our result is that there exist algorithms computing optimal prefix free codes in time within $O(n \lg\lg n)$ within the RAM model[4]: a time complexity within $O(n(1+\lg\alpha))$ is an improvement only for values of $\alpha \in o(\lg n)$.

4.5 Perspectives

One could hope for an algorithm with a complexity that matches the lower bound of $\Omega(k(1+\mathcal{H}(n_1,\ldots,n_h)))$ suggested by information theory, where the output contains n_i codes of length l_i, for some integer vector $(l_1,\ldots,l_h)$ of distinct codelengths and some integer h measuring the number of distinct codelengths. Our current analysis does not yield such a result: the gap lengths between queries in the list of weights are not as regular as $(l_1,\ldots,l_h)$, but a refined analysis might. Minor improvements of our results could be obtained by studying the problem in external memory, where deferred data structures have also been developed [19, 3], or when the alphabet size is larger than two, as in the original article from Huffman [11].

Another promising line of research is given by variants of the original problem, such as Optimal Bounded Length Prefix Free Codes, where the maximal length of each word of the prefix free code must be less than or equal to a parameter l, while still minimizing the entropy of the code; or such as the Order Constrained Prefix Free Codes, where the order of the words of the codes is constrained to be the same as the order of the weights. Both problems have complexity $O(n\log n)$ in the worst case over instances of fixed input size n, while having linear complexity when all the weights are within a factor of two of each other, exactly as in the original problem.

Many communication solutions use an optimal prefix free code computed offline. A logical step would be to study if any can now afford to compute a new optimal prefix free code more frequently, and see their compression performance improved by a faster prefix free code algorithm.

Acknowledgements: The author is funded by the Millennium Nucleus RC130003 "Information and Coordination in Networks". He would like to thank Peyman Afshani and Seth Pettie for interesting discussions during the author's visit to the center MADALGO in January 2014; Jouni Siren for detecting a central error in a previous version of this work; Gonzalo Navarro for suggesting the application to the boosting technique from Ferragina *et al.* [9]; Charlie Clarke, Gordon Cormack, and J. Ian Munro for helping clarify the history of van Leeuwen's algorithm [14]; Renato Cerro for various English corrections; various people who have reviewed and commented on various preliminary drafts and presentations of related work: Carlos Ochoa, Francisco Claude-Faust, Javiel Rojas, Peyman Afshani, Roberto Konow, Seth Pettie, Timothy Chan, and Travis Gagie.

[4] The algorithm proposed by van Leeuwen [14] reduces in time linear in the number of symbols of the alphabet the computation of an optimal prefix free code to their sorting, and Han [10] described how to sort a set of n integers (which input symbol frequencies are) in time within $O(n\lg\lg n)$ in the RAM model.

References

1 Jérémy Barbay. Optimal prefix free codes with partial sorting. *arXiv preprint arXiv:1602.00023*, 2016. URL: `http://arxiv.org/1602.00023`.

2 Jérémy Barbay, Ankur Gupta, Seungbum Jo, Srinivasa Rao Satti, and Jonathan Sorenson. Theory and implementation of online multiselection algorithms. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, 2013.

3 Jérémy Barbay, Ankur Gupta, S. Srinivasa Rao, and Jonathan Sorenson. Dynamic online multiselection in internal and external memory. In *Proceedings of the International Workshop on Algorithms and Computation (WALCOM)*, 2014.

4 Ahmed A. Belal and Amr Elmasry. Distribution-sensitive construction of minimum-redundancy prefix codes. *CoRR*, abs/cs/0509015, 2005. Version of Tue, 21 Dec 2010 14:22:41 GMT, with downgraded results from the ones in the conference version [5].

5 Ahmed A. Belal and Amr Elmasry. Distribution-sensitive construction of minimum-redundancy prefix codes. In Bruno Durand and Wolfgang Thomas, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3884 of *Lecture Notes in Computer Science*, pages 92–103. Springer, 2006.

6 Ahmed A. Belal and Amr Elmasry. Verification of minimum-redundancy prefix codes. *IEEE Transactions on Information Theory (TIT)*, 52(4):1399–1404, 2006.

7 Yu-Tai Ching, Kurt Mehlhorn, and Michiel H.M. Smid. Dynamic deferred data structuring. *Information Processing Letters (IPL)*, 35(1):37–40, June 1990.

8 Shimon Even and Guy Even. *Graph Algorithms, Second Edition.* Cambridge University Press, 2012.

9 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

10 Yijie Han. Deterministic sorting in $o(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50(1):96 – 105, 2004. URL: `http://www.sciencedirect.com/science/article/pii/S019667740300155X`, `doi:http://dx.doi.org/10.1016/j.jalgor.2003.09.001`.

11 David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE)*, 40(9):1098–1101, September 1952.

12 Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*, pages 103–114, 2005.

13 R. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM Journal of Computing (SJC)*, 17(5):883–902, 1988. URL: `http://dx.doi.org/10.1137/0217055`, `arXiv:http://dx.doi.org/10.1137/0217055`, `doi:10.1137/0217055`.

14 J. Van Leeuwen. On the construction of Huffman trees. In *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*, pages 382–410, Edinburgh University, 1976.

15 Ruy Luiz Milidiú, Artur Alves Pessoa, and Eduardo Sany Laber. A space-economical algorithm for minimum-redundancy coding. Technical report, Departamento de Informática, PUC-RJ, Rio de, 1998.

16 Ruy Luiz Milidiú, Artur Alves Pessoa, and Eduardo Sany Laber. Three space-economical algorithms for calculating minimum-redundancy prefix codes. *IEEE Transactions on Information Theory (TIT)*, 47(6):2185–2198, September 2001. URL: `http://dx.doi.org/10.1109/18.945242`, `doi:10.1109/18.945242`.

17 Alistair Moffat and Andrew Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory (TIT)*, pages 1650–1657, 1998.

18 E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

19 Jop F. Sibeyn. External selection. *Journal of Algorithms (JALG)*, 58(2):104–117, 2006.