University of Ljubljana
Institute of Mathematics, Physics and Mechanics
Department of Mathematics
Jadranska 19, 1000 Ljubljana, Slovenia

Preprint series, Vol. 36 (1998), 600

SUB-LINEAR DECODING OF HUFFMAN CODES ALMOST IN-PLACE

Andrej Brodnik Svante Carlsson

ISSN 1318-4865

May 7, 1998

Ljubljana, May 7, 1998

Sub-linear Decoding of Huffman Codes Almost In-Place

Andrej Brodnik* Svante Carlsson†

May 6, 1998

Abstract

We present a succinct data structure storing the Huffman encoding that permits sub-linear decoding in the number of transmitted bits. The size of the extra storage except for the storage of the symbols in the alphabet for the new data structure is $O(l \log N)$ bits, where l is the longest Huffman code and N is the number of symbols in the alphabet. We present a solution that typically decodes texts of sizes ranging from a few hundreds up to 68 000 with only one third to one fifth of the number of memory accesses of that of regular Huffman implementations. In our solution, the overhead structure where we do all but one memory access to, is never more than 342 bytes. This will with a very high probability reside in cache, which means that the actual decoding time compares even better.

1 Introduction

If you have an alphabet of N symbols that you would like to encode the typical solution would be to use $\lceil \log n \rceil$ bits to encode N different numbers, each number corresponding to a symbol. This is done, for example, in ASCII-coding. If all symbols in a text that should be coded are equally frequent this will also give a minimal size encoding in the number of bits used. The encoding of a symbol is done by a standard dictionary look-up and the decoding is done by a constant-time table look-up.

If the frequency between the symbols varies one can construct a smaller encoding of a given text. Let us arrange N symbols with normalized frequencies ν_i $(0 \leq i < N)$ at leaves of a binary tree, where l_i is the depth of the leaf. The binary tree with the minimal weighted external path length

$$\overline{l} = \min \sum_{i=0}^{N-1} \nu_i l_i \tag{1}$$

is called the *Huffman tree* (cf. [5, 13]).

A path from the root to the leaf in a binary tree can be encoded as a binary string where descending to the left (right) is represented with 0 (1) in the string. Using such a technique we can uniquely encode each symbol in the text. Moreover, these encodings are *prefix-free*. The encoding constructed using a Huffman tree, is called *Huffman encoding*.

*Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia and Department of Computer Science, Luleå University, Sweden,

†Department of Computer Science, Luleå University, Sweden

It is easy to see that the Huffman encoding, because of the property from eq. (1), gives the best possible compression of a finite text if one has to use one static code for each symbol of the alphabet. Therefore the encoding is extensively used in various fields of computer science, such as picture compression ([1]), HDTV ([4]), data transmission ([4, 5]), etc. We gain this decrease of the size of the encoded text at the cost of increased decoding time, which is linear in the number of bits of the text.

In this paper we are considering the size of the data structure to store the Huffman tree and the time necessary to decode a given Huffman code of a symbol. The best solution known so far requires $D+2N-3$ space for the data structure (D is the size of dictionary storing the symbols) and $O(l)$ time to decode the symbol ([1]). Moreover, the data structure, and hence, the accompanying algorithm, is very involved. In this paper we improve the space bound to $D + O(l \log N)$ bits for the data structure with a hidden constant at most 4 (l is the longest Huffman code); and the time bound to $\Theta(\log l')$ worst-case (l' is the number of different lengths of Huffman codes). The average decoding time for any Huffman tree is bounded by $\Theta(\log \log n)$. In fact, if the symbols have equal probabilities we have an in-place, constant time decoding solution.

The paper has a usual structure: first we do the homework by browsing through the literature searching for similar and related solutions; then we present general ideas how to change the Huffman tree not changing the cost of encoding described by the optimization function in eq. (1). In the core of the paper we present the basic structure and the decoding algorithm. This solution is further improved to the best possible on the average time-wise and almost optimal space-wise. Finally, we present some practical results of our algorithms as they are used in the *Phone Book of Slovenia*, showing a remarkable speed-up and space reduction.

