# Fast decoding of prefix encoded texts

## EXTENDED ABSTRACT

*Eyal Bergman* and *Shmuel T. Klein*
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
{bergmae1,tomi}@cs.biu.ac.il
Tel: ++(972–3) 531 8865 Fax: ++(972–3) 736 0498

**Abstract:** New variants of partial decoding tables are presented that can be used to accelerate the decoding of texts compressed by any prefix code, such as Huffman's. They are motivated by a variety of tradeoffs between decompression speed and required auxiliary space, and apply to any shape of the tree, not only the canonical one. Performance is evaluated both analytically and by experiments, showing that the necessary tables can be reduced drastically, with hardly any loss in performance.

## 1. Introduction

Huffman coding is still one of the most popular compression methods, both as a stand-alone technique, as well as a part or in combination with other methods, like `gzip` or `JPEG`. The standard decoding method for general Huffman encoded texts uses a Huffman tree, which is repeatedly traversed from the root to one of its leaves, as guided by the bit sequence of the compressed file. These bit manipulations are very time consuming, which may be critical in many on-line applications, for which decoding must be very fast. Several methods have been developed to accelerate the decoding, based on various data structures. These, on the other hand, implied a new problem, that of available internal memory, as for faster decoding, larger amounts of RAM were necessary.

In fact, one may consider three competing criteria according to which various decoding methods should be judged. The first is *compression efficiency*: this sounds trivial since Huffman codes are optimal, at least once the model of what exactly is to be encoded is fixed, and thus the set of probabilities is given. But our discussion is relevant also to the more general set of prefix codes, even if they are not optimal. Moreover, to achieve improvements regarding the other criteria, it might sometimes be justified to replace Huffman codes by non optimal alternatives. The second criterion is *time complexity*, which will be measured by the average number of bits that can be decoded in a single operation. The third criterion is the amount of *internal memory* required to store the specific data structures upon which each decoding method is built.

These criteria tend in opposite directions, and improving on one comes generally at the expense on worsening at least on one of the others, if not on both. The present work extends previous suggestions by seeking a reasonable tradeoff. In the next section, we review some of the background and present a new method in Section 3. All the methods are then compared in an experimental section at the end.

## 2. Previous work

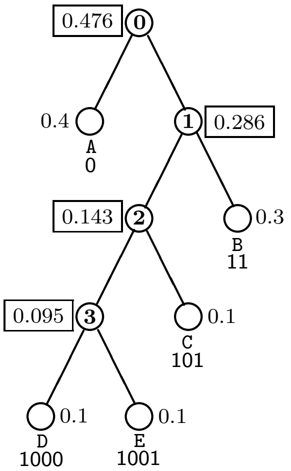
The main tool for improving the decoding time is devising a method that allows the processing of several bits in every iteration. A simple way of achieving this goal is to use higher order Huffman codes instead of the standard binary ones. If $2^k$-ary codes are used, then each "character" can be replaced by one of the possible $k$-bit strings, so the encoded file may be processed by blocks of $k$ bits at a time. However, there might be a serious loss in compressibility for larger $k$. The optimal binary codes can be kept if special codes are considered or additional data structures are used for the blockwise decoding.

One of the special codes that are often suggested in this connection are *canonical* Huffman codes [10]. They are based on using Huffman's algorithm only to evaluate the codeword lengths, but then assigning the actual codewords systematically so that ordering them lexicographically also arranges them by non-decreasing length. This is exploited to fast decoding by a sequence of cascading comparisons in [6], and by the use of skeleton trees in [5]. Canonical codes can be built for the codeword lengths of the optimal Huffman codes, so there is *a priori* no reason to use non-canonical codes, but in certain applications, other Huffman codes can be justified, for example when the coding serves not only for compression, but also for encryption, or when synchronizing codewords or sequences are of concern [2].

Efficient decoding of $k$ bits in every iteration is made possible by using a set of $m$ auxiliary tables, which are prepared in advance for every given prefix code [1]. A similar method appears also in [9], and various variants have been suggested more recently in [4, 7, 3, 8].

The basic scheme is as follows. The number of entries in each table is $2^k$, corresponding to the $2^k$ possible values of the $k$-bit patterns. Each entry is of the form $(W, j)$, where $W$ is a sequence of characters and $j$ ($0 \le j < m$) is the index of the next table to be used. The idea is that entry $i$, $0 \le i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence $W$ of characters from the $k$-bit block representing the integer $i$ ($W$ may be empty when there are codewords of more than $k$ bits); usually some of the last bits of the block will not be decipherable, being the prefix $P$ of more than one codeword; $j$ will then be the index of the table corresponding to that prefix (if $P = \Lambda$, where $\Lambda$ denotes the empty string, then $j = 0$). Table number $j$ is constructed in a similar way except for the fact that entry $i$ will contain the analysis of the bit pattern formed by the prefixing of $P$ to the binary representation of $i$. We thus need a table for every possible proper prefix of the given codewords; the number of these prefixes is obviously equal to the number of internal nodes of the appropriate Huffman-tree (the root corresponding to the empty string and the leaves corresponding to the codewords), so that $m = N - 1$, where $N$ is the size of the alphabet.

More formally, let $P_j$, $0 \le j < N-1$, be an enumeration of all the proper prefixes of the codewords (no special relationship needs to exist between $j$ and $P_j$, except for the fact that $P_0 = \Lambda$). In table $j$ corresponding to $P_j$, the $i$-th entry, $T(j,i)$, is defined as follows: let $B$ be the bit-string composed of the juxtaposition of $P_j$ to the left of the $k$-bit binary representation of $i$. Let $W$ be the (possibly empty) longest sequence of characters that can be decoded from $B$, and $P_\ell$ the remaining undecipherable bits of $B$; then $T(j,i) = (W, \ell)$.




| Entry | Pattern for Table 0 | Table 0 $W$ | Table 0 $\ell$ | Table 1 $W$ | Table 1 $\ell$ | Table 2 $W$ | Table 2 $\ell$ | Table 3 $W$ | Table 3 $\ell$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 000 | AAA | 0 | D | 0 | DA | 0 | DAA | 0 |
| 1 | 001 | AA | 1 | E | 0 | D | 1 | DA | 1 |
| 2 | 010 | A | 2 | CA | 0 | EA | 0 | D | 2 |
| 3 | 011 | AB | 0 | C | 1 | E | 1 | DB | 0 |
| 4 | 100 | — | 3 | BAA | 0 | CAA | 0 | EAA | 0 |
| 5 | 101 | C | 0 | BA | 1 | CA | 1 | EA | 1 |
| 6 | 110 | BA | 0 | B | 2 | C | 2 | E | 2 |
| 7 | 111 | B | 1 | BB | 0 | CB | 0 | EB | 0 |

FIGURE 1: *Huffman tree and partial decoding tables*

As an example, consider the alphabet {A,B,C,D,E}, with codewords {0, 11, 101, 1000, 1001} respectively, and choose $k = 3$. There are 4 possible proper prefixes: $\Lambda$,1,10,100, hence 4 corresponding tables indexed 0,1,2,3 respectively, and these are given in Figure 1, along with the corresponding Huffman tree that has its internal nodes numbered accordingly. The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1, 2 and 3 are obtained by prefixing '1', '10' or '100', respectively, to the strings in 'Pattern'. If the encoded text, which serves as input string to this decoding routine, consist of 100 101 110 000 101, we access sequentially Table 0 at entry 4, Table 3 at entry 5, Table 1 at entry 6, Table 2 at entry 0 and Table 0 at entry 5, yielding the output strings DA B DA C.