NOTATION: In this paper we consider symbols with normalized frequencies ν_i, $0 < i \leq N$ and $\sum_{i=0}^{N-1} \nu_i = 1$. The length of the Huffman code of the i^{th} symbol is l_i bits, the longest code is l bits long, the number of different code lengths is l' and the weighted length of all codes is $\overline{l} = \sum_{i=0}^{N-1} \nu_i l_i$. To write down all the symbols we need D space.

2 Homework

Construction of the Huffman codes minimizes eq. (1) over all possible trees (cf. [5]). There exists a simple greedy algorithm which, from the list of symbols with frequencies, constructs Huffman codes in $O(N \log N)$ time (cf. [2, pp. 339–343]). The algorithm first sorts symbols in descending order of frequencies ($\Theta(N \log N)$ time), and then constructs the Huffman tree ($\Theta(N)$ time). The whole construction takes $O(N)$ registers of space, but can be brought down to $N + O(1)$ registers ([9]).

There are numerous variations to the basic problem. In the basic version we have as an input the list of symbols with their frequencies. However, we do not always have a possibility to construct such a list. In this case we use dynamic Huffman coding (cf. [13, 14]). Even more generalized version of the problem is when we do not have the list of symbols either (cf. [7]). Yet another version of the problem is to construct Huffman codes that must be shorter than some predefined constant (cf. [8]). The solution presented in this paper can be applied to all these different versions of the problem. However, our solution only substantially speeds up their decoding process, while (in general) it does not decrease the size of the data structure.

When the Huffman encoding is constructed appears the problem of its decoding. The decoding

algorithm has to be very fast and simple. The main reason is that it usually runs on a cheap hardware without too much memory (e.g. when used for HDTV etc.). Therefore a number of different decoding algorithms were constructed which try to minimize the size of the data structure and simultaneously decrease the decoding time (cf. [11, 4, 1, 6]). All of the mentioned algorithms try to traverse the Huffman tree constructed by the common "merging algorithm" (cf. [2, pp. 339–340]). They apply at least one of the following two techniques: they "wrap" the Huffman tree in such way that the leaves are replaced by the root of the tree; and/or they layer the tree (graph) into a small number of layers (preferably constant number) that are inspected quickly. Technically, the first technique gives a directed graph which needs to be traversed and when the root is reached, the code is decoded (cf. finite automaton); while the second technique gives an r-ary trie (graph). Applying only the first technique can give us $O(l)$ time and $D+O(N)$ space solution, while applying only the second technique can give us $O(1)$ time and $O(D + 2^l)$ space solution. Obviously, there is a wide spectrum of solutions between both techniques which, however, employ involved pointer passing data structures.

3 Mutation of the Huffman Tree and the Basic Data Structure

Vitter ([13]) states the following important so-called characterization of Huffman trees (cf. Figure 1):

Lemma 1 (Sibling Property) *A binary tree with N leaves is a Huffman tree iff:*

1. *the N leaves have nonnegative weights ν_i $(0 < i \leq N)$ and the weight of each internal node is the sum of the weights of its children; and*
2. *the nodes can be numbered by weight, so that nodes $2j + 1$ and $2j$ are* siblings *$(0 < j < N)$ and their common parent node is higher in the numbering.*

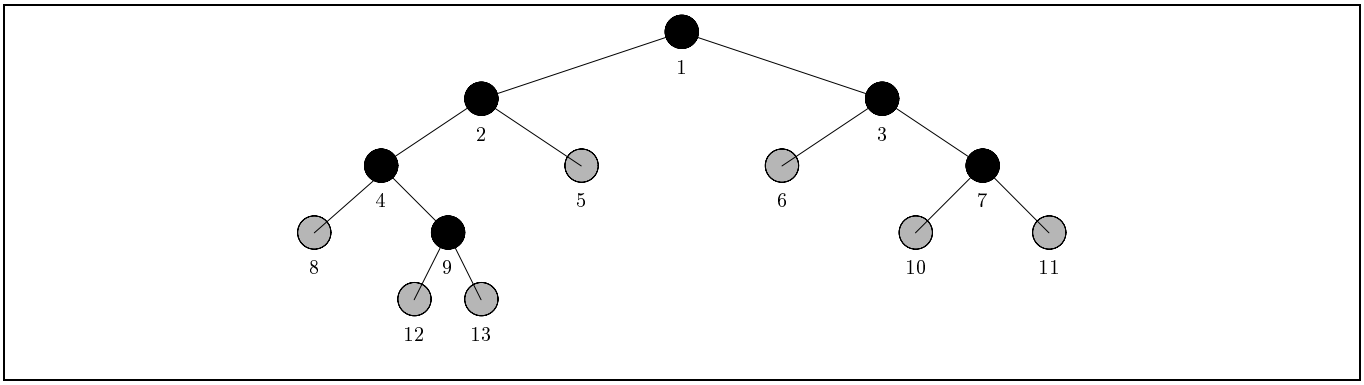

Figure 1: An example of Huffman tree with numbered nodes.

The important consequence of the Lemma 1 is:

Corollary 1 *The nodes of a Huffman tree are sorted by their weights level by level from top to bottom. This is true for all nodes and separately for internal (*black *in Figure 1) and external (*gray *in Figure 1) nodes.*

In the rest of the text we will refer to the numbers in Figure 1 as labels. In particular, the labels at leaves can be, w.l.o.g., considered symbols we are encoding.

Because of $\Omega(N \log N)$ lower bound on sorting under the comparison based model and since we can traverse nodes of a binary tree level by level in $O(N)$ time Corollary 1 gives us a trivial lower bound:

Lemma 2 *Given a list of N symbols with frequencies ν_i $(0 \leq i < N)$, it takes $\Omega(N \log N)$ to construct the Huffman tree.*

A straight forward implementation of the Huffman tree uses at least $2N-2$ pointers to represent the tree. It is easy to see that it is necessary to use $2N - o(N)$ bits to represent the shape of the tree, since the Huffman tree can take any shape with appropriate assignment of probabilities to the symbols.

To decode a symbol a bit string into the corresponding symbol takes time proportional to the number of bits in the string (l). This is optimal in the pointer machine model.

3.1 Mutation of the Huffman Tree

Since the lower bound on the size of the structure depended on the number of different shapes of Huffman trees and the decoding lower bound depended on the pointer machine model, we will restrict the number of possible shapes and allow table look-ups to beat these lower bounds.

We will refer to the internal nodes of a Huffman tree as *black* nodes, to the external nodes (leaves) as *gray* nodes and to the missing nodes (nodes that would be descendants of gray nodes) as *white* nodes.

The crucial step in the construction of the minimal size Huffman tree is the following rearrangement producing the *mutated Huffman tree* (cf. Figure 2):

Definition 1 *We rearrange (permute) the nodes on the level i $(0 \leq i < l)$ of the Huffman tree so:*

1. *that first (on the left) are b_i black nodes, then g_i gray nodes and on the far end are w_i white nodes*[1]*; and*

2. *that the relative order of the same coloured nodes does not change.*

A special case of the mutated Huffman tree is a left-aligned Huffman tree.

During the process of node rearrangement the depth of a leaf does not change and thus the length of the code does not change either. Therefore, the code generated from the mutated tree remains optimal. Moreover, Corollary 1 is valid for the mutated tree as well. In the rest of the paper we will refer to the Huffman code as the code produced by using the mutated Huffman tree.