The general decoding routine is thus extremely simple. Let $M[f;t]$ denote the substring of the encoded string serving as input stream to the decoding, that starts at bit number $f$ and extends up to and including bit number $t$; let $j$ be the index of the currently used table and $T(j,\ell)$ the $\ell$-th entry of table $j$:

```
j ← 0
for   f ← 1   to   length of input   do
      (output, j) ← T(j, M[f; f + k − 1])
      f ← f + k
```

The larger is $k$, the greater is the number of characters that can be decoded in a single iteration, thus transferring a substantial part of the decoding time to the preprocessing stage. The size of the tables, however, is $\Omega(N2^k)$, so it grows exponentially with $k$, and may become prohibitive for large alphabets and even moderately large

$k$. For example, if $N = 30000$ and $k$ is chosen as 16 and every table entry requires 6 bytes, the tables, which should be stored in RAM, would need about 11GB!

Nevertheless, large alphabets are not rare in data compression. While Huffman's classical algorithm is often explained on the basis of encoding individual characters, it applies in fact as well to the encoding of any set of well-defined items into which the file to be compressed can be unambiguously parsed, such as bigrams or even words. In fact, for the compression of the textual databases of large Information Retrieval Systems (IRS), the "alphabet" is often defined as the set of different words [10]: Huffman coding can then yield performances that are close to that of the best competing methods, and the fact that a huge Huffman tree has to be kept is not a concern, since the alphabet is stored anyway as the *dictionary* of the IRS, and the tree structure can be encoded very efficiently. This motivates the search for tradeoffs, which are dealt with in the next section.

# 3. Designing new tradeoffs

One of the main goals of [1] was the complete avoidance of any bit-manipulations, so as to facilitate the decoding in any high-level language. If one relaxes this constraint, and agrees to decode certain bits more than once, the number of tables and their sizes can be reduced.

## 3.1 Reduced partial decoding tables

Particularly in the case of a large alphabet, the blocksize $k$ could be chosen smaller than the longest codeword, and tables would be constructed not for all the internal nodes, but only for those on levels that are multiples of $k$, that is for the root (level 0), and all the internal nodes on levels $k$, $2k$, $3k$, etc. There is an obvious gain in the number of tables, which comes at the price of a slower decoding pace: as before, the table entries consist first of the decoding $W$ of a bit string $B$ obtained by concatenating some prefix to the binary representation of the entry index. If $B$ is not completely decipherable, the remainder $P_j$ is used in the previous setting as index to the next table. For the new variant, if $|P_j|$, the length of the remainder, is smaller than $k$, then no corresponding table has been stored, so these $|P_j|$ bits have to be reread in the next iteration. We shall refer to this variant as the *reduced (partial decoding) tables*.

The table entries are thus extended to include a third component: a back skip $b$, indicating how many bits should the pointer into the input string be moved back. Using the above notations, $T(j, i)$ will consist of the triplet $(W, \ell, b)$, and the decoding routine is given by

$j \leftarrow 0 \qquad back \leftarrow 0$
for $\quad f \leftarrow 1 \quad$ to $\quad$ length of input $\quad$ do
$\qquad (\text{output}, j, back) \leftarrow T(j, M[f; f+k-1])$
$\qquad f \leftarrow f + k - back$

| Entry | Pattern for Table 0 | Table 0 | | | Table 3 | | |
|---|---|---|---|---|---|---|---|
| | | $W$ | $\ell$ | $b$ | $W$ | $\ell$ | $b$ |
| 0 | 000 | `AAA` | 0 | 0 | `DAA` | 0 | 0 |
| 1 | 001 | `AA` | 0 | 1 | `DA` | 0 | 1 |
| 2 | 010 | `A` | 0 | 2 | `D` | 0 | 2 |
| 3 | 011 | `AB` | 0 | 0 | `DB` | 0 | 0 |
| 4 | 100 | — | 3 | 0 | `EAA` | 0 | 0 |
| 5 | 101 | `C` | 0 | 0 | `EA` | 0 | 1 |
| 6 | 110 | `BA` | 0 | 0 | `E` | 0 | 2 |
| 7 | 111 | `B` | 0 | 1 | `EB` | 0 | 0 |

FIGURE 2: *Reduced partial decoding tables*

As example, consider the same Huffman tree and the same input string as above with $k = 3$. Only two tables remain, Table 0 and Table 3, given in Figure 2. Decoding is now performed by six table accesses rather than only 5 with the original tables, using the sequence of blocks 100, 101, **1**11, **1**00, 001, **1**01 to access tables 0, 3, 0, 0, 3, 0, respectively, where bits read twice are bold faced.