3.2 Basic Data Structure

Before giving the representation of the basic data structure and the decoding algorithm we need the following lemmata:

[1] Obviously, $b_i + g_i + w_i = 2^i$.

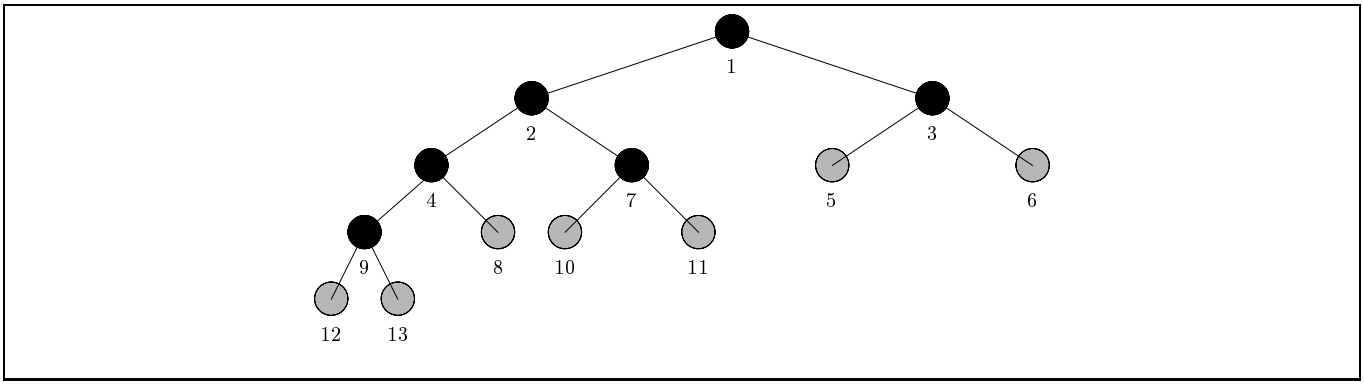

Figure 2: The mutated Huffman tree of Figure 1.

Lemma 3 *The symbol stored at the g^{th} gray node ($0 \leq g < g_i$) on the level l_i of the mutated Huffman tree has the Huffman code corresponding to the binary representation using l_i bits of the number $b_i + g$.*

Proof: Cutting the mutated Huffman tree at the level l_i gives a complete binary tree with b_i black leaves, g_i gray leaves and w_i white leaves; in particular the g^{th} gray leaf is the $b_i + g$-th leaf of the cut tree. Obviously, the binary representation, using l_i bits, of the number $b_i + g$ represents exactly the path from the root to the leaf. $\mathcal{QED}$

Lemma 4 *Let c be a l_i bits long bit-string. If*

$c < b_i$: c is a prefix of some Huffman code;

$b_i \leq c < b_i + g_i$: c is a Huffman code; and

$b_i + g_i \leq c$: c contains as a proper prefix some Huffman code.

Proof: The second case follows trivially from Lemma 3, while the first and the last from the definition of the mutated Huffman tree (see Definition 1). $\mathcal{QED}$

Now we are ready to show how we can get below the previous space lower bound by using the mutated Huffman tree.

Theorem 1 *There exists a data structure that uses only $O(l \log N)$ bits, except what is necessary to store the alphabet, that permits $O(l)$ worst case and $O(\bar{l})$ average case decoding time.*

Proof: By Corollary 1 the leaves of the mutated Huffman tree are sorted by non-decreasing weight from the bottom to the top. This brings us to the following two-part data structure:

- sorted list of symbols `table` (size D); and
- an index of l pairs $(b_i, \texttt{indx}_i)$, where $\texttt{indx}_i$ is the index of the first leaf on level l_i in the `table` (size $O(l \log N)$).

The decoding algorithm reads (shifts) in bit at a time until the number `code` represented by the l_i-bit shifted in bit-string falls in the range $b_i \leq \texttt{code} < b_i + g_i$. The worst case running time of the algorithm is obviously $O(l)$, while its correctness follows directly from Lemma 4. The average

case running time is proportional to the sum of $\nu_i \cdot l_i$ over all symbols – hence by eq. (1) it is $O(\bar{l})$. $\mathcal{QED}$

Note, that sortedness of symbols is not the precondition. It is sufficient that leaves from the given level of the tree are stored in consecutive locations of the `table`.

An important observation for this, and all our new algorithms, is that it makes only one reference to a symbol stored in the `table` when it is decoded. The rest of the time is spent in a very small table with indices that will almost always be in the cache.

Most Huffman trees do not have leaves on all levels, but only on a few. An immediate improvement of the algorithm in the previous theorem is not to search linearly through all levels, but only through those l' levels that have leaves. By keeping track of the number of levels to skip (bits to shift in) in the small index table we get solution presented in Algorithm 1. Hence:

Corollary 2 *Algorithm 1 uses data structure of size $D+O(l' \log N)$ bits and decodes Huffman code in $O(l')$ worst-case time.*

The average time for decoding using Algorithm 1 depends very much on the probability distribution, but it is at least never bigger than $\bar{l}$.

```
typedef struct {
  unsigned short skipBits;
  tNumNodes blacks;
  tNumNodes short grays;
} tIndx;
static const tIndx indx[]= ...;
tNumNodes Decode () {
  tNumNodes indxOfSymbol= 0;
  tNumNodes code= 0;
  tNumLevels i= 0;
  while (indx[i].blacks < code) {
    indxOfSymbol+= indx[i].grays;
    ShiftIn(indx[i].skipBits, code);
    i++;
  };
  return indxOfSymbol + (code - indx[i].blacks);
} /* Decode */
```

Algorithm 1: Linear decoding algorithm.

4 Faster Decoding

In Algorithm 1 we search for the level of the symbol linearly from the top and in this section we improve the solution by using binary search on the levels of the mutated Huffman tree (cf. [3]) – for a toy example see Figure 3. The search, when probing on some level, shifts in or out the appropriate number of bits from the input stream and checks if the computed code corresponds to the black, gray or white node. If the corresponding node is:

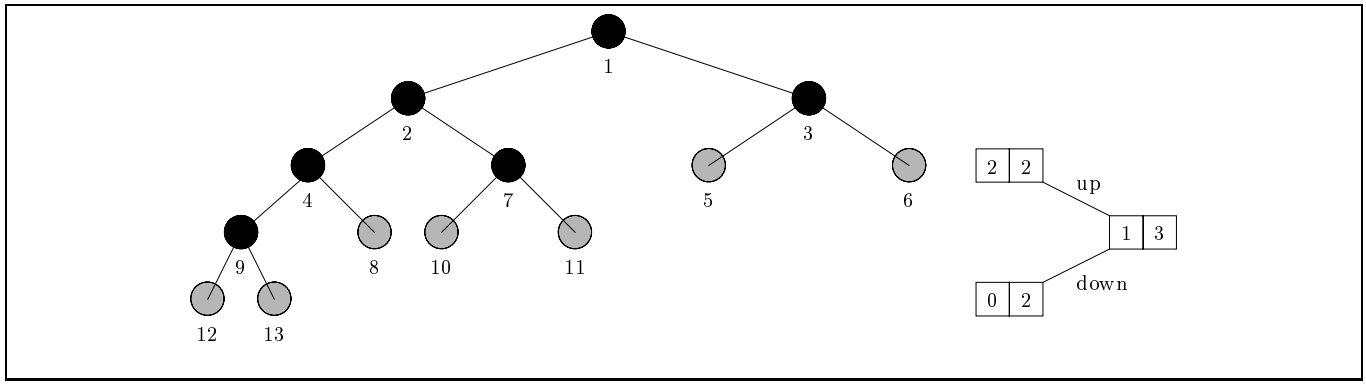

Figure 3: An example of a binary search tree on the levels of the mutated Huffman tree.