To get an estimate of the average number $b$ of bits that have to be processed twice in each block, or equivalently, to get the average number of bits $k - b$ processed by a single table access with blocksize $k$, we introduce the following notations. Let $\mathcal{T}$ denote the Huffman tree corresponding to a given Huffman code. The elements which are encoded appear with probabilities $p_1, \ldots, p_N$ in the text, and the lengths of the corresponding Huffman codewords are $\ell_1, \ldots, \ell_N$, respectively. We shall also use the notation $p_y$ for the probability of the element corresponding to the leaf $y$. Denote by $\mathcal{L}$ the set of the leaves of $\mathcal{T}$, and by $\mathcal{I}$ the set of its internal nodes. For each $x \in \mathcal{I}$, we define $\mathcal{T}_x$ as the subtree of $\mathcal{T}$ rooted at $x$, and we denote by $\mathcal{L}_x = \mathcal{L} \cap \mathcal{T}_x$ the set of its leaves. We further denote by $\ell(x)$ the depth in the tree of the internal node $x$, i.e., the length of the corresponding prefix. The internal nodes $\mathcal{I}$ correspond to the positions at which a codeword might be cut by a $k$-bit block boundary. In particular, the root $r$ of the tree, which belongs to $\mathcal{I}$, corresponds to the special case where the block boundary falls between two codewords, i.e., there was an empty remainder in the decoding of the binary string indexing this entry.

In a first stage, it seems that we may assume that a block boundary occurs at random in any possible position, that is, at any internal node of $\mathcal{T}$. This is an approximation, since in certain cases, not all the positions are possible cut-points, nor do those that are possible all appear with the same probability. For example, if both the block-size and all the codeword lengths are even, then no codeword can be cut by a block boundary after an odd number of bits. But for many real-life distributions, especially for the large ones with thousands or even millions of elements, the corresponding Huffman codes have codewords of all possible lengths in a certain range, so we conclude that our assumption can be justified. However, due to the backskips of the Reduced Tables algorithm, a block boundary can in fact only be at an internal node $x$, called below a *permissible* node, for which $\ell(x) < k$ or $\ell(x) \bmod k = 0$. For suppose $\ell(x) = k + j$ with $0 < j < k$. Since the block size is $k$, this means that the previous block started at an internal node on level $j+1$, contradicting the fact that in the algorithm all blocks start at levels that are multiples of $k$, plus 1. Denote the set of permissible internal nodes by $\mathcal{P}$.

Consider then the fact of having a block boundary in a certain position as if it were generated by the following random process: the compressed text consisting of a given sequence of concatenated codewords, we "throw" at random boundaries into this string, that is, we pick randomly bit positions among the permissible ones, which shall act as the ending positions of the blocks. In this sense, we can speak about the probability of having a block boundary in a certain position. For a given permissible internal node $x \in \mathcal{P}$, we evaluate the probability $P(x)$ of the position corresponding to $x$ being picked as a boundary point as follows.

Each leaf of the Huffman tree is associated with a probability $p_i$, and the probability associated with an internal node $y$ is the sum of the probabilities associated with the two children of $y$. Adding the probabilities associated with all the permissible internal nodes, we get $W = \sum_{z \in \mathcal{P}} p_z$, and the probability $P(x)$ is given by

$$P(x) = \frac{\sum_{y \in \mathcal{L}_x} p_y}{W}.$$

This is indeed a probability distribution, as $\sum_{x \in \mathcal{P}} P(x) = 1$. In the particular case where $k$ is at least the depth of the tree minus 1, all the internal nodes are permissible, and we get $W = \sum_i^N p_i \ell_i$ is the average codeword length. Returning to our running example, the tree in Figure 1 has next to its leaves an assumed probability distribution, with `A`, `B`, `C`, `D` and `E` appearing with probabilities 0.4, 0.3, 0.1, 0.1 and 0.1, respectively; the corresponding values for $P(x)$, assuming $k \geq 3$, appear in boxes next to the internal nodes.

If the last bit in a $k$-bit block corresponds to $x \in \mathcal{P}$, such that $\ell(x)$ is a multiple of $k$, then there is no remainder in the partial decoding, and the variable *back* in the above algorithm will be zero. If $\ell(x)$ is not a multiple of $k$, then $\ell(x) < k$ and the number of bits to be moved back is $\ell(x)$. Averaging over all the possible positions of the block boundary, we get as estimate for $b$:

$$E(b) = \sum_{x \in \mathcal{P}} P(x)\, \ell(x) = \frac{1}{W} \sum_{x \in \mathcal{P}} \left( \ell(x) \sum_{y \in \mathcal{L}_x} p_y \right). \qquad (1)$$

For our example tree, if $k$ is chosen as 2, 3 or 4, the estimated values of $k - b$ would be 1.83, 2.43 and 3.14, respectively.

### 3.2 Bounded reduced partial decoding tables

More space can be saved if variable values of $k$ can be used. Tables corresponding to internal nodes that are roots of subtrees of depth less than $k$ may contain several copies of the same information. For instance, table 3 in the last example corresponds to a tree of depth 1. Therefore the $k-1$ rightmost bits of the $k$-bit block are decoded twice: the upper and lower halves of table 3 are identical, except for the first character in $W$ which is D in the upper and E in the lower part. This wasted space can be saved if we adapt the blocksize $k$ to each node.

Each of the prefixes $P_j$ corresponds to one of the internal nodes $v_j$, denote by $d(j)$ the depth of the subtree rooted at $v_j$. Table $j$ will contain only $2^{d(j)}$ entries and a new auxiliary vector $k[j]$ will be used, defined by

$$k[j] \leftarrow \min(k, d(j)).$$

The above algorithm is still valid after having replaced the two occurrences of $k$ by $k[j]$. This variant will be referred to as the *bounded (reduced partial decoding) tables*.

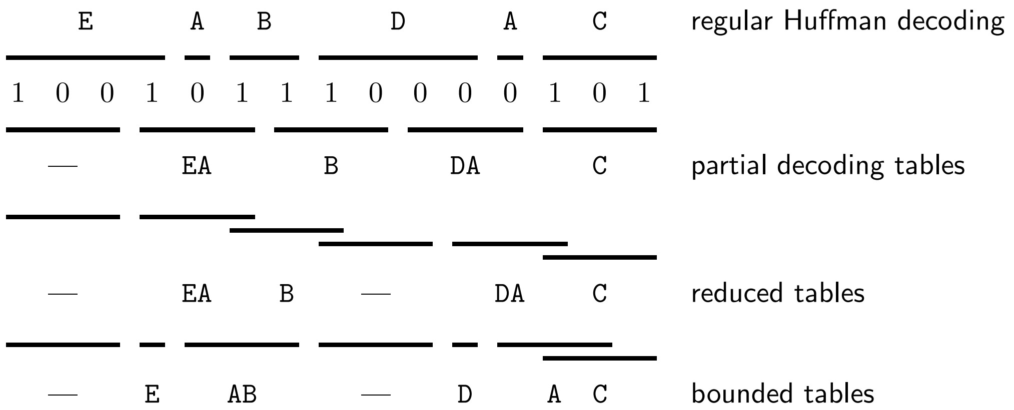



FIGURE 3: *Example using original, reduced and bounded partial decoding tables*

Figure 3 shows the input string and above it its parsing into codewords. Below appear first the parsing into consecutive $k$-bit blocks using the original tables, then the parsing into partially overlapping $k$-bit blocks with the reduced tables, finally the parsing into variable length block using the bounded tables. Note that for simplicity, we do not deal in this abstract with the case that the last block may be shorter than $k$ bits.