- black, we are too high in the tree and we have to go down;
- gray, we found the symbol; and
- white, we are too low in the tree and we have to go up.

Hence:

Theorem 2 *There exists a data structure of size* $D + O(l' \log N)$ *bits that permits decoding of the Huffman code in* $O(\log l')$ *time.*

By blindly using the logarithmic search on the levels we can actually get a higher average search cost than in the linear search. This will be the case if, for instance, the frequencies are exponentially decreasing. Instead, we should build an optimal binary search tree on those levels of the Huffman tree that contain leaves. A level is weighted by the sum of frequencies of leaves on the given level.

We want to show that the average decoding time for any Huffman tree is bounded by $O(\log \log N)$ if we use the optimal search tree for searching the levels. To do this we first have to prove some properties of the Huffman tree.

Lemma 5 *The total weight of a non-leaf subtree on level* i *of a Huffman tree is at most* $(2/3)^i$. *The weight of a leaf on level* i *is at most* $(2/3)^{i-2} \cdot (1/2)$.

Proof: Consider an internal node that is a child of the root of the Huffman tree. If the total weight of that subtree is more than 2/3 then its sibling must have a weight that is less than 1/3 and one of its children has a weight that is more than 1/3. This contradicts Corollary 1 since a node on a higher level has a lower weight than a node on a lower level.

A grand-child of a node can never be a weight higher than 1/2 of the total weight, since it then would have to be the child of that node.

By using the same technique in an induction we get the stated results. $\mathcal{QED}$

Lemma 6 *A Huffman tree with* N *symbols can only have* $O(\log N)$ *levels where the total weights of the leaves on each of these levels exceeds* $1/N^2$.

Proof: From Lemma 5 we know that the topmost $2 \log_{3/2} N$ levels can each have only one leaf and still have a total weight bigger than $1/N^2$.

On the following levels we know that on level $(2\log_{3/2} N) + i$ there has to be at least $(3/2)^i$ leaves to make the total weights of the leaves at least $1/N^2$. This follows almost directly from the previous Lemma.

Since there are only N leaves in total these can only help make $\log_{3/2} N$ levels below level $2\log_{3/2} N$ having a total weight of at least $1/N^2$. In total, there could only be a logarithmic number of levels with that high weight. $\mathcal{QED}$

Using this observation, we can show that the following theorem:

Theorem 3 *There exists a data structure of size $D + O(l' \log N)$ bits that permits decoding of the Huffman code in $O(\log \log N)$ average time for any probability distribution.*

Proof: Consider the following binary search of levels: Construct a perfectly balanced binary search tree of the levels with a weight that is at least $1/N^2$. The rest of the levels are now inserted one by one in the balanced search tree by a standard insertion method.

From Lemma 6 we know that there could only be a logarithmic number of levels in the top part of the search tree, and each of them can be found in $O(\log \log N)$ time.

There could be a linear number of levels in the small part, and each of them could take at most linear time, but since the probability for each of these levels is less than $1/N^2$ they will only contribute with a constant additive term to the expected search time.

We have now showed that there always exists one binary search tree with a search time that is $O(\log \log N)$. We know that the optimal binary search tree has the lowest expected search time it is at least as good as the one we have described. This concludes the proof. $\mathcal{QED}$

It is worth mentioning that this algorithm improves the worst average decoding time from $O(\log N)$ for the standard Huffman decodings. We are able to break this lower bound because we are able to use table look-ups instead of only following pointers.

The Algorithm 2 presents a reasonably practical decoding algorithm. As shown in the algorithm the constant in the size of the data structure is at most 6. Also, for the practical purposes the algorithm should be coded with `goto` statements which eliminates one conditional branching and even further speeds up the decoding.

5 Tree Construction

In all Huffman tree construction algorithms, the first step is to sort the symbols. This takes $O(N \log N)$ time.