### 3.3 Weighted reduced partial decoding tables

The last variant may further be improved, because letting the blocksize $k[j]$ depend only on the depth of the subtree rooted at the current internal node $v_j$ does not take the shape of this tree into account. This suggest the following family of tradeoffs that can be controlled by a factor $\alpha$ according to the available space and time resources.

Set a parameter $0 \leq \alpha \leq 1$, where $\alpha = 1$ will correspond to the standard Huffman decoding. The smaller $\alpha$, the more RAM is needed for the decoding tables, but on the other hand, decoding will be faster as more bits are decoded simultaneously. The extreme case $\alpha = 0$ corresponds to having each codeword decoded by a single table access, but requires a table of size $O(2^{max})$, where $max$ is the maximal length of a codeword. For a given (binary) Huffman tree, define $n_i$ as the number of its leaves on level $i$, and $m_i$ as the number of "unused" leaves on level $i$, if we would add more nodes so as to transform the upper $i$ levels of the tree into a complete binary tree, having $2^i$ nodes on level $i$. A leaf of this complete tree is called *unused* if it is not a node (leaf or internal) of the original underlying Huffman tree. For example, any Huffman tree with 3 leaves has $n_1 = 1$ and $n_2 = 2$, so in this case we would get $m_1 = 0$ and $m_2 = 2$. In general, $m_{i+1} = 2(m_i + n_i)$.

The definitions of $n_i$ and $m_i$ can be extended to every subtree, and we shall denote $n_i[j]$ and $m_i[j]$ the values corresponding to the subtree rooted at vertex $v_j$. Without changing the above algorithm for bounded tables, one can change the definition of the external vector $k[j]$ to

$$k[j] = \max\left\{ i \;\middle|\; 1 - \frac{m_i[j]}{2^i} \;\geq\; \alpha \right\}.$$

Note that if $n_i[j] = 0$ for all $1 \leq i \leq t$, then $k[j]$ is at least $t$, even for $\alpha = 1$, so that the decoding block may contain more than a single bit in many cases. The idea is that $\frac{m_i}{2^i}$, the proportion of unused nodes on level $i$, can serve as a measure of the skewness of the tree and thus help finding a good balance for the blocksize.

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $n_i$ | 1 | 1 | 1 | 2 |
| $m_i$ | 0 | 2 | 6 | 14 |
| $1 - \frac{m_i}{2^i}$ | 1 | $\frac{1}{2}$ | $\frac{1}{4}$ | $\frac{1}{8}$ |

FIGURE 4: *Parameters of example tree*

Figure 4 brings the values corresponding to the root of our example tree. If $\alpha$ is chosen as $\frac{1}{4}$, $k[0]$ will be 3 and since the subtree rooted at $v_3$ has depth 1, we get $k[3] = 1$, so in this case, the algorithm will be identical to the bounded tables above. If $\alpha = \frac{1}{2}$, the two nodes for which tables are constructed are $v_0$ (the root) and $v_2$, and one gets $k[0] = k[2] = 2$, so that each table will have 4 entries.

## 4. Experimental results

The new methods were tested and compared on two natural language databases: the King James Version (KJV) of the English Bible, consisting of about 3.3 MB of text, and 36.5 MB of Wall Street Journal (WSJ) issues that appeared in 1989. Huffman codes were generated for three possible encoding models: individual characters, letter bigrams, and entire words. Table 1 displays some of the relevant statistics, including the estimated number of bits decoded in a single table access for the bounded tables method, as given in equation (1), using $k = 8$.