The standard algorithm proceeds as a simple greedy algorithm ([2, pp. 339–340]). We use initially similar algorithm though augmented with an auxiliary structure recording the first node on each level and connecting all nodes in a linked list instead of building a binary tree. In the second phase we walk through all levels, count black and gray nodes, and rearrange them according to Definition 1. This takes $O(N)$ time.

If we choose to use Algorithm 2 instead of Algorithm 1 in a previous step we also compute the total weight of leaves on each level not changing the run time. However, we need to add one more step – construction of the optimal binary tree. For this we use dynamic programming, which takes $O(l'^2 \log l')$ time. In the worst, and very pathological, case, l can be as large as N.

The described procedure is used for constructing and mutating of a static Huffman tree. When considering a dynamic Huffman tree we use the structure presented by Vitter in [13, 14] and mutate

```
typedef struct {
  tCodeLength shiftBits;
  tNumNodes blacks;
  tNumNodes grays;
  tCodeLength upper;
  tCodeLength lower;
  tNumNodes firstOnLevel;
} tIndx;
static const tIndx indx[]= ...;
static const tCodeLength startAt= ...;
tNumNodes Decode () {
  tCodeLength code= 0;
  bool goUp= false;
  tCodeLength i= startAt;
  int gray;
  for (;;) {
    if (goUp) ShiftOutBits(indx[i].shiftBits, code);
    else      ShiftInBits(indx[i].shiftBits, code);
    gray= code - indx[i].blacks;
    if (gray < 0)                  { i= indx[i].lower; goUp= false; }
    else if (gray < indx[i].grays) return indx[i].firstOnLevel + gray;
    else                           { i= indx[i].upper; goUp= true; };
  };
} /* Decode */
```

Algorithm 2: Logarithmic decoding algorithm

the tree when the change occurs. The changes are always only affecting neighbouring levels ([13, Lemma 4.2]) and since we do not need keep the nodes on the level sorted, the changes in the mutated tree are easier to make and much more seldom. A similar approach is also used with a Huffman encoding of the infinite symbol alphabet (cf. [7]).

6 Practical Results

We compare three implementations of Huffman encoding: classical pointer based, linear with skips from the Algorithm 1 and the logarithmic one from the Algorithm 2. In order to put all considered implementations on the common denominator we assume that the Huffman tree is stored in two pieces: the *dictionary* containing all (variable length) symbols, and the *shape* data structure storing the Huffman tree shape (cf. Theorem 1). For each implementation we compute the size of the second piece – the shape (the dictionary is the same for all of them), and the number of comparisons performed by the decoding algorithm. Because of the two-piece data structure, all algorithms have to make an extra access to get the symbol from the dictionary.

In the pointer based implementation we assume usual representation of a binary tree (cf. [2, pp. 244–251]): the tree node contains a flag, defining if the node is a leaf or not, and one or two pointers. The decoding is essentially the same as TREE-SEARCH algorithm on [2, p. 247] – i.e. it

performs two comparisons per level.

The data used to compare the implementations are taken from the *Phone Book of Slovenia* ([12]). We took 21 different sets of data (texts) containing different numbers of symbols, from a few to more than sixty thousand, and different frequencies of these symbols. For each of the sets we computed the size of shapes for all compared implementations and the number of comparisons necessary to decode the complete text.

The Table 1 contains results of tests: the first column represents the number of symbols N, the following four columns the normalized sizes of the data structure and the last four columns the normalized number of comparisons for the decoding algorithms. The logarithmic implementation columns contain two values: the normalized, with value 100, and the actual size (number of comparisons). The actual size is measured in the number of bytes. All normalized values are rounded.

N	size				num. of comparisons			
	pointers	linear	logarithmic		pointers	linear	logarithmic	
3	117	56	100	36	238	84	100	979200
70	907	56	100	108	510	103	100	935
118	4589	56	100	36	1367	177	100	128
346	1583	56	100	306	272	89	100	5667921
426	2071	56	100	288	260	88	100	514256
731	7107	56	100	144	541	170	100	5884
741	5239	56	100	198	357	116	100	84654
846	6580	56	100	180	388	134	100	16861
890	8653	56	100	144	539	169	100	5369
2815	14596	56	100	270	478	98	100	2159426
4902	31772	56	100	216	560	184	100	116714
6374	33050	56	100	270	341	117	100	1224944
7885	40885	56	100	270	524	164	100	826258
8218	37599	56	100	306	251	91	100	1704831
8956	53583	56	100	234	462	171	100	116833
11025	61250	56	100	252	545	148	100	2299377
13881	67477	56	100	288	415	149	100	1000010
15667	71679	56	100	306	370	110	100	4323033
33134	161068	56	100	288	464	162	100	1812684
41830	232389	56	100	252	549	146	100	2586956
68618	280892	56	100	342	495	162	100	17737377

Table 1: Comparison of different implementations of the Huffman encoding.

7 Conclusion

We have presented a way to mutate a Huffman tree to have a more succinct representation of the tree and also to get a considerable speed up of the decoding compared to previous versions of Huffman trees. We typically use only a few hundred bytes extra storage for a 68 thousand

symbols alphabet as an overhead structure, and we use only one third to one fifth of the number of comparisons compared to other versions. As we hope to be able to show in the final version of this paper the actual time for decoding compares even more favorably to our algorithms.

Also the theoretical results of this paper is worth noticing. The Huffman code has been around for more than 45 years, and now we decrease the decoding time from $O(\log N)$ to $O(\log \log N)$ by using a much smaller representation than before.

We have not found any drawbacks of our algorithms, and we would strongly recommend the usage of them, or variations thereof, in all places where Huffman coding is used.

There remains an open problem if we can further (theoretically) decrease the size of the data structure. In our current approach we essentially group N keys into l subsets. In general we have $\left\lceil \binom{N}{l} \right\rceil$ such groupings and thus we need $\lg \left\lceil \binom{N}{l} \right\rceil = O(l \log N)$ bits for the description. In our case not all groupings are valid and hence this bound is too high. But how much? Note, that the size of our data structure matches the $O(l \log N)$ bound.

Addendum

When we finished the research we learned that more or less the same approach was taken by Moffat and Turpin in [10]. They give also empirical data to support their approach but do not perform theoretical analysis of the technique.

In meantime we also implemented and tested our data structure. The results of tests and their analysis from the algorithmic engineering point of view will bi subject of the full version of this contribution.

References

[1] K.-L. Chung. Efficient Huffman decoding. *Information Processing Letters*, 61(2):97–99, 1996.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[3] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1977.

[4] R. Hashemian. Memory efficient and high-speed search Huffman coding. *IEEE Transactions on Communications*, 43(10):2576–2581, October 1995.

[5] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(10):1098–1101, September 1952.

[6] V. Iyengar and K. Chakrabarty. An efficient finite-state machine implementation of Huffman decoders. *Information Processing Letters*, 64(6):271–275, 1997.

[7] A. Kato, T.S. Han, and H. Nagaoka. Huffman coding with an infinite alphabet. *IEEE Transactions on Information Theory*, 42(3):977–984, May 1996.

[8] L.L. Larmore and D.S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, July 1990.

[9] A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In *Proceedings 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 393–402. Springer-Verlag, 1995.

[10] A. Moffat and A. Turpin. On implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

[11] H. Tanaka. Data structure of Huffman codes and its application to efficient encoding and decoding. *IEEE Transactions on Information Theory*, 33(1):154–156, January 1987.

[12] Telekom Slovenije & ADACTA. Telefonski imenik Slovenije. URL: `http://tis.telekom.si`.

[13] J.S. Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM*, 34(4):825–845, October 1987.

[14] J.S. Vitter. Algorithm 673, Dynamic Huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, June 1989.