Table 2 compares the results on the word based Huffman codes. The sizes of the compressed files were 0.64 MB for KJV and 7.23 MB for WSJ. The column headed *Bit* corresponds to the regular bit per bit Huffman decoding. The next column brings the values of the *Partial decoding tables* of [1] described in Section 2 above. In the column headed *Binary Forest* are the results of a time/space tradeoff also suggested in [1], and the following columns headed *reduced*, *bounded* and *weighted* are the

| | KJV chars | KJV pairs | KJV words | WSJ chars | WSJ pairs | WSJ words |
|---|---|---|---|---|---|---|
| number of elements $N$ | 65 | 949 | 11669 | 77 | 2770 | 115136 |
| depth of tree | 21 | 20 | 19 | 18 | 24 | 22 |
| average codeword length | 4.23 | 7.53 | 8.80 | 4.56 | 8.08 | 11.20 |
| estimated $k-b$ | 6.10 | 5.06 | 5.08 | 6.06 | 4.94 | 5.01 |

TABLE 1: *Statistics of example files*

new methods suggested in Section 3. Note that the case $\alpha = 0$ is equivalent to the Bounded Tables method with no initial limit on $k$.

For each of the test database, the first line brings the maximal size $k$ of the block of bits that is decoded as one unit. The next line, headed *bpa* is in fact the average value of $k$ used during the decoding. It is the average number of decoded *bits per table access*, evaluated as the total number of such accesses divided by the size of the compressed file in bits. The last line, headed *RAM*, gives the size of the required auxiliary storage in MB. For the partial decoding tables, RAM has been evaluated as $N \times 2^k \times$ the number of bytes necessary for each entry, for all other methods, it corresponds to the actual space requirements for the tables in our implementation that has not been optimized. For instance, a similar non-optimized implementation of the partial decoding tables for KJV would require 51.4 instead of just 17 MB. Nevertheless, we see that the necessary space is of an other order of magnitude, especially for the weighted methods, and this reduction came without really affecting the decoding speed.

| | | *Bit* | *Partial decode tables* | *Binary Forest* | *Reduced tables* | *Bounded tables* $\alpha=0$ | *Weighted* $\alpha=\frac{1}{2}$ | *Weighted* $\alpha=1$ |
|---|---|---|---|---|---|---|---|---|
| KJV | $k$ | 1 | 8 | 12 | 8 | 14 | 12 | 12 |
| | *bpa* | 1 | 8 | 2.08 | 6.37 | 9.81 | 8.09 | 7.62 |
| | *RAM* | 0.21 | 17 | 0.31 | 8.7 | 0.47 | 0.29 | 0.25 |
| WSJ | $k$ | 1 | 8 | 14 | 8 | 16 | 14 | 14 |
| | *bpa* | 1 | 8 | 2.11 | 6.35 | 11.14 | 9.45 | 8.68 |
| | *RAM* | 2.1 | 197 | 2.5 | 34.1 | 3.9 | 2.7 | 2.3 |

TABLE 2: *Comparison of decoding methods*

# References

[1] Choueka Y., Fraenkel A.S., Klein S.T., Perl Y., Efficient Variants of Huffman Codes in High Level Languages, *Proc. 8-th ACM-SIGIR Conf.*, Montreal (1985) 122–131.

[2] Ferguson T.J., Rabinowitz J.H., Self-synchronizing Huffman codes, *IEEE Trans. on Information Theory* **IT–20** (1984) 687–693.

[3] Frederiksson K., Tarhio J., Processing of Huffman compressed texts with a super-alphabet, *Proc. SPIRE*, Manaus, Brazil, (2003) 108–121.

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

[5] Klein S.T., Skeleton Trees for the efficient decoding of Huffman encoded texts, *Information Retrieval* **3** (2000) 7–23.

[6] Moffat A., Turpin A., On the implementation of Minimum-Redundancy Prefix codes, *Proc. DCC*, Snowbird, Utah (1996) 170–179.

[7] Nekrich Y., Decoding of Canonical Huffman codes with Look-Up tables, *Proc. DCC*, Snowbird, Utah (2000) 566.

[8] Pajarola R., Fast prefix code processing, *Proc. IEEE ITCC Conference*, Las Vegas, Nevada, USA, (2003) 206–211.

[9] Sieminski A., Fast decoding of Huffman codes, *Information Processing Letters* **26** (1998) 237–241.

[10] Witten I.H., Moffat A., Bell T.C., *Managing Gigabytes: Compression and Indexing Documents and Images*, Van Nostrand Reinhold, New York (1994).